DISSERTATION

Long-term Workload Monitoring:

Workload Management
On Distributed OS/2 Server Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.Prof. Dr. Mehdi Jazayeri
Institut für verteilte Systeme
Technische Universität Wien

o.Univ.Prof. Dr. Günter Haring
Inistitut für Informatik und Wirtschaftsinformatik
Universität Wien

eingereicht an der Technischen Universität Wien,
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Günther Strasser
In der Hagenau 13/1, 1130 Wien
Matr.Nr. 8225024
geboren am 29. Mai 1964 in Wien

Wien, im März 2000

Kurzfassung

Im Laufe der letzten zehn Jahre hat der Begriff "Systems Management" im Bereich verteilter Client/Server Systeme zunehmend Bedeutung erlangt. In dem Maße, in dem billige PC Hardware und die dazu verfügbare kommerzielle Software Einzug in die Datenverarbeitung der Unternehmen genommen hat, sahen die IT Manager die Notwendigkeit, diese Technolgie und die damit verbundenen Probleme und Kosten unter Kontrolle zu bringen. Da nun leistungsfähige PC-Serverhardware einen nicht unwesentlichen Teil der Last in der EDV trägt, ist es notwendig, die wesentlichen Systems Managementdisziplinen, wie z.B. Kapazitäts- und Lastmanagement, analog den Methoden am Großrechner zu implementieren.

Während es aber am Großrechner mehr als vierzig Jahre Erfahrung mit diesen Disziplinen gibt, fehlt es im Bereich verteilter PC Server z.B. im Bereich der Lastanalyse und Kapazitätsmanagement sowohl an der Theroie als auch verfügbaren Werkzeugen. Für die transaktions- bzw. batchorientierte Software, wie sie meist auf Großrechnern läuft und die im Zusammenspiel des Betriebssystems, der Middleware-Komponenten und der Anwendungssoftware besteht, gibt es eine umfangreiche theoretische Grundlage, die zum Teil ihre Wurzeln in der Abbildung von Kommunikationskanälen hat, und eine Menge an erpropten Hilfsmitteln, die es ermöglichen, die Last an bestimmten Punkten zu messen und an andere Ressourcen zu verteilen. Alles, was es im PC-Bereich in diese Richtung gibt, stammt direkt vom Großrechner ab. In der Praxis hat sich jedoch gezeigt, daß die Anwendung dieser Methoden und Werkzeuge keine brauchbaren Ergebnisse liefert. Eine der Hauptgründe dafür liegt darin, daß die Betriebssysteme am PC - wir betrachten hier vor allem OS/2 - mit der Grundidee eines interaktiven und "ungeordneten" arbeitens mit einem Benutzer vollkommen anderen Gesetzen gehorchen als die sehr strikten Großrechner. Dazu kommt, daß verteilte Systeme inzwischen extrem kompliziert sind und aus so vielen Komponenten bestehen, daß es kaum möglich ist, hier "manuelle" Methoden der Messung, Modellierung und Berechnung einzusetzen.

Diese Dissertation zeigt den Wert und die Notwendigkeit der Langzeit-Lastüberwachung auf und beschreibt die wesentlichen Probleme, die damit verbunden ist. Es wird aufgezeigt, wie diese Probleme gelöst werden können und welchen Nutzen man aus den gewonnen Informationen ziehen kann. Das Hauptaugenmerk liegt auf dem Werkzeug SRVMONPM, das diese Erkenntnisse im Rahmen einer Reihe von Programmen implementiert. Die Ergebnisse zweier Fallstudien werden grafisch aufbereitet dargestellt. Auf Grund der Erkenntnisse aus den beiden Studien und anderen Erfahrungen, die im Rahmen der Arbeiten mit dem Werkzeug gewonnen wurden, kommen wir zu dem Schluß, daß ein solches Werkzeug die Kontrollierbarkeit und Steuerbarkeit eines komplexen, verteilten Systems verbessert, daß es aber eine Reihe von Problemen gibt, die nicht einfach durch ein Werkzeug gelöst werden können.

Der wesentliche Beitrag dieser Dissertation besteht im Entwurf und der Umsetzung einer Methodik zur Erfassung und Analyse umfangreicher Systemparameter aus verteilten Systemen über einen langen Zeitraum hinweg, die es ermöglicht, solche Systeme, die aus sehr vielen heterogenen Teilen bestehen, in ihrer Gesamtheit zu erfassen und zu durchleuchten. Neben statistischen Standardauswertungen wurde ein spezieller Algorithmus entwickelt, der es ermöglicht, ohne Kenntnisse oder Annahmen über das beobachtete System Korrelationen zwischen beliebigen Systemparametern zu finden. Dies ist deshalb wichtig, weil es auf Grund fehlenden Wissens bzw. Information praktisch unmöglich ist, für ein reales System die notwenidgen Voraussetzungen klassischer Methoden zu erfüllen. Die daraus resultierende Methode arbeitet frei von Annahmen und ist auch für die Einbindung jedlicher weiterer Systeminformationen offen.

Table of Contents

Table of Figures

Abstract

Over the last ten years the term "systems management" has become increasingly popular in the area of client/server computing and distributed systems. As low-cost hardware and commercial software for the PC were populated all over the enterprise, IT managers saw the need to obtain control of the technology as well as the costs associated with it. Meanwhile powerful PC server machines have come to bear a good deal of the workload of the IT infrastructure. As with mainframe-based systems, therefore, capacity and workload management have become fundamental disciplines within systems management of client/server environments.

While there are forty years of experience for mainframes, capacity management for distributed PC servers lacks both theory and software. Transactional and batch oriented mainframe software - which means the combination of operating system, middleware and application software - are equipped with a profound theory in modeling the processes and rich set of existing and proven tools for measuring and distributing a workload and for planning for the near future. Everything that exists in the client/server area is a direct extension of a similar method or tool from the traditional way of computing. Analysts find it hard to apply these on PC servers, however, mainly because PC operating systems origin from highly interactive personal (meaning individual) computing, which follows different rules. In addition, today's systems become so complex and consist of so many components that it is practically impossible to apply methods of traditional computing paradigms.

This dissertation discusses the value and necessity of long-term monitoring and shows the many problems associated with that task. It explains how the problems can be solved and how one can benefit from the recorded information. The main focus is on SRVMONPM as the tool set that implements the method. The results of two case studies in real commercial environments are presented. Based on the case studies and the experiences from a number of people who applied the tool set in their work we conclude that SRVMONPM improves the controllability of distributed systems, but that there are still a number of problems that cannot be solved by a sole method or tool.

The contribution of this dissertation is the introduction of a general method (implemented as a number of software tools) that enables an analyst to monitor and collect information about the dynamic nature of distributed PC-based server systems from an unlimited number of heterogeneous components, which are distributed over a (potentially large) number of server machines, over a long period of time. The information is compiled and recorded. In addition to common statistical methods, a special mechanism is supplied to detect correlation between any attributes that were recorded for the components. The method is based on the assumption that there is no knowledge about the details of interdependencies and relations between any of the systems involved. It can therefore be applied in different circumstances, and it is open for the inclusion of new information.

Preface

About The Author:

Günther Strasser was born in 1964 in Vienna, Austria. He attended the Technical University of Vienna (TU Wien) from 1982 to 1987 where he studied Computer Science (Informatik). His areas of special interest were computer languages, compiler and computer graphics.

After graduation with the degree of "Diplom-Ingenieur", he worked for two small software development companies in Vienna and Lower Austria. There he was concerned with the development of special-purpose CAD-systems on UNIX-Workstations (from Sun Microsystems). From 1988 to 1989 he completed his military service in the Austrian Armed Forces and then joined IBM in the area of GUI development under OS/2.

After two years in the development the focus of the department shifted to the management of large PC- and workstation based networks. His new tasks were consulting in the area of distributed systems management and the design of infrastructure and systems management solutions. In the course of that work he came across a number of problems concerned with estimating, assessing and constructing server topologies in complex client/server environments.

The lack of adequate tools and the absence of any documented material about server usage and load triggered his interest in this topic. It led to the implementation of a load monitor and analysis application and studies of load data in typical business workgroup environments.

Since 1996 he was more involved with project management in large infrastructure and software development projects.

Trademarks

OS/2™ is a trademark of IBM
Windows™ is a trademark of Microsoft.

1. Introduction

The ever growing complexity of today's computer systems makes it more and more difficult to size a new system and to predict the behavior due to a changing workload or the result of tuning activities. While the classical workload was rather well defined and measurable - consisting of batch-jobs, transactions and interactive applications, running on the same system - now a wide range of software programs exists on PC server systems that are not easy to control and manage. Often it is unclear what the workload on such systems is and how it can be measured. There are no interactive applications, but the server communicates with a number of clients. There are no batch jobs as there is no batch control facility, but there may be programs that are scheduled due to an event (for example a timer). Often, there are no transactions[1], but there are service requests that have to be handled by the server software. The difference is that a transaction has a name and is associated with a program/module/code and required resources. The system (operating system and transaction monitor) and, consequently, performance monitoring tools and workload managers have control over the transactions. Transactions are very "visible". On the other hand, the communication between a server software and its clients is private, often unpublished and handled as an internal detail of the software. A network monitor (or protocol analyzer) can detect how many messages and how much information flow over the network, but from this information it is not possible to derive the beginning or the end of a service request, the number of requests, etc. Due to the multi-threaded (multi-tasking) nature of server software, there is no "end of a request" that would be comparable to a mainframe transaction. Incoming messages are forwarded and a client may receive an acknowledgment long before the processing of the request is completed.

This paper presents the work being done

1. to **explore** what actual **information** is available **on-line** from PC based server applications. OS/2 was chosen as the operating platform. Information includes all data about workload, performance, status, etc. that can be queried from an application.

1. to **implement** a robust and efficient monitoring system that supports unattended **7x24 operation**, **long-term recording** of data, and **smooth integration** of **heterogeneous data** from different sources. It is open for the **inclusion** of new data sources.

1. to **analyze** and **present** the recorded information in such a way that an infrastructure designer can actually work with the data.

1. to **perform** a number of **case studies** in commercial environments in order to generate some **material** that **documents actual workloads** and the dynamic change of important parameters in the monitored system. This information would be useful for all persons involved in client/server infrastructure design.

1. to **detect** automatically **relations or dependencies** between the many parameters. The theory, to build a network of related parameters and to form a machine-build *tabular model* from that, is validated. It was found impossible to build a complete network that could be used for simulation or analysis tasks.

Please note: a table of "terms used in this paper" forms part of the appendices at the end of this thesis. Please refer to it for definitions of terms appearing for the first time.

[1] Transaction monitors exist for OS/2 and Windows NT. Usually, the transaction monitor is part of a three tier architecture with a major transaction system located at a mainframe.

1.1. Assumptions and Research Target

As described later in this paper, part of the problem in designing a new infrastructure or adapting an existing one is the total lack of reliable information about the current system and the way the system is used by the user community at the time an architect is confronted with a real-life situation. This does not mean that there is not some information at hand - but usually, even "simple" metrics like number of server machines or number of users are wrong: the documents are out-dated and have not been updated during the past years, other departments have changed the system and have not notify the IT-department[2]; there is no central user administration and the user and access databases that are part of server software systems do not reflect the real number of users, and so on.

Assumptions

Because this is a very common situation, the following assumptions became fundamental for this thesis:

1. There must be a way to collect and record reliable information about the system over a long period of time - at best the whole life-time of the system.

1. A distributed system is a complex construction consisting of many subsystems and components. There are a lot of dependencies between them. Some of them are well defined, documented and, therefore, known. Many of them are not. If such data is available it must be possible to find “hidden” dependencies.

1. It is unhelpful when one subsystem's collection and presentation of workload data is incompatible to other subsystems. It must be possible to build a system that collects and consolidates information from many subsystems. This consolidated information can provide better insight into the whole system and its components.

1. Snapshots or short-term data (the last n minutes) cannot be exploited in a reasonable way. Long-term measurements and suitable analysis tools are necessary and it must be possible to do this on toady's PCs without the need of expensive mainframe-resources.

1. With this data and the knowledge about the dependencies it will be possible to build a (data-driven) model engine that can make use of the data and help in evaluating system behavior under changed conditions.

Research Questions & Targets

The most important question concerns the actual workload patterns on real commercial (office) client/server systems. What are the services that are really used, how many people and workstations participate, when and in which intensity are services requested? What information is available or could be retrieved (in contrast to what is offered by existing tools)? Everything else in the process of creating an IT architecture is based on that knowledge.

Secondly, assuming it would be possible to gather and record detailed workload and usage information about a system, the question arises how one can benefit most from the data. One of the things that an architect needs in his job is a means to predict the effects of changes to the environment (e.g. introduction of a new application, more users, more devices) or to the system itself (e.g. more memory, faster servers, etc.).

[2] Another "challenge" of the client/server world, as it is easy and - when involving hardware - cheap to modify existing or add new systems.

The classic approach to predicting the behavior of a system is the construction of an **analytical** model that can be used to check the consequences of changes to parameters or components in the model. Such a model consists of a number of equations that can be solved. There is a lot of literature about analytical models, e.g. [5] and [7].

The problem with analytical models is that it is extremely difficult to build one. The analyst needs a thorough knowledge of the details and internals of the target system. It is hard to prove its quality and - the model is always wrong. Menascé states in [7] that *the best model of a system is the system itself*.

For each new technological development and each new software system new modeling techniques have to be found. It takes much longer for someone to find a suitable model than it takes to build the technology. Therefore, in general it is not possible to build models for current client/server systems (though solutions for specific problems exist). While the area of classic terminal and batch oriented, single or eventually multiple CPU mainframe processing is very well understood, today's networks with a huge number of more or less independent processing units that communicate to one another and access service machines with different usage patterns, are a magnitude more complex and a field of ongoing research work.

Analytical models have other drawbacks. Usually they are so computationally intensive that it is not possible to solve them on a computer system at reasonable costs. To circumvent that problem, approximation algorithms exist that provide less precise but often sufficient results (usually min/max boundaries). When modeling a system the analyst needs to know a lot of intricate details about this system. Considering the examples in books and papers the attentive reader will notice what explicit and implicit assumptions are made about hardware and software. Models are based on measurements of some low level, hardware and operating system related metrics. Often, it is assumed that one has detailed insight into the system under consideration (like the developer of the software may have). In practice one does not have the required knowledge about the hardware or about the server software, firstly, because some of it may be an intellectual property of the producer and cannot be published and secondly, because the developers never take the time to describe the internals of a product.

The second approach to the problem is the use of **simulators**. Again the analyst needs a lot of knowledge about the internals of the system. With the use of tools or programming a model of the system is built and the simulator may answer questions about performance and behavior. The same problems exists: technology progresses faster than simulators and one does not have the details needed to build a valid model.

Both approaches tend to create models that match what the analyst would like to have. Because it is not possible to formally prove the "correctness" of the model, in each case the model must be calibrated. If not carefully done this can easily jeopardize the whole procedure.

This paper is about obtaining more knowledge about a system under consideration. Our approach consists of creating a sound basis of measured data and deriving dependencies and relations from that database. In order to do this we extend the term *resource*. While most literature deals only with hardware resources (CPU, memory, I/O channels, disk subsystems), we have to consider the resources that are provided by the server software (application software). Every server software provides resources that are defined by the software (e.g. the number of possible connections, the number of open file handles, the number of threads in a system, the number of database index entries in memory, the number of network buffers, etc.). This can be visible to the client (*external resource*) or it can be a "private" detail of the software (*internal resource*). Very often, an administrator has to size these resources (when limits are defined in configuration files), but has no idea what values to use. At the University of Carleton a number of projects are under way that address the problem of software resources that are essential to the behavior of a system [31].

To overcome these problems this paper presents the idea of using the given system itself (instead of a model). By monitoring and recording all available parameters over a long period it should be possible to "understand the behavior" of the system. Eventually it may be possible to detect relations between resources to build a **tabular** model that can be used for prediction and simulation. A tabular model would be a collection tables. Each tables sets a number of parameters into a relation and makes it possible to look up a value when the other parameters are given. Chapter 7 explains the idea in more detail. However, we have not succeeded in creating a complete set of such tables.

With this idea as a basis, a lot of other questions come to mind:

- Which relations exist between the uses of different resource on the same or different servers by applications running on network clients?
- Is it possible to collect and process load information of an enterprise-wide client/server network with a completely OS/2 based solution?
- What kind of data can be retrieved?
- What value does this data have for a LAN administrator?
- What value does this data have for an infrastructure planer?
- What value does this data have for an client/server application designer?
- Is it possible to derive any feasible assertion about the future developments of a system by looking at the history of load data?
- Is it possible to implement automatic system control based on feedback to the current workload data?

This dissertation discusses possible ways to find answers to the questions above based on the information that is built by long-term monitoring.

1.2. Definitions

The following terms are used throughout the document and it is important to understand how they are used in this context. Several definitions are derived from [7].

Throughput

is the rate at which requests are serviced. A more mathematical definition is: The average throughput is the quotient of the number of requests finished and the length of the observed time interval [4].

$$D(t) = \frac{\Delta d(t)}{\Delta t}$$

(Theoretical) Capacity

is defined as the maximum rate at which a computing system, component or resource can perform work. Thus, capacity is the upper limit of throughput.

Effective Capacity

is the largest throughput at which response time remains acceptable. This implies that the amount of acceptable response time is specified. In fact, responsiveness limits the amount of effective work processed by a system [10].

Intensity

Intensity is defined as the number of transactions (*service requests*) generated by each concurrent client [32], [33]. This definition refers to the need to distinguish *active users* of a system versus *concurrent users*. While it may be easy to measure the number of

active users, concurrence and intensity are much more important for an analyst. However they have to be derived from other data.

Resource

is a component of the computer system that is necessary to satisfy a service request.

We have to distinguish **external resources**, which are visible to other applications or users and are provided as a service to others, and **internal resources**, which are required by the server software itself to perform certain tasks, but which are not visible or accessible to other processes on the host computer.

In reality one of the problems of systems administration is that every server application provides services which require certain resources. These resources may be provided by other components (e.g. the operating system or the network adapter, etc.) or they may be introduced by the application itself (e.g. number of available communication handles, memory buffers, etc.). Often it is necessary to configure the number and/or sizes of allocated and provided application resources and these numbers are fixed for the lifetime of the application.

A client who uses services of a server application consumes more or less directly resources of every software layer over a period of time (see Figure 1). These resources are not "consumed" or destroyed due to their usage. Therefore we can assume that a resource becomes available again after a service request has been completed or a client has given up a service.

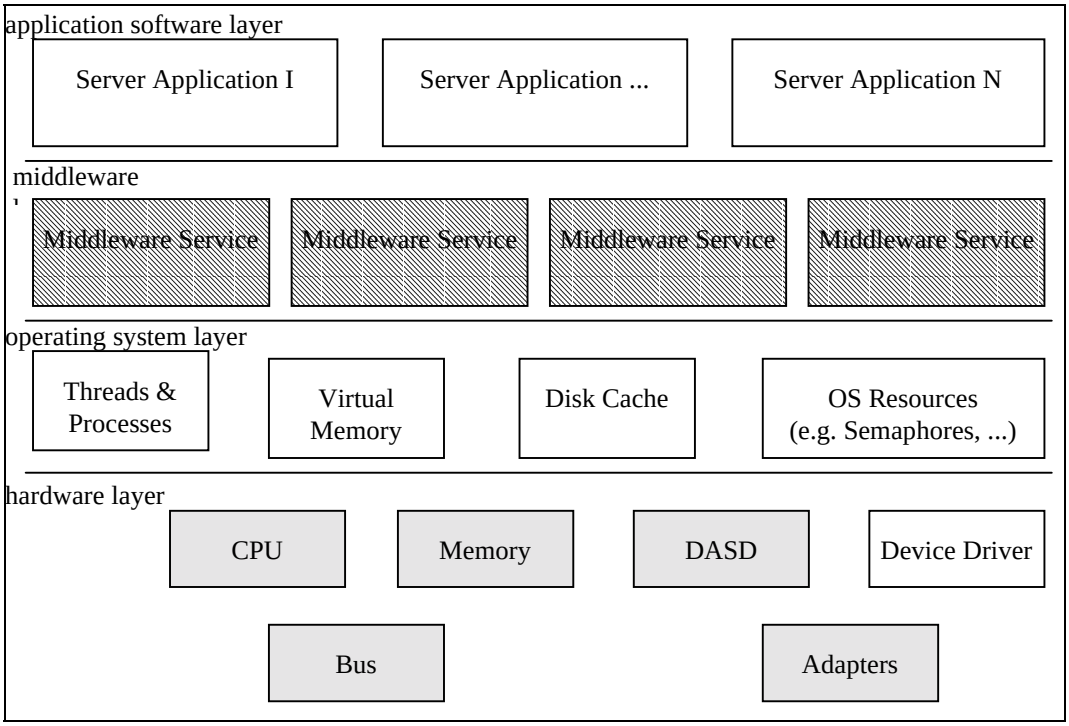


Figure 1. Layers Of Provided Resources

Workload

of a computer system designates all the processing requests submitted to a system by the user community during any given period of time [8]. Broken down to a server application, its workload is any kind of consumption or temporary use of its resources due to service requests that have been submitted to it.

Monitor

is a tool used for measuring the level of activity of a computer system [8]. A *software monitor* consists of routines inserted into the software of a computer system with the aim of recording status and events of the system [7], [14].

Although it is necessary to have a piece of software that stores information about metrics and provides an interface to access the information, our approach to building a monitor assumes that it is **not** possible to insert code into a software system and that the monitor has therefore to use existing sources of information. Thus the monitor is not part of the software that is to be measured. It was an essential part of this work to find sources for a monitor in the area of (commercial) OS/2 systems and applications.

1.3. Background

This work was triggered by the fact that it is extremely difficult to plan and implement the infrastructure of today's client/server solutions. Someone involved in client/server infrastructure design will come to the conclusion that some of the tools necessary for the proper analysis and design of such an infrastructure are missing.

Quote from an article from *Software Magazine* [Korzeniowski] (quoted from [1]):

"Client/server capacity planning [is] mostly a black art today. Capacity planners who have applied the discipline [of capacity planning] to host-centric systems find it extremely challenging to apply [it] to networked systems because the same types of tools are not yet available. ... Often, a manager guesses what computer, node and link capacity to install and then waits for the result."

One of the main problems is the lack of precise information about the environment and the behavior of users and applications. Existing tools commonly focus on the lower network layers or on a certain application but there is no way to get an overview of the whole picture and drill down for more details where it seems to be important.

Performance Analysis

Figure 2 gives an overview of the major techniques that deal with performance analysis in general [4].

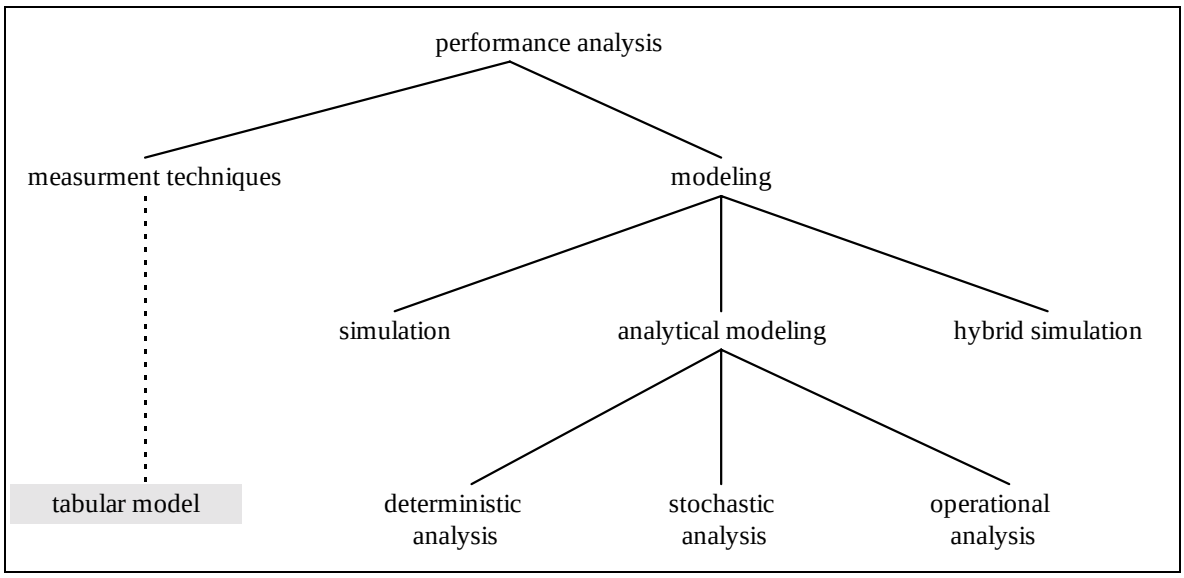


Figure 2. Performance Analysis Techniques

The term "tabular model" has been added to the picture. Chapter 7 explains the idea of the tabular model and the way it is intended to create it.

Workload Analysis

Workload analysis is a technique to assign the set of all possible service requests to a limited number of groups. Workload characteristics are defined for each group. The attributes of the group represents its members in later steps. Workload analysis is a part and corner stone of performance modeling. Model calculation is reduced to a number of workload classes.

Systems Management

Today, people involved with systems management are faced with large numbers of desktop systems spread over the country, all connected via some kind of communication device forming a network of heterogeneous and relatively uncontrolled nodes.

In contrast to a centrally managed and well defined host system where a given set of transactions of known size formed the biggest part of the workload, end-users are free to use shared resources on the network, execute programs, access distant services, exchange data, etc. and it is hard to determine what is going on in all the systems[3]. Thus, we are faced with a shift from well defined classes of workload (consisting of known batch jobs and transactions) to individual, uncontrolled and hardly predictable service requests which originate from applications that execute on end-user workstations.

An extra layer of complexity comes with the use of many different products within an environment. While single products may sometimes offer some support in monitoring resource consumption and workload of their own services, they are neither integrated into some sort of common solution nor do they provide clues on how they use system resources and interact with other components. Besides that, most applications do not even offer this kind of support.

While the environment becomes more complex and difficult to handle, there is great pressure to reduce the overall costs of the IT systems. It is also expected that the operation will become cheaper due to the decline in hardware and software prices. Often this means less personnel, tight timelines and stringent budgets. One cannot expect to have much time to deal with a certain tool and its functionality. It must be simple to operate and its value for the work of the IT people must be obvious. Somehow it seems that a profound workload analysis has become an academic discipline while in the field there are no people who can handle the task.

In absence of any useful information about how the system behaves it is very difficult to track down problems and to plan future changes or extensions.

The only exception to this situation is the network layer itself. There are several products available which provide the ability to monitor network traffic and react on certain conditions. Many products are focused on special hardware, especially in conjunctions with network infrastructure hardware like routers or gateways. Such devices are capable of collecting workload and performance data and handing them over to an application which controls their operation.

Long-term Monitoring

As far as we know there have been no long-term (spanning a period of a year or more) studies of workload in a real-life commercial environment, either in a classic mainframe environment or in a distributed client server environment. Therefore it does not come as a surprise that there is not much data available that could be used as the basis of one of the techniques mentioned above.

[3] With distributed objects on the horizon the situation will become worse.

1.4. Related Work

This chapter presents a number of different approaches to workload monitoring both with an academic background and from the IT industry. Related topics like accounting and application instrumentation are briefly covered because they may offer valueable input for collecting, analyzing and understanding workloads. Because there has not been much focus on OS/2 and because the problem of monitoring load and a distributed network of nuzmerous heterogeneous computer systems we do not limit our references to OS/2 specific work but we try to give a general overview. At the web site of the Computer Measurement Group [9] discussions and standard proposals concerning measurement and monitoring can be found.

Hardware Monitors

A hardware monitor is a measurement tool that detects events within a computer system by sensing predefined signals [7]. It focus on hardware issues and is not able to provide data about server software. Hardware monitors are not discussed any further. The rest of this paper deals with *software monitors*.

(single machine) Software Monitors

Software monitors record any information that is available to programs and operating systems. That makes software monitors a powerful tool for analyzing computer systems. The IBM Resource Management Facility (RMF) and the Unisys Software Instrumentation (SIP) are examples of software monitors that provide performance information. Note that usually such tools are (integrated) parts of the operating system and therefore are focused on metrics relevant to the operating system.

Software monitors may operate either in *event trace mode* or in *sampling mode*. If the event rate becomes very high, the overhead of event tracing may become unbearable [16]. Ref. [7] states that when compared to event trace mode, sampling provides a less detailed observation of a computer system. Although the monitor that was implemented as part of this work operates in sampling mode, no data are missed. Many events are captured and counted by the software that provides the workload data. Therefore a sampling monitor has access to that kind of information too.

Monitoring Infrastructure

Recently, the Distributed Management Taskforce (DMTF) released its Common Information Model (CIM) [26]. CIM is a general, object-oriented, extensible description of computer systems and their resources and it contains guidelines on how to extend and implement it. This addresses the problem of systems management tool providers with heterogeneous systems. The idea is to establish a commonly accepted, standard systems management interface between the providers of hardware, system software and application software. Systems management tools should then be able to use the interface to obtain available data and use provided functionality and implement management functions on top of that without the detailed knowledge about products required today.

Quoted from the DMTF web site about CIM: "CIM is a conceptual model that is not bound to a particular implementation. This allows it to be used to exchange management information in a variety of ways; four of these ways are illustrated in

Figure 3. It is possible to use these ways in combination within a management application.

As a repository, the constructs defined in the model are stored in a database. These constructs are not instances of the object, relationship, and so on; but rather are definitions for someone to use in establishing objects and relationships. The meta model

used by CIM is stored in a repository that becomes a representation of the meta model. This is accomplished by mapping the meta-model constructs into the physical schema of the targeted repository, then populating the repository with the classes and properties expressed in the Core model, Common model and Extension schemas.

For an application DBMS, the CIM is mapped into the physical schema of a targeted DBMS (for example, relational). The information stored in the database consists of actual instances of the constructs. Applications can exchange information when they have access to a common DBMS and the mapping occurs in a predictable way.

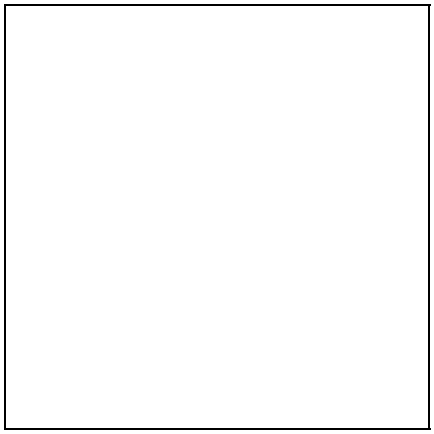


Figure 3. Four ways of using CIM

For application objects, the CIM is used to create a set of application objects in a particular language. Applications can exchange information when they can bind to the application objects.

For exchange parameters, the CIM expressed in some agreed-to syntax is a neutral form used to exchange management information by way of a standard set of object APIs. The exchange can be accomplished via a direct set of API calls, or it can be accomplished by exchange-oriented APIs which can create the appropriate object in the local implementation technology."

Figure 4 shows the structure of the meta schema of the CIM. It describes the major elements that can make up a model of a system.

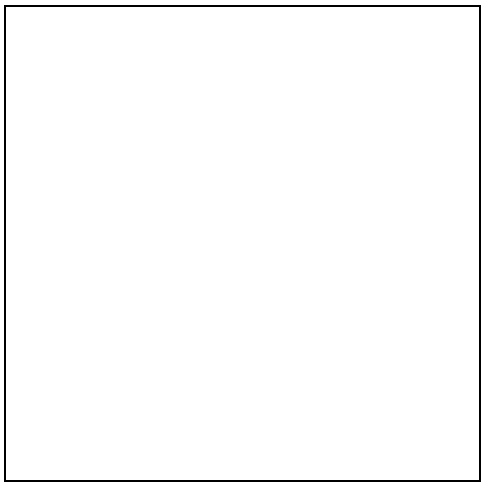


Figure 4. CIM meta schema structure

The Windows Measurement Instrumentation (WMI) is Microsoft's implementation of CIM. It contains a subset of the full model and extends this subset with Windows NT specific classes and properties. The Tivoli Distributed Manager for NT [34] from Tivoli is the first product to make use of WMI to monitor performance data. While WMI is an

add-on for the current version of Windows NT, it will be fully integrated in the next version, Windows 2000. In this version Windows management tools use WMI to perform their functions.

While the idea is to separate management functionality and system details, WMI shows that CIM heavily depends on platform specific implementations and that different providers are not likely willing to open their proprietary systems. Different implementations may work in very different ways and may not be compatible to one another.

Regarding measurment and monitoring CIM opens a way for server applications to define their resources, services and dependencies. They should contain *provider code* that is able to create and feed instances of CIM objects. At this time there is no OS/2 implementation for CIM underway therefore the role of CIM for an OS/2 based system is more theoretically. The WMI implementation is limited to the operating system and we will see in the future whether server software suppliers and tool producers adapt to the CIM model of systems management.

Instrumented Distributed Applications

The following two references address the problem of lack of control about distributed applications. While common measurement tools focus on a single server and monitor hardware related or operating system specific attributes the overall performance of an application that is distributed over a number of machines remains in the dark. They describe systems where access to the source code is assumed. The application code gets instrumented, that is code is inserted to perform required functions for the monitoring system. Note, that our own work is based on the assumption that the application code is not available and cannot be altered in favour of the monitor.

Application Response Monitoring (ARM)

A special problem in the management of distributed applications is addressed by the ARM [36] architecture and related products. ARM is meant to fill the gap in the classical monitoring of distinct machines. It will give the system administrator and the application developer a picture of the application's performance from the user's view.

First, the term *transaction* is reintroduced to distributed client/server applications. The developer must define distinct transaction types. During the execution of the program transactions are performed by a user. The response time of each transaction is measured. The correlation application monitors the event flow and calculates response times and statistics for each transaction type. In order to do so the application code must be altered to insert calls to the ARM API. All parts of the application have to signal when they start with their part of a transaction and when they are done with it. Figure 5 contains an example of an application that consists of a client part and two server parts that are placed on different server machines.

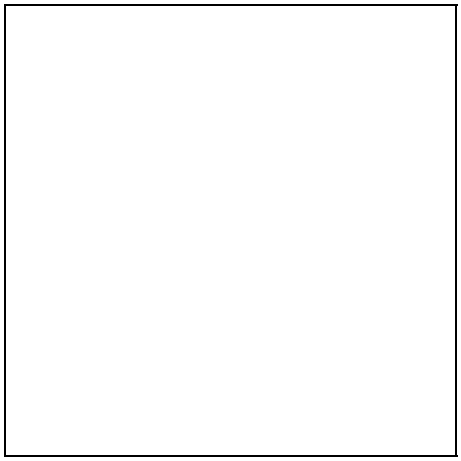


Figure 5. ARM notification flow

When the client starts a new transaction it notifies this to an ARM agent and receives an unique transaction id. It is the responsibility of the application to hand on this id to other parts of the application. The *correlation application* in the figure is part of a monitoring tool that receives the notification from the distributed system. It will receive many events and has to correlate those notifications that belong to the same transaction. When server B receives the request for the transaction it notifies its ARM agent providing the transaction id. This is done for every request to another part of the application. When server B has finished processing of the transaction which included all calls to other servers it notifies the ARM agent about the end of its part of the transaction. Finally the client notifies the ARM agent that the whole transaction has been completed.

The correlation application collects this information and may be able to generate statistics about the response time related to an application, user, server, LAN segment, etc. Every transaction and every request to another part of the application results in some notifications to the ARM system. This generates a considerable amount of network traffic. Therefore the ARM paper suggests to limit the use of ARM to a few machines in the network and to problem situations.

If applications are instrumented for ARM it would offer some insight into some significant metrics like

- overall average response time per transaction class,
- average response time per transaction class per application part (server),
- network related delays,
- number of transactions per user, per client and per server (application related workload per server),
- request distribution within each transaction class.

An interesting detail of the ARM implementation is that the part of the ARM agent, that is used to receive notifications from an application, executes in the address space of the application (i.e. is part of the application) and has to do some IPC (inter process communication) to forward the information to the ARM agent at the local machine. This IPC is rather expensive and adds to the response time of the application.

ARM focuses on application response time. Considering general workload monitoring ARM could provide valueable data that has to be correlated with other workload and performance data from other sources.

Making distributed applications manageable through instrumentation [37]

This approach goes beyond mere retrieving information about resources or applications but tries to address the management of distributed applications. That means that a remote (central) systems management tool may apply functions on the application.

Again it is assumed that the source code of the application is available and can be altered to fit to the needs of the system.

The authors introduce the terms *manager*, for the central component, *management agent*, for code that collects information, and *managed object*, which represents actual systems or resources. This is very similar to the terms used in this work. Their instrumentation architecture contains *sensors*, which encapsulate management information and are responsible for collecting, maintaining and processing information, *actuators*, which encapsulate management functions which exert control over a managed process (application), and *probes*, which is code, that is inserted into the application to faciliate the applications interactions with sensors and actuators.

A systems management application may interact with the agent, not knowing the technology that is behind it, for example, the management system may be based on SNMP. The problem is the insertion of probes into an application. The authors propose the following *probe points*:

- Entry points to processes (and threads, if we extend the concept to OS/2)
- Exit points (when processes or threads end)
- Inter-process communication (whenever one process communicates with another process)
- Operating system or middle-ware service invocation
- Exception and signal handlers
- Custom points (other points the developer thinks may be useful for managing the application)

Because it would be very difficult or impossible to insert these probes manually the authors propose a four level implementation that includes instrumented system libraries and compilers (where the compiler already inserts required probes, IDL instrumentation, function or class wrapper instrumentation and hand coded instrumentation (for custom probe points). As this project focuses on possibilities on how to support (semi-automatic) instrumentation of distributed application there is still no experience with an operational system with an application instrumented at the proposed detail.

Accounting Systems

These are tools primarily intended as means of apportioning charges to users of a system [15], [16]. They are usually an integral part of most multi-user operating systems, e.g. IBM/SMF (System Management Facility), VMS ACCOUNTING, UNIX/sar (System Activity Reporter) [17], SNI ACCOUNTING/RAV (Rechenzentrum-Abrechnungs-Verfahren). These examples are all mainframe-based tools. There has to be a central user definition and a central accounting system and the operating system associates each use of a system resource to a user account and notifies the accounting system. Tasks the are shared by many users (like a DBMS) are either not traced or are billed to a separate account.

For distributed PC-based servers there is a demand for accounting systems too. Although SRVMONPM provides some of the needed functionality, it is not an accounting system. The main obstacle to this respect is that many software servers on OS/2 (and Windows NT) implement their own user administration and security system(s), and neither the operating system nor a software monitor knows, who is responsible for each request or resource consumption. Moreover, there is no direct accounting for resource usage per request.

Program Analyzers

Program analyzers are useful tools for understanding the performance behavior and bottlenecks of a software product particularly in software development. Today every software development package includes such a tool. The compiler or translator/ interpreter inserts monitoring hooks into the machine or intermediate code. A program

analyzer can use this hook to monitor and analyze the system. Examples are the IBM Performance Analyzer, which is part of the Visual Age programming environment, or the UNIX *perfmon*[4] tool [18]. On the mainframe, such tools exist for DB2, IMS and CICS.

Remote Monitoring based on Remote Command Execution

Projects that deal with performance or workload balancing (for example, the HiCon project [3]) have to retrieve information about the load on a system or a software component. If a central component has to monitor a number of different machines it has to access this information remotely. There is nothing like a standard way to accomplish this. Projects that are UNIX based very often rely on the UNIX feature of remote consoles and remote command execution. Doing so UNIX commands like "ps" or "vmstat" run on the remote machine and retrieve information about the remote host. Their textual output is captured and sent back to the requesting machine. There the text output is processed.

Mainly academic projects and purely UNIX oriented management applications make use of that. This approach has a number of drawbacks:

- The monitor is limited to commands that come with UNIX. They are operating system related and it is not possible to retrieve information other than what is offered in a console.
- It is very expensive in terms of system and network resources. The requester has to establish a connection and a remote console each time a bit of information is retrieved. A considerable number of network messages is exchanged during that process.
- It requires the existance of user authentication information at each host machine. Besides the effort to establish and maintain that information it may open up a security hole because access permissions are created that are not associated to a user.
- It is not possible to synchronize data retrieval over a number of machines.
- Remote consoles do not exist on most OS/2 machines (utilities exist that would do it) and most relevant information is not accesible via text commands. OS/2 and all other PC operating systems are much more GUI oriented and do not lend themselves to this way of information retrieval.

Commercial Systems Management Products

When the work on SRVMONPM started in 1994 there was no remarkable systems monitoring tool available for OS/2 or any other Intel-based operating system. The situation did not really improve since then, due mainly to the decline of OS/2 in favor of UNIX-based server OS (e.g. LINUX) and Windows NT. Nevertheless today all solution provider of systems management packages count monitoring as a core functionality of their packages for distributes PC environments.

The term *monitoring* (or monitor) is used with different meanings, or at least, has very different implementations. There are several approaches which differ in their target environments and problem focuses.

Firstly, there are tools which aim at a single machine or at very small environments, which consist of one or two servers and some dozens clients. Tools of this category focus on hardware-related resources. By accessing the hardware or by using hidden

[4] The author developed an OS/2 version of this tool that was published as part of the IBM EWS program.

(unpublished) operating system hooks they provide the user with detailed information about the current state of the machine or a device.

The most simplest tools display a single snapshot of an item, e.g. in a GUI control that mimics a meter. More sophisticated tools provide current and recent data over a certain time interval and display them in a graphical window.

Such tools are easy to use and can give valuable information when looking for a certain problem or bottleneck during testing or problem evaluation. Via a log file or entries in a log database the information can be accessed through other programs, e.g. a timetable can be loaded into a spreadsheet.

The disadvantages are that one is unable to get information about high level services (e.g. a shared file system or a transaction service), that there is no long term logging and that the evaluation has to take place directly at the machine which is being monitored. The information is directly related to the activities on the machine and therefore no relation to other services or machine can be established. None of these tools can be used remotely from a central site.

Several work-arounds for some problems exist: e.g. add-on tools collect log information and store them in a "central" database within the LAN. Another add-on tool can collect LAN-based databases and join them onto a central host database. Simple analysis can be performed on this data but with their focus on hardware performance it is not possible to get a clue about what has been going on in the system.

Examples: System Performance Monitor (SPM/2)
TME 10 Netfinity (see www.tivoli.com)

Secondly, several network managers have been extended toward system monitoring. This group of applications are UNIX based. Their capabilities are based on either SNMP or the remote command execution of the UNIX operating systems. In both cases the manager sends a request to the target system, the target system processes the request - either the SNMP agent (see [27]) builds a MIB or the target UNIX executes a command - and receives a textual answer (results on *stdout*). The answer is interpreted as a numeric value and the values are collected and displayed.

This approach is rather simple and basic to a (graphical) network management application, which is already able to monitor network load. Unfortunately it did not work well (for a PC based client/server environment):

1. Both solutions suffer from the fact that the network traffic increases tremendously with the number of monitored resources. Note that the number of server PCs is usually much larger than the number of (expensive) network hardware boxes and that each server may offer many resources that may be of interest.

   One work-around is to use a very long update interval which in turn decreases the quality and meaning of the resulting data. In addition, due to their user interface and difficult setup procedures, the tools do not lend themselves to monitoring a large amount of resources.

   Consequently, monitoring is restricted to use with certain resources (≈ systems), and this only when necessary (e.g. in case of problems).

1. If *SNMP* is used to access load or performance data, the solution is IP based. It will not reach machines which do not have an IP stack - and that is likely in the world of Intel-based servers[5]. A possible work-around is to install an IP stack and an SNMP

[5] E.g. many Token-Ring LANs use NETBIOS or IPX stacks.

agent at all server machines. The question is whether or not a company is willing to buy into it because of monitoring.

1. If *remote command execution* is used the solution is limited to the UNIX world. Moreover it opens a possible security hole because the monitor may execute any command. One has to plan and restrict possible commands very carefully. For OS/2 based servers a theoretical work-around is to write a proxy agent which does the same work on the OS/2 machine and reports the result back to a UNIX workstation. If the IBM LAN server NOS is in use, this function is available with the correct setup. In contrast to the UNIX operating under OS/2, there are no corresponding operating system commands which return performance information.

Examples: HP OpenView, see http://www.openview.hp.com/

Thirdly, if it is not possible to get the information onto a central management machine one may move the management and processing to the resource. As most systems management packages contain software distribution functionality it can be used to install extra software at any machine. The agent bears the extra task of accessing performance/load information and checking it against a number of rules (which implement a **policy**). If a rule (most often a threshold value) is violated the agent sends a trap to the central management site. Otherwise it is silent and does not put extra load on the network.

Many error transport and notification services exist on all systems that work in any conventional environment. The administrator usually has no idea what is occurring on his systems. Only when something deviates from the defined standard does he get a notification. While it may be sufficient to control a large system this way it is impossible to record integrated, timely and precise information about the workload.

Example: Tivoli TME 10, Tivoli (see www.tivoli.com)

Commercial Performance Monitors

Here are some examples of widely used performance monitors for the mainframe environment. No similar tools do exist in the same quality for distributed server systems[6]. Usually such tools offer the graphical and/or textual) presentation of several hardware and OS-related performance metrics, the export of these data into a file or database, and the definition of rules or policies that trigger exception handler or automation routines based on the incoming data. The tools and their functionality are very OS-specific:

- **OmegaMon** (Candle [23], see also www.candle.com). OmegaMon is a real-time oriented monitor for MVS and VM. Extensions for IMS, DB2, VTAM and CICS are available. The tool can be used to display on-line overview information about the system down to very detailed data at transaction-level. Some level of automation is supported.

- The **Windows NT Performance Monitor** is a tool that comes with the Windows NT operating system. It is able to display a number of attributes of the local operating system in a graphic window. The user can use monitoring for the attributes he/she is interested in [29].

- **Platinum ServerVision** is able to monitor UNIX and Windows NT based server systems. The main focus of the tool is systems management and it offers analysis of operating system performance and threshold alarming (aside from other functions).

[6] For example, [10] mentions only some protocol analyzing and network monitor tools in that area.

- **RMon/AIX** is an example of an AIX (UNIX) monitor that is able to cooperate with central systems management applications (see ref. [28]).

- **SM2** (or **ARGUS**, which was the predecessor of SM2) is for the BS2000 operating system [4], [24], [25]. This is also an OS-specific monitor. It provides OS-related performance metrics.

- **MGX** does not fit well into the list of monitors, because, by and large, it is not a monitor. It relies on other tools - mainly SMF - to produce log records. Thus MGX can be used to read the records and to produce input for the SAS statistics software. Based on SAS, MGX can do a lot of reporting and analyzing of the data. For systems that do not directly support SMF, tools are provided that generate the required SMF-output ([30], www.mxg.com/prodinfo.htm).

Workload Data Analysis

According to ref. [4], analysis of measured data is usually a manual process done by an analyst. Due to the complexity of the computer systems and the volume of data, today there are attempts made to automate the process. The first of these approaches are software monitors that try to analyze the data on-line as well as pin-pointing and reporting certain unusual events. More advanced systems try to use expert systems. The first version of such a system is being used for detecting bottlenecks [19].

Furthermore, Zorn in [4] goes into the problem of measuring and analyzing distribute objects. He identifies as the major problem areas:

- *heterogeneous objects*: in distributed systems like a LAN it is very likely that one has to deal with different objects (for example, different server applications like a database, web server, groupware server, file and print server, names server, etc.). Another problem is the semantically difference in the meanings of resource attributes at different locations/objects (for example logical disks and file systems).

- *central or decentral recording and preprocessing of information*: Considering that the communication overhead to send all information to a central monitor system, it is suggested to preprocess the information at the location of the distributed resources. This imposes the threat that required data are not available at a central site. An agent that monitors a resource attribute, writes a log and only sends a message on a certain event is an example for decentralized recording and preprocessing.

- *central or decentral control over the monitoring process*: In the case of problems, the central site remains uninformed about the status of monitoring, thus the value of the whole measurement is in doubt. Those approaches that expect to receive information only in case of problems at the agent's host will remain in an unclear situation if the communication infrastructure is broken which may not be detected by a listening central monitor. In that case it may mistakenly assume that there are no problems at all.

- *creation of a common time basis*: Usually time stamps are added to sample values to enable the representation of the development of the value. If the monitoring is performed in different computer systems, it would be necessary for the system clocks to provide (nearly) the same time stamps. The problem becomes even greater, if new, aggregated values are computed at the central site.

For some of the problems, e.g. the problem of aligning system clocks, solutions do exist [20], [21]. Zorn concludes that the joint analysis and interpretation of interconnected relations and problems are more or less undiscovered as yet and that new methods and tools will have to be developed , e.g. [22].

1.5. Organization of this Thesis

This dissertation consists of the following parts:

Chapter 1 introduces the reader to the topic of workload monitoring. It presents some background information about the topic and gives a brief overview of related information and references.

Chapter 2 contains a short summary of the results from the work with a number of case studies and related work with the monitoring software. The most important observations are outlined. We try to explain our observations and provide assumptions about the cause of measured results.

Chapter 3 presents requirements and architecture of the monitoring software. It introduces the major components that make up the monitor. The reader should be able to understand the problems that are addressed and how the components of the tool act in concert. The following chapters cover the technical details on each component.

Chapter 4 discusses in detail the monitor agent that retrieves local or remote workload information. The agent is responsible for controlling required monitoring code, resource detection and forwarding data. Stability, robustness and secured data consistency are the most important focus areas.

Chapter 5 explains the communication infrastructure that was built to allow an efficient, fast and reliable transport of synchronized sample data. Minimizing the network traffic, dealing with different network protocols in parallel, securing data transmission in unsecure networks and enabling robust operation by supporting redundant cooperation of many units are explained in detail.

Chapter 6 shows how the central component collects and processes the available information. This component is resonsible for all on-line processing of workload data which includes data retrieval and (time-) consolidation, storage management, alert definition and monitoring, alert triggering and automatic initiation of actions in case of defined events.

Chapter 7 deals with the way the information is stored and analyzed. The chapter covers the workload database that is the result of long term monitoring. Problems concerning the volume of information, robustness and automatic database maintainance and error handling are explained in more detail.

Chapter 8, which makes up half of this dissertation, contains the condensed results from two case studies. Because the amount of collected information is large insignificant data has been omitted and the graphical representation of each parameter is folded into a "virtual" week. In addition, interesting results from the assoziation (correlation) detection are given at the end of this chapter.

Chapter 9 concludes the dissertation and discusses the findings and some of the remaining problems in that area.

2. Summary of results

2.1. Why long-term monitoring

This dissertation focuses on the problem of long-term monitoring, that is at least six to twelve months or more, in distributed PC server systems. The major goal was to find a way to provide a sound basis for IT planning and architecture work. Two of the most well known areas of long term monitoring are the *observation of the stars* and *weather observation*. Early in the history of modern man it became clear that it was necessary to record important information. This triggers the necessity of implementing a way to record the data in a permanent way. Many scientists think that this may have been the incentive in the invention of alphabets, scripts and mathematics. Both observation areas record information over which the observer has no influence. Both are also illustrations of many problems associated with long-term monitoring, for example information was not stored durable enough that later generations could still read or understand it. A wrong understanding of the nature of the data led to the recording of invalid data and the omission of important data. These two examples can be compared to the monitoring and recording of parameters in a computer system over a period of several months or years.

PC server systems add a number of additional problems: a much higher frequency of samples (information changes within seconds), a big number of server applications, which provide very different data (if any), big differences in the volume and scale, a rapid evolution with ever changing products, interfaces and data types, an unstable infrastructure with a lot of changes, and many more.

If we use the observation paradigm from above again, we can understand it to imply stars appearing and disappearing every night, with the observer never knowing where to look for them. It would be unclear whether to measure daily rainfall in millimeter, centimeter, or meter. How would the instrumentation of a meteorologist look like, would not be scale and parameters (temperature, rainfall, humidity, etc.) be the same every day?

Another central issue of this study forms the question: what metrics are available on-line from current server applications? While it may be possible to determine the amount of yearly rainfall from rock sediments, it is more efficient to collect the water when it comes down to the surface. The result of this research is that there is a lot of information, but not always, material needed to work in the area of configuring and planning for performance and capacity. The appendix E contains the complete list of attributes that were found to be measurable on-line at the software systems under consideration.

As a result a product-independent monitoring system was developed that demonstrates that it is possibe to implement a native OS/2-based system on standard PCs without the need for expensive UNIX or mainframe hosts. The main problems that had to be solved were:

- The construction of a **general purpose** monitoring system open for various data sources and able to include different types of data formats. Sources, from which data has to be integrated, are the operating system (OS/2), the network operating system, device drivers, protocol stacks and server applications like a database, an e-mail server, groupware applications and so on. No assumptions about the nature of data and dependencies on certain software products must be introduced into the system (for example, the tool must work if there is no TCP/IP available on the host machine). Chapter 3 gives an overview about the requirements and the architecture of the tool and chapter 4 describes the implementation concepts in detail.

- The **discovery process** has the task of finding the resources that can be monitored. It must be avoided that an analyst has to define and select resources for monitoring. As resources in a distributed system, like shared devices, network connections or even server machines, may come and go dynamically, a flexible and automated detection mechanism is necessary. Chapter 4 and chapter 6 discuss the problem at the level of a single server and also at the level of the whole system.

- Because it is not possible to rely on the existence of a certain resource, it is necessary to track whether data is available for a certain sample period. Many tools simply replace a missing value by zero. That gives a wrong impression and hides potential problems. In addition later analyses of the data is distorted. The concept of **not a number** is introduced to distinguish between a valid and a missing sample value.

- The **time synchronization** of samples is a well known problem in the area of monitoring. In order to compare data series from different machines it has to be known when samples are taken and whether two samples from different servers have been generated during the same sample interval. As clock synchronization over different server systems is not guaranteed, timestamps from different systems cannot be compared or used as a reference. The length of time data travels from its source to the central processing site (and data store) has to be taken into account too.

- The **existence of different message formats** may occur in a distributed system. Software updates may take some time in a large installation. During that time different versions of a software may operate on different machines. A tool that relies on the communication of many components over a network must be able to operate during software updates.

- The **heterogeneous environment** (network protocols, server applications, different versions of the OS, of the applications and of the tool itself) makes it difficult to build stable software able to run unattended and with no need for setup and operator intervention. Besides the OS/2 kernel itself there is no guarantee for any other component on a specific system. A monitor tool must be flexible enough to adapt to the environment where it is supposed to execute. This should be automated to avoid the need for manual configuration.

- General **resource constraints** (CPU, physical memory, disk space) need special focus. PC hardware and software is still limited compared with UNIX workstations or mainframes. Some attention has to be given to the problem of processing huge amounts of information on such systems.

- The **inadequate stability** of a number of software components are a major hurdle for long-term monitoring. The ability for uninterrupted execution of a process for years is not a given attribute of any software. A robust agent engine is needed that is able to run without interruptions for years and is able to recover from system or subcomponent faults.

- The efficient **storage of long-term workload data** is a problem concerning available diskspace[7] and the later processing of huge amounts of information. Simply because of the time it takes to collect so much data, the resulting database is an expensive asset that has to be protected against loss or destruction. This includes the problem of migration when the structure of the database has to be changed between versions of the monitor software. Chapter 7 describes the processing of the information.

- **Data errors due to instrument faults** result in meaningless data. It became clear that the "instruments" getting read by the monitor can fail over time, especially when

[7] Using the NT system performance monitor with logging turned on a single sample value for one system can be 35 kB in size. Assuming a sample interval of one minute, one year of monitoring for 100 servers would need more than 1800 GB of storage and that only is the figure for the operating system.

the host system gets in a crisis. During that time they can deliver completely invalid information. Counters can over- or underflow and be mapped to strange numerical values. In contrast to short-term monitors in a lab, which are observed by an operator, such events may pass undiscovered and generate invalid data records in the database.

- The **correlation of a number of parameters** may point to unknown relations or potential problems. One central usage of the resulting workload database is the detection of correlation and potential dependencies between parameters and resources. The big number of resources and resource attributes and the length of the monitoring period demand high performance algorithms for processing.

A number of case studies that took up to three years showed the robustness and functionality of the tool set and provided a lot of valuable data about commercial client/server office systems. The analysis of this data confirmed the cyclical nature of a part of the information. In the foreground the daily and weekly cycles became visible and beyond that, seasonal changes could be seen. However many of the measured attributes of the systems are not cyclical and do not depend on the ups and downs of the user population. To a greater extent than initially expected many attributes are relatively constant or "random".

In contrast to our expectations, the correlation detection did not yield many correlations between attributes on the same host and only a few obvious correlations between attributes on different hosts. Many approaches to sizing and planning assume such relations between certain attributes. The fact that such relations cannot be seen in the data confirm the opinions of many professionals that classical methods from the mainframe cannot be applied easily to distributed systems.

The main reasons for this may be that

- there is no information about internals of a software (for example, the usage of internal resources like buffer pools, handles, sort space, etc.),
- software behavior changes due to optimizations (for examples, a cache) or poor quality in course of time; restarting the application or the machine may reset the system again;
- complex (multidimensional) relations, where several attributes influence another attribute, cannot be detected from the workload database
- the workload submitted to the servers and the consumption of resources due to the load are much more stochastic than expected,
- there may be time delays in reactions between related attributes; a special case is a resource that, when allocated, will never be released again,
- usage pattern of many services are rather irregular as the processing takes place at independent workstations able to store or cache a lot of data, and the way such an end-user system behaves depends on a number of unpredictable factors (ranging from the time that a user decides to use a function to the possibility of programming macros or new functions in many commercial office packages), and last but not least,
- the sources of the monitor may be unprecise or even wrong.

Looking at analytical models it became clear that many of the classic metrics used to solve a model are not available or have a different meaning under OS/2. Examples of common metrics that cannot be measured in many cases are:

- the utilization of a resource, especially CPU (see below)
- the number of active users on a certain host or a certain server application
- the resources consumed by a job
- the number of visits of a job at a certain resource
- the system duration of a job
- usually the number of jobs (or service requests), if there is the notion of „job“, and then, consequently, all job-related metrics

CPU utilization is one of the most visible „measurable“ attributes, when dealing with performance, and a good example of common misunderstandings. There are different possible interpretations and ways of measuring it. One very low-level OS-related interpretation is that the utilization of the CPU is the quotient of the number of used time slices and the number of available time slices. If ten out of 100 time-slices are scheduled to a waiting process, we have a 10% utilization; if 100 time-slices are used, we get 100% utilization (and we assume that this resource is fully saturated). From the viewpoint of the CPU (and the OS) this measurement is correct. A number of conclusions can be derived from that.

In practice, due to „background“ demon processes and a sophisticated priority scheduling schema within OS/2, the CPU may be assigned to a waiting low-priority process if there is nothing else to do. Nevertheless, the CPU is available at any time, if something more important has to be done. Counting used versus not-used time slices does not give any clues about the workload that is handled by the CPU. Because OS/2 does not report CPU utilization a measurement algorithm has been developed in order to report useful measurement data that filters unrelated background tasks.

Two long-term case studies were evaluated. Many charts, depicting average and extreme values of a "virtual" week, calculated from a year worth of data were given for all important attributes concerning network services, network traffic, file system access/usage and the operating system. The results of the correlation detection are given here, too. They show, which relations between different attributes can be seen (and proved) by the recorded data.

2.2. General Observations

A major part of this work was the application of the described techniques to a number of real environments. Due to the amount of data that was accumulated over time it is not possible to show all data with details in this paper. Therefore important and interesting results have been extracted from the data collection and are presented on the following pages.

Aging of Load Data

When we began with first evaluations of the data we found that the information about the servers in the evaluation environment quickly became "out-dated". This means the value and derivable message of the information decreased due to constantly changing conditions in the environment. For example, some of the server machines were upgraded or exchanged over time while their names remained the same. From the point of view of software nothing changed, but of course CPU, network adapters or memory usage cannot be compared with the old machine. Another example is the addition of exchange of DASD devices with higher capacity.

A conclusion of this is that a meaningful analysis of many parameters need some sort of additional information about the environment. It may also be possible to track changes of the configuration and to enrich analysis with the static information - although this was not implemented in this work.

Volatility of Information

The whole system, and consequently the analysis part, are built around the idea of continuous monitoring. When working with the data the considered time frame is one year. Constantly, old data disappear and new data is added to the picture. Consequently the results of calculations based on these data changes every day.

Because the usage of the servers changes over time, pictures and analysis results in this paper are snapshots taken at a certain time.

Data Representation

As shown in chapter 8 the observed load in the monitored show cases is cyclic. As the user community consists of office workers (in Austria), there are five work days from about 7:00 to 20:00 and two days with hardly any load on the weekend.

Because of this and in order to fit a year's worth of collected data onto a page, the information used in this paper is converted into the format for an condensed "average week" consisting of 24*7 values. Each represents the average of the sample values of the same time slot over all of the weeks of the last year with minimums and maximums added to the graph.

Sample Errors

During the analysis of the data another problem became apparent: invalid sample values retrieved during the on-line monitoring phase inflict significant damage to the log information and the results of the analysis. Although the access classes detect and filter out-of-bound values, some of the remaining net data still contain invalid log information (like values greater than 100 for percentage values expected to be between zero and 100). In such cases it was necessary to either improve data quality manually (by removing invalid data from the samples) or compare compressed information with on-going on-line data in order to correct or verify conclusions drawn from these data.

2.3. Case Study Summary

This section contains an overview of all conclusions common to the various case studies. Chapter 8 presents a detailed overview of the gathered data for each case. Each item represents a summary of our observations. Below each of them we provide a hypothesis on the cause of the measurements.

Where all cases yield similar results

1. The number of active users grows continuously between 7:00 and 12:00. There is no sudden "burst" at any time. After 12:00 the number of users declines again with nearly no point in time at which all users are logged on to the system.

   Our hypothesis: In contrast to common assumptions people do not arrive until 9:00 o'clock and start their computer related work even later. After noon, many disconnect from the servers suggesting that work can be done locally or people leave their office like part-time works do. This is an indication for a changing work behavior due to the introduction of new work and employment models (like "flex-time" and "home offices").

1. 20% - 25% of all users remain "active" during non-working hours. They do not turn off their machines. About 50% of these users do not even turn off their machines during weekends.

   Our hypothesis: A significant amount of people will never turn off their PCs because it takes a long time to start the machine, to connect to the network again and to load applications and data. It follows that the number of "active" users is no indication for any workload.

1. "Long weekends" (due to holidays) influence significantly the average number of users on Thursdays and Fridays.

   Our hypothesis: In Austria there are a number of long weekends with Thursday and Friday being holidays. This can be seen in the data because there is significantly less activity.

1. The peak time in the number of connections that users establish to servers is about 10:00 am - two hours before the peak in the number of users.

   Our hypothesis: The number of connections is a much better indication of user activity. A connection has to be established when needed and it is disconnected or deactivated when there is no activity for a certain amount of time (like one hour). After logon connections to application and data servers are established and necessary information is downloaded to the workstation. The fact that the peak in connections occurs much earlier than the peak in active users suggests that users that connect late during the day do not need server resources and that most users do not need the server later on.

   An interesting observation is that a LAN can only be used in that way because users do not arrive at about the same time and then start with their work (see observation 1).

1. Most connections are not needed or used. Depending on the connection policy of the domain, they are disconnected after some time or they remain until the user logs off from the system.

   Our hypothesis: To simplify the use of servers most users have a number of resources associated to their LAN profile. When the log on to the network domain

the system establishes connections to these resources automatically. Usually at least a home directory and a number of shared network printers are registered in the profile. This observation implies that many users would not need these connections but do not care about them.

1. Only few users work with files on the server. Some load applications from servers but most install and use software from their local PC disk. On average one or two files are opened per user.

   Our hypothesis: This observation reflects a shift in the way workstations and servers are operated. Not long ago industry strength PC hardware was rather expensive. PCs were not equipped with large storage devices. In addition it was errorprone and time consuming to install many software packages on big number of enduser machines. Therefore application software was installed at an application server and it users started the programs from the server.

   Then hardware prices have dropped significantly and after some time more and more PCs have become very well equipped. Many common software packages need some local installation in any case, like many Windows based applications and therefore there is not much manpower to save with a server installation. And as the size of software packages grow into the range of megabytes, loading everything over a network becomes much too slow compared to a fast disk. Dissatisfied by poor performance and ongoing difficulties with server-based installations people started to install all software locally. The introduction of laptop computers covering a good deal of the user community has brought the final demise of the application server.

   The introduction of groupware applications further reduces the importance of mere file sharing services. Business data are not stored on a file server because of its lack of management functionality. File servers are still in use but only for tasks like downloading common files. Therefore their use has declined significantly during the time this paper was written.

1. The amount of information that is sent to servers peaks very early, even before the peak of connections. Not much data is sent compared to the capacity of the equipment. The amount varies with different cases (50 kB/h and 70-80kB/h).

   Our hypothesis: This observation is another indication that the servers are used to load application and data once in the morning after user start their work.

1. Many important load parameters correlate directly with the number of active users.

   Our hypothesis: Although one conclusion is that the use of server connections decreases many users use the servers for their work and contribute to the overall load. Most of the workload on the server seems to be directly related to user activity. There is not much batch processing or time-delayed work.

1. The number of directory entries changes significantly during the (average) week. Per users about 15 entries are created or removed per week.

   Our hypothesis: Directory entries are a sign for the organization of data on a disk. The numbers indicate that many directories and files are only of a temporary nature

1. The graphs for the "number of directories" and "allocated diskspace" are practically identical. On average each directory contains one megabyte of data.

   Our hypothesis: This indicates that disk cleanup occurs directory-wise. A related observation is that most users do not care about old data, be it files on a server disk or old mail on their e-mail server. Usage of servers disks soon reach the capacity of

the device. Only the intervention of an administrator can free up some disk space. An administrator cannot handle data on a file level. He has to move or remove whole structures.

A related observation is that the average number of files and directories seem to have some relation. To some respect the number of files depends on the number of directories. When a directory is removed all its files will also be removed. Our assumption was that static directory structures are created and that the life cycle of datafiles is managed within that structure. The measurements suggest that directories are not static containers for more dynamic contents but are created and removed as data come and go.

The consequence is that applications should not build and work with (lists of) static references to directory entries because such references will grow very fast and point to entries that already disappeared. The access control component of the network access layer suffered from that problem. It maintains a list of directories with associated security and access permission information. In order to give access to new directories the administrator has to submit a function that adds new directories with new security information to the existing list. This list grows fast and after some time the performance drops due to access control.

1. Most files are accessed on Mondays.

   Our hypothesis: From the data about aging of server files we conclude that users do most of their server related work on Mondays and do not touch such files again during the week. We assume that files are downloaded to workstations at the beginning of the week. When work is done during the week the file is copied back to the server.

   Consequently, short-term measurements performed to retrieve information about an environment should include data for a typical Monday.

1. Judging from disk activities, Monday morning is the time of most user activities. The rest of the week this parameter does not depend as directly on the number of active users.

   Our hypothesis: The same assumptions as for observation 11 apply for this observation. It is interesting that disk activity does not seem to have something to do with user activity. Caches at the server and the workstations and the increasing use of local disks at enduser workstations contribute to this observation.

1. Some tests with high volumes of disk operations (generated by a number of copy operations from and to clients) show that the cache can maintain high efficiency for several minutes. If the load remains high the efficiency drops and an increasing number of requests have to be routed to the disk subsystem. This slows down the speed of file I/O over the network.

   Our hypothesis: The cache at the server as a certain capacity. Compared to data copy in memory (from the network controller to the cache memory) the network transport is very slow. For some time the cache is able to hide disk write access which is even slower. The server receives data as fast as the network allows for. While it receives data it is able to write part of it to the disk. After some time the cache capacity is complete used. Now, when the server receives more data the sending workstation has to wait until the cache is able to write away some older data and to make room for the new information.

   During normal operation this situation is hardly observed because the cache is large enough to keep common PC files and the difference between network speed and disk write speed is relatively narrow.

1. The ratio between write requests and disk writes is about 10:1 on average.

   Our hypothesis: This observation indicates that the cache is able to cover most write requests and that it is hardly the case that an operation has to wait until a disk write is actually performed.

1. There seems to be as much background traffic on the network as user related traffic.
   Our hypothesis: Even if there are not many active users and if it can be assumed that nobody is really working on any workstations much network traffic can be observed. It seems that the communication between the components connected to the network generates a significant amount of traffic. During office hours user related traffic is added on top of that.

   This has to be taken into account when capacity planning is done based on estimated user traffic.

1. There is an obvious correlation between the number of connections and the number of Ti (timer) expirations (when the system has to confirm if the connection is still up and in use).

   Our hypothesis: The token ring adapter implements a number of timers. One important timer is the "Ti" timer. This timer is used to check whether an open connection still exists. When this timer expires (triggers) the adapter tries to communicates with the partner on the other side of the connection. The partner may either confirm that the connection is still open. Or it may signal that the partner application has closed the connection or has ended completely. Or the partner may not answer at all (if, for instance, the machine is shut down). In the later cases the connection is considered "inactive" and is closed. If there is a process (thread) active on this connection it will receive a return code signaling the termination of the connection.

   The more connections exist the more timer expirations will occur. As long as a connection exists it will generate some network traffic even if there is no user activity on this connection. Understanding this relation it becomes clear that the number of connections is relevant for planning activities and that unneeded connections should be avoided.

1. The amount of disk storage that has to be dedicated to the spool area of the printer queues is very small compared with the capacity of common storage devices.

   Our hypothesis: In spite the impression that print jobs from typical Windows applications are large, most print jobs are in fact very small in size. We assume that the reason for this is the low speed of workstation printers and LAN printers (compared to the high speed printers of the mainframe). Because it takes that much time to print a larger document or book users avoid to print such documents.

   We observed a similar development when such high speed devices for the mainframe where replaced with new network printers. These printers are shared between Windows and OS/2 based office applications and the mainframe. Because they are much slower than the older printers and often need user intervention (feed paper, handle paper problems, etc.) less people spool big print jobs to these printers. Besides that, the traffic at these printers cannot be monitored because they accept input from very different sources. In contrast to the old architecture there is not a single server machine which controls the flow of print data to a printer of that type.

1. The amount of time blocked print jobs spend in spooler queues is amazing: up to thirty eight days.

   Our hypothesis: This simply means that the responsible people do not care about

their printer queues. This is related with an organizational change. The point of control is constantly moved up to more central units: from the workgroup level to the department level to a multi department level to a company wide level to an international level. In the beginning someone within the workgroup had control over the servers and could repair a problem (for instance restart a printer queue) immediately. At an international level with a scope of control of tenthousand devices it takes days until someone reacts on a specific printer problem.

Where nothing special could be found

In spite of the author's expectations, about a number of parameters and assumed dependencies, no general rule or correlation could be found:

1. Most of the graphs that show user related workloads like open files, connections, sessions, data traffic, etc., do not display the double peak line that is usually used for estimating user behavior. The double peak is based on the assumption that users do most of their work before and after lunch break.

   Our hypothesis: As already pointed out before we assume that server based information is need most after users start their work in the morning. After workstations loaded applications and data the network traffic and the server load drops.

1. The amount of data transferred between clients and server machines does not correlate to the number of active users.

   This is a special case to the general observation above.

1. The amount of allocated diskspace that is actually used during everyday work does not correlate with the number of active users.

   Our hypothesis: As mentioned before a very interesting observation is that most data are accessed during the first two work days of the week. Then the information is not used for the rest of the week. Taking into account that on average nearly no data has its last usage date four to ten days back we come to the conclusion that information is used for some time, usually accessed at the beginning of the week. If work is done most of the data are not used any more. We conclude that information that is not accessed for more than ten days can be migrated to an archive or backup device. Instead of more and more disk space a host-like data migration facility, that moves unused data to a cheaper medium, would be much more cost effective.

   Looking at the measurements we estimate that only about 5% to 15% of the disk capacity is actually in used while the rest will not be accessed any more. This has to be considered with some caution because of observation 4 (see below).

1. Regular backup activities which touch a lot of files on server disks obscures the measurements of actual disk access. Complete (in contrast to incremental) backup procedures that act on a file level access many files and modify the access information of the file system. Therefore the measurements about the amount of server disk space currently in use may not reflect actual user behavior.

   Our hypothesis: Because of the assumption that a backup procedure does not change anything in the file system we expected to retrieve more information about user behavior concerning the use of server based files and data.

   Besides the effects of backup procedures another activity further obscures any observation about data usage: regular changes in the server environment which includes adding and removing disks and servers. Directories and their files are moved to new disks. Such activities change the capacity of the total system and the

access information of the data is modified.

Space that can be accessed by users is used up quickly. Therefore no real growth in diskspace allocation has been measured. Accessible disks always operate at the capacity limit. The available disk space that has been measured is space reserved for system use or other tasks.

There were no significant trends detected. For case study I a general decrease in the usage of the servers was found. One reason for this is the migration of a number of users to other domains (servers). Another reason is the on-going movement to the use of laptop computers which operate in a more server-independent way than normal desktop-PCs.

Conclusion of the Case Studies

During the time this paper was written the role of server machines in the observed areas slowly changed. In the beginning the sharing of (expensive) resources, mainly disks and printers, has been the main task for servers. Centralizing resources at a server has given the administration more control on resources and maintenance has been easier. There have been attempts to avoid errorprone software distribution by making use of server installation. The significant increase in hardware capacity, the low ability of centralized administration units towards good responsiveness concerning new requirements and resolving problems, and the move to laptop computers has redefined the role of servers. There is an enormous need to share data between many users but that demands for sophisticated applications that are able to organize and manage data, to manage versions and access to information and that simplify presentation and retrieval of information.

Internet technology and groupware software fill this demands. Lotus Notes is one example for such applications. It combines an easy and flexible construction tool for data and workflow management applications with the build-in ability of internet access with all the necessary security functionality. HTML and Java enabled browser can access such servers via TCP/IP. Other network protocols are increasingly replaced by TCP/IP and some PC servers found a new task as DHCP server (dynamic IP address management) or name server or proxy server. Server machines of this type provide their CPU power and network resources.

Supplying a growing number of users with much more complex functions requires more experienced and skilled administration and more powerful hardware as well. In the observed environment that means further centralization and the move from PC servers to top-of-the-line UNIX machines or (MVS-based) mainframe computers.

SRVMONPM - Server Monitor for the OS/2 Presentation Manager

The Implementation Of A Workload Monitoring Infrastructure and Data Processing Application

3. Architectural Overview of SRVMONPM

This chapter gives an overview of the requirements and resulting architecture of the workload monitor and introduces the major components that make up the complete solution.

3.1. Requirements and Primary Design Objectives

Expand the span of control from one machine to a network of connected machines

"The network is the system." This statement originates from someone at Sun Microsystems and today we are approaching this vision at great pace. IBM has adopted the term "network centric computing", which means that there is nearly no stand-alone or closed-environment processing. Most applications require the communication between and interaction of a number of computer systems to fulfill a certain task. While applications, infrastructure and development tools (like CORBA, DCE, the world wide web, etc.) make the network more and more transparent, systems management still focuses on the individual hardware box. Soon the placement of resources will be transparent to the developer, the application and the user. The provider of a certain server may "move" in the network. But who is going to maintain this knowledge? Who will be in charge to decide which service is to be used?

Current implementations partly fail, due to the fact that much of the responsibility has been moved to the administration and the people who set up **middleware** components. In order to understand and plan the necessary components for the network of the (near) future, it will be important to understand relations among servers and to become able to reroute service requests to the place where free capacity[8] is still available. A service is needed that offers reliable information about *time & place* of available resources.

In view of the above, an integrated view and integrated, consolidated data should be presented to the user of the tools that work with the data. Instead of focusing on certain details, the interconnected system as a whole should be monitored and analyzed.

Synchronize and map data from different sources and machines into a uniform time grid

Normally, if one looks for workload data, one finds that applications provide a more or less well defined snapshot of their current status, e.g. the number of transactions since the system was started. This kind of information is not very useful if one does not consider time and duration in which such observations are made. Very often the snapshot itself is not of great interest while changes of state can deliver important information about a system.

[8] A general assumption is that (for the near future) the capacity of computer resources is limited, partly due to the cost of upgrading many PC servers, partly due to physical capacity limits, and performance management is worth the effort.

Observations of state changes relative to a certain period of time are a key factor which must be taken into account. A major design goal in this work was to find a way to form the connection between system changes and the related time frame visible and **understandable** to the administrator or planner. Great effort was invested to develop some kind of processing and representation that reflected this connection.

Product independence and open interface

In this context a product is a set of software components from a particular provider that, together, provide some service on a server. It must be possible to connect a product to the monitor and to collect load and performance data about it. Usually a business can choose from a number of products to deliver required function. **Product independence** quickly became a very important design goal because it is not possible to rely on the availability of a specific product.. Being independent still does not add much value if one is not able to provide the data to the application. Therefore an **open interface** is necessary that can be used by anybody to enable monitoring for new sources of data. Any product should be connectable. If one focuses on certain applications while ignoring others the picture is never complete and important input for an infrastructure design may be missed.

Completeness of collected data

One major objective is the ability to create a workload database that can be used for answering questions about the observed environment later on. Up to a year of information (or even more if the owner allows for it) should be kept in this database. Because it may not be clear which data will be needed in the future all known resources with all their attributes must be retrieved and stored in the database. Even if the user currently is not interested in certain values and has disabled their display. Data collection, recording and display must be kept apart.

Appropriate graphical representation

Many applications provide information about their workload in one way or another. Often it is not easy to find this information and make use of it. It may not even be visible to the administrator. It is necessary to buy additional tools to extract them. Most of the time such workload data are non-intuitive, textual and product specific. Instead, this workload monitoring and the analysis application should build an engine that is able to handle all kinds of workload (and performance) information. The information has to be collected and presented to the user in a way that makes it easy to understand and to work with. A flexible and customizable **graphical representation** of data combined with numbers - where necessary - seems to be the best and most suitable way to translate the volume of raw data into clear information.

Different graphic tools are needed for processing and displaying current information and for processing and analyzing data from the database. Monitoring actual data should include some kind of alerting to enable a user to trigger some action or to get pointed to a specific situation. Because of the expected number of resources and attributes it would not be possible to look at all of them. The big number of managed servers is one of the big differences to managing a host. Looking at the data of a single machine can be done, checking hundreds of servers cannot be a manual task.

Ease of use

From the beginning it was intended that the code produced during this study should go beyond mere academic research work. An important target was that the tool be a real value-add for people involved in infrastructure design and management. **User friendliness** and **ease of use** had to be considered wherever possible. Seamless integration into the OS/2 desktop, state-of-the-art OS/2 PM application behavior,

attractive window and graphics design, and complete and comprehensive on-line information on windows and functions to achieve user acceptance.

Efficiency - avoid degradation of target machine's performance

Most parts of the application are designed using an object-oriented approach. The implementation is done using C++[9]. The monitoring infrastructure must be very careful with its resource consumption and must be very efficient. Software components that will execute at a productive server are not allowed to influence or degrade the performance of the services. The "footprint" of the monitor must be kept at a minimum. Even the front end is intended to run as one of many applications on the desktop of an administrator workstation. Therefore all parts must perform well and must be sparingly with all workstation resources (especially RAM/working set, network and CPU).

Some thoughts were devoted to this requirement, and as we will later see, several design decisions made in favor of efficiency.

Adaptability

Today's networks consist of heterogeneous components. Hardware and software from different sources are connected and communicate via open or proprietary protocols. Therefore all parts that make up the application must be able to **adapt to the environment** where they are executing. The part running at the client should execute on any computer operated by OS/2 and deliver as much data as possible. This is however not an easy task. In the project not all obstacles could in reality be overcome and even now there are still several problem areas waiting for a satisfactory solution.

New server software is appearing in steadily decreasing product cycles. It seems to be very important that the set of tools can be **expanded for new software** with a small effort and without the need of design and code changes.

Portability is not one of the primary goals. While the code of all components of the monitor are written in C++ and can easily be transported to other platforms, the code that retrieves information about the system has to be very product and platform specific. Therefore this code is not portable. To be able to connect other platforms, new agents will have to be written from scratch. All GUI parts are written for OS/2's presentation manager and it would be difficult to port that code to other platforms.

Robustness - automatic resolution of system/application problems

The monitoring application is not meant for experiments in a lab in a well-defined environment taking place over a period of time, but for employment in a near 7x24 hour mode[10] in all possible situations of a real client/server network. For deployment in productive environments the tool must be able to operate without user intervention for months or years[11]. Problems or malfunction of the monitor or required external components must be detected and resolved. If necessary the on-line monitor must notify an enduser.

Support of different management strategies

The way systems management is done may vary from enterprise to enterprise. Many tend towards a centralized management. Some have to support distributed management, e.g. where an organization is built by a number of very different groups in widespread locations.

[9] IBM C-Set++ on OS/2 Warp.
[10] 24 hours, seven days a week; in other words, uninterrupted operation.
[11] This proved more a challenge than originally anticipated in the project and was time and effort consuming. It was not as simple as "no program errors" - which is itself not without complications.

Any systems management application must be able to support different styles of management. A user must not be forced to change his organization in favor of a certain tool.

From that follows that any number of management consoles should be able to access the information. The tool must route workload data to one or more places. Users may define the data they are interested in and what should be monitored but different users must be able to work independently.

Flexible configuration

While we often use the term *client/server environment*, in fact there are large differences between different enterprises and between different locations of the same enterprise. Differences in hardware and software release levels, in the way the network is built, and in the way systems management is organized allows a great deal of flexibility for a tool which should fit into most scenarios.

In order to offer flexibility for the deployment of workload monitoring the application has to be split into several pieces. Each has its specialized task and can be put into the most suitable place in the environment. It is able to detect in which environment it operates and automatically activates the code to work correctly. Each component should act as independent from other components as possible.

Another important concept, which has already been mentioned, is the implementation of **continuos operation** within all components. Neither user errors nor network failure may disturb the monitoring tool. Each component must recover from errors without operator intervention. Components should handle errors silently. An administrator may use alerts in the managing station to be notified about potential problems. Monitoring has to continue when some components have to be replaced during software maintenance activities.

Scalability

The implementation of any kind of function in a large network is a challenge for systems management personal. People must be cautious, and do not particularly enjoy implanting agents on a server. Therefore a stepwise growth path from a simple "one PC monitors a group of servers" evaluation to a full enterprise wide load management has to be supported. Local and remote data query are supported. Filtering and manual specification allow the retrieving of data subsets to further reduce additional network overhead.

A state-of-the-art tool must allow for a quick move into its functionality without long preparations and setup activities. Once the user has understood the potential and functionality of the tool he may be willing to invest more effort to capacitate other functions. This approach has already paid off. Within IBM a growing community of interested users and consultants have discovered the tool and are cooperating with the author.

Open design

An important design objective in the project is to build a kernel which implements the necessary infrastructure for monitoring and is open for new components. Components may provide new data and/or functions which extend the scope of the tool. In order to reach this goal several new techniques were developed which made use of the possibilities of OS/2 in combination with the power of an object oriented programming language (C++). Examples of openness for new components are the agent code, alerts, and views.

A well defined object oriented interface can be used by third parties to integrate new information into the workload application.

Simple agent code

In order to provide any (remote) functionality on a controlled machine some kind of agent is necessary which forms the platform for management functions. In contrast to existing commercial products the agent must be slim and simple. It is not acceptable for an agent to consume noticeable system resources[12]. The main task of the agent is to transform data into a protocol which is used to transport information across the network.

The agent is divided into to parts: monitor modules that implement the knowledge on how to detect resources of and retrieve data from a specific product and an engine that provides the infrastructure for the modules. That are services like memory management and network transport. A monitor module also provides some information to control presentation and analyses. The code in the module is structured in monitoring classes. They are not allowed to stress the target system. Due to given base classes and a well defined interface it will be simple for a third party to write a new module for the agent.

Simple design, few functions and easy implementation should lead to very robust and stable software. This is a prerequisite for running agent software on servers which may be critical to a business or enterprise.

Transparent network

Today, OS/2 supports a broad range of network hardware adapters and protocols. Unlike the SNMP approach, the operability of the load monitor does not have to depend on the network stack which is used on the target computer. To keep the design simple the underlying network must be transparent to higher components - especially those which may be provided by a third party. At the time of development no *middleware* was available which would hide the network completely. Therefore an intermediate software layer is created which handles network access while implementing a sound and useful interface to other components. This layer is included with the tool. Most parts of it do not have to care about used network protocols.

Avoid central control mechanism

The manager has to deal with a great number of workstations in the network. The manager never actively contacts a monitored workstation to ask for information. That is to avoid the network traffic associated with such requests. It is always the agent who sends information (but not directly to the manager). The tool provides the infrastructure to transmit and store the information. A sessionless protocol is implemented to be able to handle great numbers of supported resources and agents on low-cost hardware.

Minimize Network Overhead

Usually today's client/server environments consist of many workgroups spread over the entire enterprise. Each workgroup uses one or more servers. Some of them are locally assigned to a group, while others are used by many groups or are even unique at a location or network. The number of server machines is high and still rising (although the server hardware has more power than ever before).

Management of all the servers is often done centrally, especially for more complex tasks like workload monitoring and management, balancing and capacity planning. The reason for this is such jobs require skilled people and a business cannot afford to have some of them in each department. All these tasks are beyond the scope of a local LAN-

[12] One key inhibitor for the general use of "SystemView for OS/2" was its hunger for resources on the target machine.

administrator. The enterprise may gain some benefits by making idle capacity available to others without the need to invest in new hardware.

The monitoring application must, thus, be able to collect and handle all the data from the servers without generating noticeable overhead on the network or machines being monitored. Besides its ability for central management workload, monitoring must also be supported for the workgroup. In fact both will be used at the same time. The workgroup administrator is only interested in the data of his environment.

In order to fulfill the requirements, a staged approach for the transportation of the data has been introduced. The biggest part of network traffic is done where enough capacity is available: in the LAN. All data are gathered on a server local to that LAN and are available to interested parties. Inter-LAN traffic is kept to a minimum and by sending consolidated data blocks additional network overhead is omitted as much as possible. To keep the monitoring related load on the network low the number of messages exchanged between the different components of the tool must be as low as possible.

3.2. Building Blocks

Figure 6 and Figure 2 give an overview of the relations between the components involved. The monitoring infrastructure and the monitoring manager consist of the following parts which, in turn, make use of other infrastructure components and APIs provided by the operating system and the network:

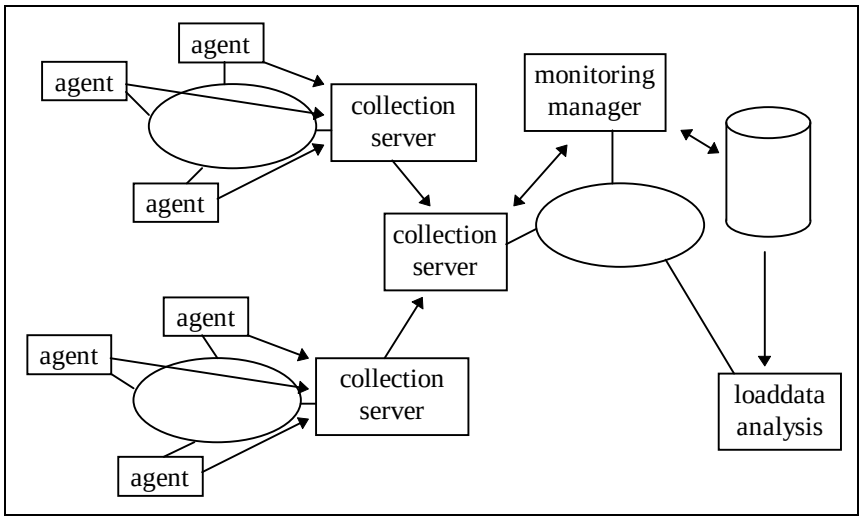


Figure 6. Architectural overview

1. The ***monitoring manager***

   This component is located at one (central) or several (decentral) management sites. Because it can exist more than once the concept supports both centralized and decentralized monitoring philosophies. It receives workload data and handles all further processing of this data - including *graphical presentation* and *recording* on a device for later analysis;

2. The ***agent***

   This component runs on all computers where workload and other resources parameters should be monitored. This includes any PC (with OS/2 as its operating

system), but the main focus lies in any kind of server (LAN, DB, application, etc.) and not in monitoring end-user workstations.

3. The ***collection server***

   It is used to receive load data from a number of agents. Neither agents nor servers are assigned only to one another, and an agent may switch from one server to another. The component stores the information. The collection server offers data to managers who request the information. In addition the data may be forwarded by a collection server to another server. The server component is network-independent: it detects available protocol stacks and supports all of them at the same time. Agents may use different network interfaces to connect with the server. As a side effect this component can be used as a gateway between different protocols.

4. The ***load data analysis tool***

   This component is used to work with the database of logged information. It provides graphical representation of the data, conversion of data for other applications, and mathematical analyses and reporting.

The following illustration shows the layers of a system involved in the monitoring process:

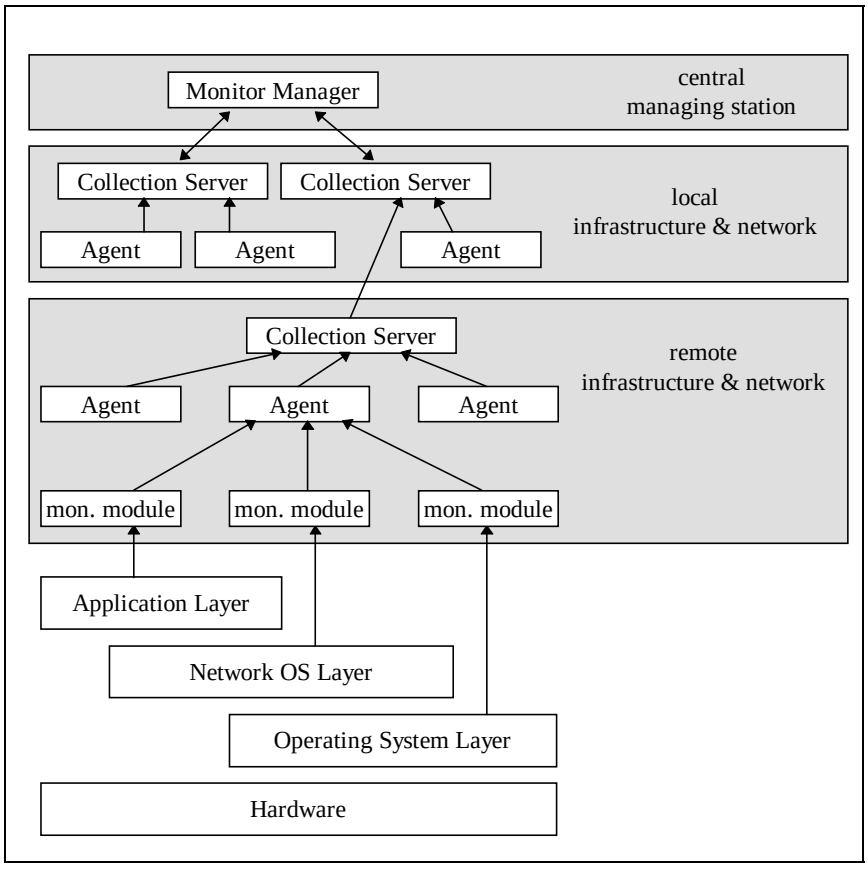


Figure 7. Monitor layer overview

The agent is an empty shell that offers some basic infrastructure services like data transfer and protocol with a server. It has no knowledge about resources. This knowledge and the ability to retrieve workload data is supplied by monitor modules that are loaded by the agent and operate within the scope of it. Monitor modules interact with the resources on the computer system. At regular time intervals agents access load information about the different software layers on a target machine. Then they use the

monitor modules to retrieve the data and transmit it to a server component. Whether the information is used by a manager, and where he could be located are irrelevant points for them. Thus, agents work rather independently from other parts of the application. All processing by the agent is triggered by a local timer (and not from outside).

The server part, which is responsible for temporary storage and transport of the data, can be cascaded throughout the net. It can receive data from agents or other servers and provide the data for the manager component and for another server.

The next three chapters describe each of the components of Figure 7 in detail.

4. Retrieving Information - The Agent

The basic means to retrieve workload information about a system is the **agent.** It may be a **local** agent, which resides on the same machine as the monitored resource, or a **proxy** agent, which resides on a different machine.

The agent itself is a piece of code that has no knowledge about the monitored system. It is just the platform for **monitor objects,** which observe the resources, and it provides the generic data management and communication protocol to the **collection server.**

In its full-function setup the monitoring application requires that an instance of the agent is running on each machine where resources like memory, disks, connections, etc. will be monitored. No individual configuration is necessary. Therefore this requirement is very easy to fulfill and adds no burden on the systems management personnel. Nevertheless, the installation of a piece of software is some effort and there are people who are very cautious before touching their running server machines. To become aquatinted to the tool fast and without the risk to jeopardize server operation a simpler setup is provided, too. In this case the tool can be executed from a single machine in the network where it detects those resources automatically, which can be monitored remotely. That is only a subset of the resources and data that can be retrieved when using the full installation. But for people, who are interested in LAN server data of their (local) domains in the first place, the simple setup may be sufficient. The different setups are explained later in full detail.

One of the great problems with the agent is that it will be used in very different environments and must adapt itself to them. If possible no operator intervention should be needed to adjust the agent. In reality the solution to that requirement showed to be a very delicate and difficult ambition. Both the generic agent code and - especially - the monitor objects are sensitive about their environment. For example, unexpected releases of subsystems can lead to some kind of malfunction. To solve this problem we developed several techniques to make the same agent code adaptable to different systems[13]. The agent is implemented by the program **SRVMCLNT.EXE** (SRVMONPM Client).

4.1. Prerequisites

Several prerequisites are necessary to enable monitoring on a certain system:

1. SRVMONPM always measures the workload of some form of **application software**. It never directly accesses any piece of hardware. In order to retrieve information the piece of software must provide an interface. A monitor object can be developed for this interface which encapsulates it and transforms the information into a form is understood by the tool. Possible interfaces are:

a public API	the most preferred and most efficient interface
stdout/stderr	the capture and interpretation (processing) of text based information
indirect/side effects	the observation of side effects on other resources which may provide information about system load

[13] Different in the sense of configuration, but not operating system.

log files	not used in the current implementation
SNMP	not used in the current implementation

2. The chosen interface must be robust enough to be called in short time intervals (default occurs once per minute) while the application is in use (for example a database while it processes transactions).

   A sampling interval of one minute seems to be a sensible choice. Most of the sources for workload data are counters. Therefore no information is lost no matter how long the interval would be. Thus, the question remains how accurate the information in the on-line display has to be. Considering alerts, when a critical situation has to last for five sample intervals before an alert is triggered, one minute is a good compromise between putting load on the monitored system and accuracy of alerts and displayed data.

3. A call to the interface should not have a noticeable influence on the normal operation of the application or the entire system. Therefore it must not consume noticeable CPU-time or memory and should not reset or modify data areas of the application.

4. The interface must support the automatic identification of the resources provided or controlled by the application. It is not possible to adjust each agent to an individual system manually. The existence of resources may be dynamic; that means the resources may appear and disappear over time. Asking for information about resources no longer in existence must not lead to any malfunction of the target application or the agent. In this case it must provide a proper error code and continue to work normally.

5. A monitored system must be connected to a network. Currently, *named pipes*, *TCP/IP* and *native NETBIOS traffic* are supported for data exchange. A machine which does not support either can not be included in the monitoring domain. "Off-line monitoring" is not supported by the agent.

4.2. Polling vs. Trapping

Given that an agent is able to retrieve and send information the question arises whether the monitoring manager (or a collection server) should actively ask for new information (**poll**) or whether the agents on each monitored machine should send new information timer-driven without request (**trap**). For example, monitoring with SNMP is based on polling: the manager in the network sends a request to each agent and asks for certain information. The agent retrieves the requested information and sends it back to the manager.

Quoted from [85]: *The agent is responsible for reporting on and maintaining the data pertaining to a device, when the manager requests it to. Agents can run on several different types of managed nodes (for example, routers, hubs, servers, and workstations).*

Polling for information has several merits:

1. Network traffic and agent activity on the target machine occurs only when there is an active manager. The manager controls all processing taking place on all the systems.

2. The agent may be built rather simply. In conjunction with monitoring, it waits for a request[14], then answers it. The agent may be stateless: as it does not

[14] An SNMP agent has to implement other functions, for example sending traps in case of problems or errors. The actual design of an agent is more complex than stated above.

implement much function, it does not have to keep a record at any information; this task is performed by the manager, who controls the agent by remote function.

3. If a standard like SNMP can be established that provides a certain set of (simple) functions on the agent side, new systems management applications can be written independently that provide new functions based on existing agent services. These go beyond the original intentions.

On the other hand, there are a number of major drawbacks with polling:

1. Even in small and medium environments the number of resources and measurable attributes is very high (several thousand). If a manager has to establish (or keep) a network connection for each resource every time and communicate with the agent to retrieve a piece of information, the necessary network traffic is enormous. The tool-imposed overhead may degrade the performance and availability of the system.

   It may be possible to reduce this overhead if the manager has only one agent instance per host or one per server software (on each host) to contact.

2. Polling is a good idea if the point of time at which the manager needs new information can not be determined, for example if the demand for new data depends on user intervention (when the user opens an MIB window on the management console).
   If the information is requested in defined intervals and the timer event is the only reason to request load data, the timer at the host can be used and the network overhead omitted.

3. The manager is responsible for the detection of new resources. However, this involves some network broadcasts which are rather expensive.

4. As the detection of resources is based on services of a certain network protocol, it would be very difficult to support several protocols at the same time. Besides the further increased network load, if broadcasts were sent via all supported protocols, it would the question at which request a certain agent should react on would need examining. The agent would receive several requests and possibly answer one before receiving another request via another network stack.

5. Looking for resources no longer available on the network generates waste traffic (as it will not yield valid data) and usually locks the caller until some time-out interval has passed. The monitoring of remote LAN resources, mentioned before, is plagued by that problem. Even if done in parallel, a number of disappeared resources or the failure of a subarea in the network can send the manager to a hold.

6. Whenever several managers are active in the network they multiply the network overhead because each of them generate the traffic mentioned before. This same problem would exist if each agent sent its data to every manager.

Of course, all problems could be solved based on polling, but the intended monitoring tool would not be at an advantage by using that scheme. This leads to the following finding:

Finding:
Very soon it became clear that *polling* was **not** suitable for this kind of monitoring. An application which has to contact hundreds of agents each minute would flood the network with request packets and corresponding answers.

This was also reflected in commercial network and system managers. Monitoring functionality was moved to the agent which generated traps or logged information to a repository.

Because of this finding a more effective way of communication has to be developed. Chapter 5. "Communication Infrastructure" describes this infrastructure in detail.

The SRVMONPM agent is triggered by timer events. The time intervals can be adjusted and therefore the overhead of the monitor is controllable. The collection server makes sure that all agents reporting to it run at the same sampling interval in order to get comparable measurements.

4.3. Agent Concept

The agent is a very simple program which executes on a workstation and is able to create and handle **monitor objects**. It does not require user or operator intervention. Therefore it does not have a user interface. It is a very efficient and slim program, consuming as few resources as possible. To be able to create monitor objects it relies on **monitor modules** which are loaded dynamically at runtime. The agent does not have its own intelligence and does not implement any kind of knowledge about monitoring. Figure 8 depicts the relation between the basic

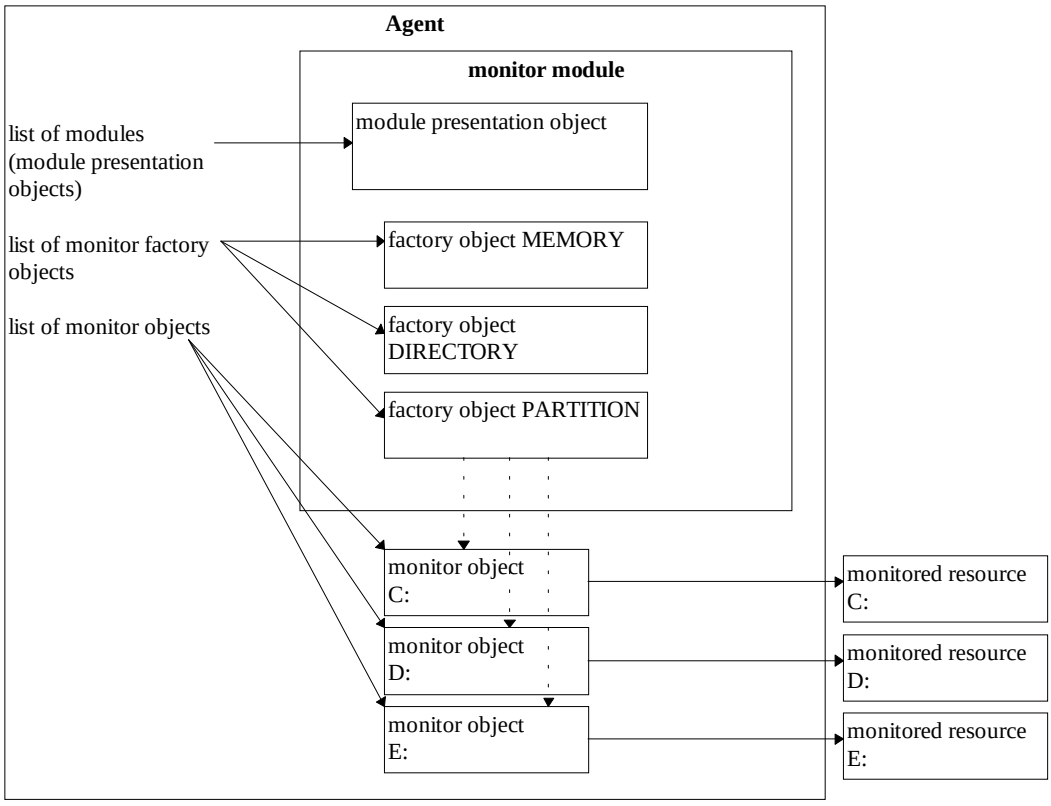


Figure 8. Elements of the agent concept

elements of an agent.

The knowledge about monitored resources lies in the *monitor module*. For each target application (which provides resources to other applications), a monitor module must be provided for the agent (and, as we will see later, for the manager). From the agent's viewpoint the monitor module provides a **module presentation object** that is used by the agent.

The *module presentation object* is the means of communication between agent and monitor module. The agent calls methods of this object to get the information and objects it needs for operation. Most importantly, it provides a list of *monitor factory objects*.

Each *monitor factory object* implements knowledge about a certain aspect of an application or resource. The main task of the class is to provide a description about the expected data and to implement a method which creates **monitor objects**. This method has to find resources (resource discovery) and create objects (object factory).

A *monitor object* implements the knowledge on how to retrieve workload data about a certain resource (e.g. the filesystem of a disk). If asked for new information it retrieves the actual data and returns them to the caller. It may store status information about the monitored resource but it does not maintain data from the past (it is not possible to ask "what happened one hour ago?").

The code associated with the classes mentioned above is loaded and executes within the context of the agent process. This means that it operates within the address space and process resources of the agent process. Therefore no additional process overhead is put on the host system.

This concept differs from conventional agent software which either has all functionality built into the agent code without the possibility to enhance its function from outside, or "subagents", (independent processes), which may register themselves at the agent. The (main) agent then delegates requests to a subagent.

4.4. Agent

The agent is basically an empty shell which does not implement much functionality. Using OO-techniques helps to keep the agent very simple and small. Consequently, the code is very robust (no errors recorded during an observation period of about six months) and can be ported to other platforms very easily.

The following paragraphs describe the main tasks of the agent, what the agent does on a machine and how the memory management for the monitor objects is done.

4.4.1. Main Tasks

Before we look at the tasks of the agent we have to understand some general observations about the agent. As seen below, the agent does not have much functionality by itself. It only makes sense together with dynamically loaded code which contains the knowledge about monitored resources. This code is organized in object-oriented classes. As we will see later, each class implements certain methods for initialization, detection and data retrieval. The knowledge about the target resource lies in the methods that access the information about resources. Data retrieval may be done locally if resources exist on the same computer, or by remote if resources are defined (and reachable) on distant machines. Both can be done under the control of monitor modules which are loaded by an agent.

4.4.1.1. Controlling the Scope of Monitoring

Before monitoring can begin, the tool has to find hosts that support monitoring and resources at these hosts. If the agent has been installed at the host and instructed to look for local resources, then it is clear (to the agent) where to search. For resources that should be monitored remotely, the task of detection is more complicated. For this case we further assume that no agent is installed at that remote host and we have to rely on other means of finding such a server machine.

Each instance of a monitor object has to know on which host machine its target resource is located. In general this information is needed to organize the data within the monitoring tool. Some monitor objects can use it to access remote resources over the network. When a monitor object is created, an object, representing the address of the target host, is handed to the constructor of the object. Given the fact that an individual instance of a monitor object must know which machine it has to deal with, the question arises of who can tell which servers are available and should be monitored.

In order to enable remote monitoring, the destination server application must support the remote access of its workload data. If that is the case it helps, if the server application has a function able to provide a list of available hosts. In addition to such a list an operator may want to include or exclude certain hosts. If this is also implemented one possible solution is to let each monitor factory object decide which servers from the list of available hosts can provide data for its monitor objects. This has several drawbacks.

Each factory class has to implement a way to detect a suitable server. Factory classes for the same application may use the same code but because classes are very independent from one another, each class has to keep track of its servers. A server machine for one application may not be a host for another application (e.g. a LAN Server may not be a DB2/2 server). Looking for a server can be very expensive in terms of run time. If this step is done for each class independently the agent may become extremely slow under certain conditions.

If each class handles its own set of known hosts it becomes difficult for the operator to control the set of possible target hosts (in reality the operator may reduce the number of servers or may add machines which cannot be detected automatically).

In order to avoid these problems and to offer more flexibility to the operator, the agent implements a mechanism to control the scope of monitoring - both in functionality (= classes) and number of servers.

The agent can be used in only one of two modes: **local** or **remote** monitoring. In local mode the agent monitors resources that are located at the same machine where it is executing. In remote mode the agent maintains a list of known hosts. If automatic host detection is made possible, the module presentation object of each monitor module is asked to return a list of hosts. These hosts are shared by all classes. The necessary processing is done in regular intervals to be able to detect new hosts over time. Factory classes are freed from this task and receive the names of possible target hosts during resource discovery from the agent.

The operator controls the mode of operation by supplying a command line option: if the arguments "/s" - for automatic server lookup - and/or "/sf <file>"[15], where

[15] Both options can be used within one invocation.

<file>[16] contains a list of servers to be monitored, and are given in the command line, the agent works in *remote* mode. Otherwise it runs in *local* mode.

The monitor modules are thus distinguished in two groups: those modules which contain monitor factory objects for local data retrieval (type is *client*; used in local mode) and those containing monitor factory objects able to access data on a distant machine (type is *server*; used in remote mode). The developer of a class has to decide which group a new module falls into. If a class can be used for both local and remote monitoring, it should be put in a module for remote monitoring.

Because there is one list of servers, each class has to check whether the referenced machine supports the functionality it needs. That happens more or less automatically when a class tries to detect resources on a server and receives error codes.

4.4.1.2. Initial Load of Monitor Modules

First of all the agent has to load all the code which actually implements data retrieval. This code comes as independent DLLs[17] which are loaded and bound at run time. Without these modules the agent does not have any noticeable functionality.

The agent uses a description file to find DLLs and the entry points into each DLL. The entry point is a function which returns a pointer to an object of the root class *MODULE_PRESENTATION*. This file contains all known modules. A developer would need to register a new DLL in this file. If the agent does not find the file it uses a default table which is part of the agent code. This table contains all modules which form part of the product as it is delivered by the author.

Depending on the monitor mode the agent selects those modules which are needed for further monitoring. For each module the agent performs the following steps. If one step fails, the module is removed and ignored in later processing:

1. load the DLL and the entry point;

   this may fail if the target application of the module is not installed (available) on the system.

2. query the module presentation object which represents the module (class MODULE_PRESENTATION) and set the OS/2 module handle for that object (the module (DLL); the handle is recognized by the loader of the module and is given to the module presentation object, because it will need it to access resources of the DLL.

3. use the object to initialize the module; this may include some steps to prepare the target application for monitoring (e.g. necessary for DB2/2 V 2.x); the object answers whether the destination server application is ready for monitoring.

4. retrieve a copyright statement from the object and display it on the console.

5. add the module to the list of supported monitor modules.

[16] Using a names file offers the possibility of reducing or enlarging the set of monitored servers. Automatic server lookup works per domain; up to five domains can be accessed from one LAN server client.
[17] Dynamic Link Library

4.4.1.3. Initial Load of Classes

After monitor modules have been loaded the agent asks each module for the classes it provides. This step is done once during startup. The number of classes may be dynamic (a module may simulate different classes which are based on the same root class) but it is not possible to add or remove classes later on.

The module presentation object is used as the "*object factory*" for monitor factory objects. Depending on the actual implementation such classes can be constructed at runtime or during the initial module load. The latter case occurs whenever instances of monitor factory objects are defined as static variables inside the module.

4.4.1.4. Setup of the Communication Infrastructure

The agent has to make sure that there is a collection server to which it can report definitions and data. The agent uses services of the communication infrastructure, described below, and is shielded from the complexity of the network. It only has to choose which communication object (and thus, which underlying communication method) should be used.

The default communication method is via standard (OS/2) *named pipes*. The operator may select another method by specifying a command line argument (e.g. "/netbios")[18].

The agent has to know which server is the partner for further communication. Either the operator specifies the name of the server via a commandline parameter ("/m <server name>") or the agent asks the communication object for a suitable partner. The second method is preferable because it makes the installation of the application simpler and the operation is more robust. If the destination server is very busy or breaks down, the agent may find another server automatically without operator intervention.

At regular intervals the agent checks whether the server is still available and able to receive information. When this check fails the agent tries to find another server. If a certain server name is defined by the operator, the agent has to wait until the server becomes available again.

No monitoring can occur at the agent's machine as long as no link to a server is established.

4.4.1.5. Resource Discovery and Monitor Object Creation

Finding all the resources to be monitored or managed from a central management site is one of the biggest challenges in systems management. Because of the great number of such resources it is not feasible to define and register them manually. This would be too error prone and would not form a solid base for further management operations. Therefore **automatic discovery** is a necessity in this area, albeit a very demanding task for the developer of systems management applications.

Each monitor factory object has to implement the knowledge on how to find resources and how to access information about them. Basically, we implemented three strategies for the classes to do this:

[18] Because the code which implements the communication object is loaded dynamically, it would be possible to add even more flexibility for the operator and allow for the specification of the module name which should be used for communication (e.g. "/commobj=SRVMCMNB.DLL"). This has not been done yet because an operator would not be able to write his own communication module in C++.

1. The target application supplies the information about available resources. Possible interfaces are an API or a configuration file which can be interpreted by the code. E.g. the IBM LAN server provides most information via an API. Therefore it is quite to detect available resources. The only problem is that the agent needs sufficient access authority (usually *system administrator*) to get hold of the data.

2. The class uses some kind of *trial-and-error* method to test the target application. This approach is feasible if the number of possible resources is small and if the operation of the application is not disturbed by queries concerning non-existent resources. E.g. this method is used to detect local disk partitions because it is simpler than the native interface and works for different versions of the operating system. The API does not.

3. The operator has to manually configure the class. As stated before, this should be avoided wherever possible but there are exceptions to the rule. In all circumstances a class has to provide a sensible default. Manual configuration is then an added benefit for more flexibility. However, the operator may choose not to bother with this extra burden.
   One example is the class *Directory Tree*. Its default is the monitoring of all file systems on all local disks. This may put undesired load on the target system. Therefore the operator can specify which parts of which file system he wants to monitor and the resource consumption of this class can thus be limited and controlled by the operator.

Under the control of the agent the classes use one of these strategies to detect available resources. The monitor factory objects create a monitor object for each discovered resource and return it to the agent. The agent has to handle memory management and processing of all monitor objects.

4.4.1.6. Data Retrieval and Preprocessing

At regular intervals the agent iterates through the set of recognized monitor objects and requests new data from them. The result is stored in a memory structure associated to the machine (server) of the monitored resource (see 4.4.3 "Memory Management"). By using the class DATAITEM the agent can assure errors and data overflows being handled correctly.

4.4.2. Agent Processing

The agent implements two important processes:

1. the data retriever and transport mechanism, which is its main task.
2. a control and restart mechanism added for robustness and unattended operation.

4.4.2.1. Data Retriever And Transport

The basic idea of the agent's engine is simple: in a loop the agent waits for a timed semaphore. When this has been triggered the agent updates its data buffers and sends it to its collection server. Figure 4 depicts the engine state diagram:

The process consists of two nested loops. The *outer loop* is responsible for the detection of a valid collection server, which can be used to send information to. Until a new server is found, or a known server contacted, the program "sleeps" for a minute then tries again, alternately. The agent is able to discover available servers in the network. For this function it relies on a low level service of the

communication infrastructure of the tool which is described in chapter 5.6.1 "Automatic Server Lookup".

When an available server has been found, the program enters the *inner loop*. After the first invocation and later, every 30 minutes, the agent updates all existing monitor objects and creates new objects for resources that have until now not been known. Note that monitor objects are never destroyed. They remain idle until their resource is active again.

After monitor objects have been created or validated, the program blocks on a timer semaphore. This semaphore is triggered once a minute. This happens independently from the number of processes blocking on this semaphore. Theoretically the semaphore has already been triggered when the program reaches this point and it then continues operation immediately. When the semaphore is triggered, all monitor objects are asked to update their data about resources. This information is collected and sent to the collection server. To do that, a connection to the server is established, a data message is sent, an acknowledgment received and the connection closed again. Following this a termination signal is checked. This signal can be set from outside to tell the agent to end its processing. If it is not set, the inner loop starts its processing from the beginning.

The program remains inside the inner loop until one of three things occurs:

1. During the attempt to establish a connection to a collection server, a communication error occurs. The program leaves the inner loop and the outer loop tries to find a collection server again.

2. An external program sets the termination signal. The inner loop detects this, sets a "incidental termination flag" and the process is terminated. It may take up to a minute between the time the signal is set by a program and the time the inner loop detects it.

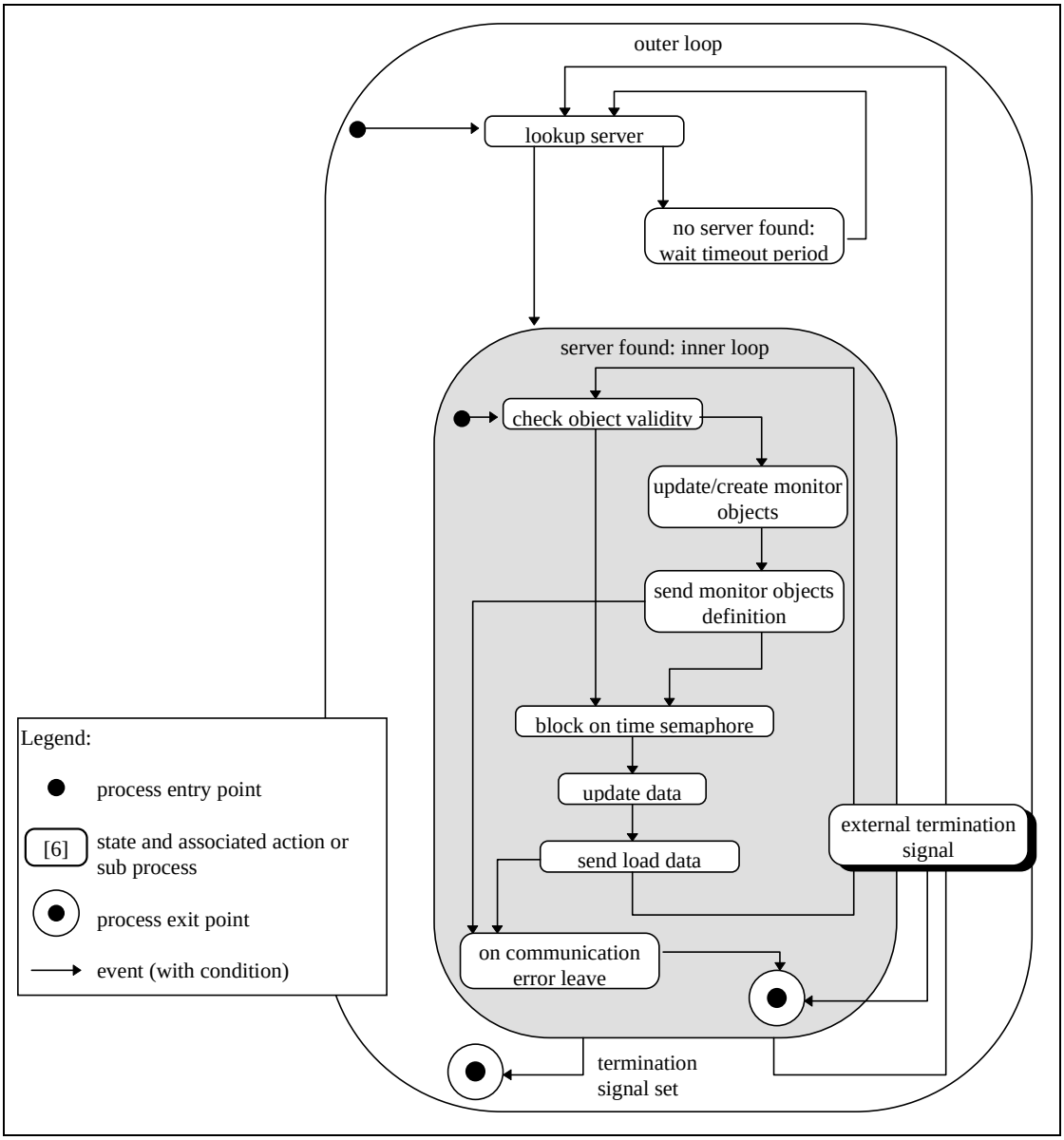


Figure 9. Agent state diagram

1. The operating system aborts the process due to some other reason, e.g. a call to an API of the destination server caused an access violation.

In order to reduce network load the agent checks whether object definitions or data values have changed during an update. If not, only header information is sent to the collection server. In reality this does not prevent data blocks being sent each time, because some data item will certainly change. The agent uses local timestamps[19] to guarantee that no outdated information is sent across the network (see chapter 5. "Communication Infrastructure").

Imbedded in the two loops is a check for an external signal which is used to terminate the agent via a program (*SRVMSTOP*). This program sets the termination signal (a shared named event semaphore) and waits for the agent to terminate. The command is useful for all automated tasks which cannot allow or afford the agent to be active, for example, a database backup procedure may require that all activities on the databases halted and no logon exists during the backup. In this case the backup procedure has to stop the agent before backup starts and restarts the agent after backup has finished.

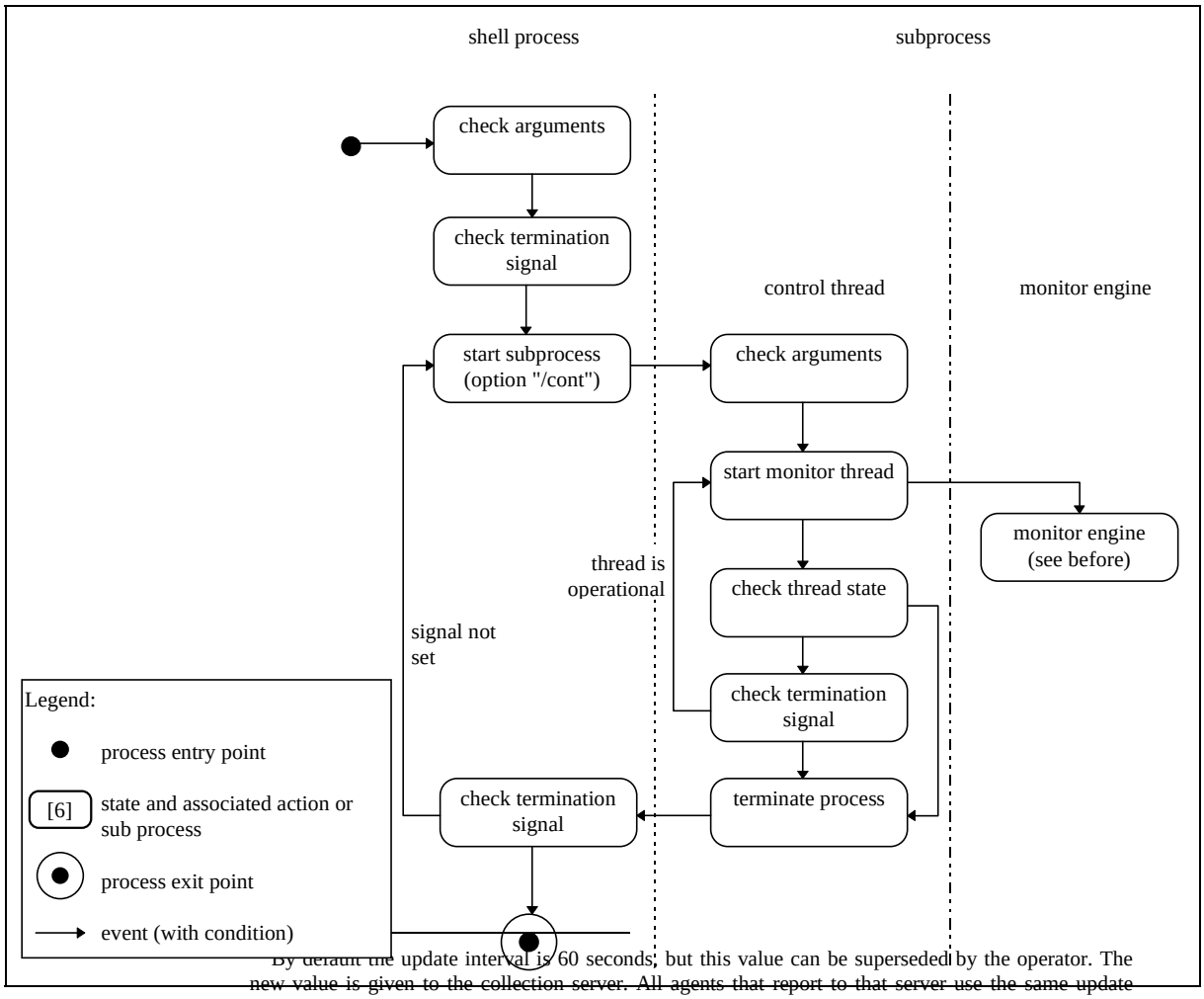
[19] The use of global timestamps was completely omitted.

4.4.2.2. Control and Restart

Usually the agent executes at remote server machines without the attendance or observation of an operator. It is critical to the success of the monitoring application that the agent runs permanently and delivers accurate data. During the case studies it became apparent that a "normal" program was not able to run continuously. Due to low level network errors and problems in other components of the computer, the agent came to a halt already after only days or weeks of operation. In such a case, the process had to be killed and restarted. To circumvent the problem and to guarantee continuos operation, the agent contains a control mechanism which is able to check that the agent is still operational. It can terminate and restart the engine in case of error.

At first we thought the core of the agent was so simple that it could not fail. Reality proved us wrong. As mentioned above, the agent went dead on some machines after an irregular time interval of several days. The agent did not trap nor did it produce any other detectable error but simply felt asleep and did not do anything. An operator had to terminate it via *<CTRL-C>* (which always worked - thus it did not get stuck due to a problem with the operating system).

Nobody knows yet why this happens but we learned that other applications which run in a 7x24 mode under OS/2 suffer from similar problems. At the first glance the cure to the symptom seems to be very simple: the main thread uses a variable to notify its current state. A second thread sleeps for 1.5 times the update interval[20] and when it wakes up it checks that the state of the main thread is changing over time and that data updates are being sent to a server. If not, the second thread uses



By default the update interval is 60 seconds, but this value can be superseded by the operator. The new value is given to the collection server. All agents that report to that server use the same update interval.

Figure 10. Control mechanism of the agent

the OS/2 API **DosKillProcess** to terminate the process. In order to recognize and react on that a shell process is needed, which starts the agent. The control thread terminates its own process in case of error. The shell process captures the termination of the child process and starts another agent process. Figure 10 shows the flow of control between the different entities.

Two separate processes are actively running to cover unexpected program hangs: The process that is started by an operator or by a startup script of the operating system, is the shell process. It initiates and checks/resets the termination signal. Note that this signal may already exist and may be set due to a prior program invocation. A subprocess is then started, which implements the monitoring. The shell process uses an OS function to block, until the subprocess terminates.

The subprocess consists of two threads (see appendix). The main thread (the thread that is started by the OS when a process is created) is the control thread. First it initiates a control data structure (position indicator, loop flag, blocking counter, all of which are used to check whether the monitor engine is still operational), and then starts a second thread for the monitoring. The second thread - the monitor engine - was described before (see Figure 9). As the monitor engine progresses through the steps it has to do, it sets a position indicator that signals the current position within the engine to the control thread. In addition, after each iteration a loop flag is set.

The control thread weakens after 1.5 times of the update interval (usually 90 seconds) and checks, whether the position indicator has changed or if the loop flag has been set. If that is the case it can be assumed that the monitor engine is still running. The loop flag is reset and the blocking counter set to zero then the termination signal is checked (see before). The control thread blocks again.

If the control thread detects that neither the position indicator nor the loop flag have changed then it is assumed that the monitor engine is blocked by some operation. A warning message is displayed at the operator console and the blocking counter is incremented. If the blocking counter is greater then 3 then the process is terminated (as explained before). If the blocking counter is less or equal 3 then the control thread blocks again. Some operations, e.g. analyses of the file system of the server, can take more time then 90 seconds. Therefore the blocking counter is used to allow for a certain excess of the update interval, while still being able to detect (and report) a potential problem quickly.

Both the shell process and the subprocess are implemented by the same program: the agent itself. An additional commandline argument "/cont" is used (for the subprocess) to distinguish parent and child process invocation. The argument may be used by an operator if he does not want automatic restart. When the child process terminates the parent process also has to check the external termination signal, too. If it is not set the control process assumes that the child process terminated due to an error and it restarts it.

4.4.2.3. Calling a Detached Process or Function

For several classes of information there is no API that can be used to access information. In these cases the character based output of other programs is used and interpreted. To get the information the other program has to be called and its *stdout* handle must be mapped to a pipe. The agent can read the pipe and work with the output.

There are several problems with this approach, however. The most significant is that reading the pipe blocks forever if no more data are in the pipe and the calling process is about to terminate. There seems to be no way to prevent this. In order to

circumvent the problem another process - the transmitter - is put into the middle of the operation:

1. The transmitter is started as a "normal" child of the agent. Therefore the agent has full control of it.

2. The transmitter calls the target program, maps handles to pipes, captures the output from the program and translates it into a useful format. These data are put into a shared memory area. The transmitter releases its CPU slice in order to give the target program a chance to produce some data and reads the pipe in blocks of 1024 bytes after each pause. If less than 1024 bytes are returned it assumes that no more output will follow and stops reading the pipe. This has two consequences:

a. the transmitter will block if the target program returns a multiple of 1024 bytes, which can occur.

b. if the target works too slowly the interpretation of its output stops early; such a slow data source is not usable by the monitoring system.

If this occurs the transmitter posts an "event semaphore" that signals the success (or at least termination) of the operation to the agent. An "event semaphore" is an OS/2 construct the allows a number of threads (processes) to wait for the arrival of an event. Another thread "posts" the event semaphore. That starts all waiting threads. A waiting thread can specify the maximum time it is willing to wait. If the time expires and nobody posted the semaphore the thread is started but receives a return code indicating that the time expired.

3. The agent blocks on the event semaphore with a certain time-out value ($9/10^{th}$ of the data update interval). This waiting may result in one of two possibilities: the semaphore is posted (see paragraph above), the operation has been successful and the data can be copied from the shared memory area. The other possibility is the semaphore API returns a time-out error, the transmitter is blocked (because no or invalid output has been produced by the target program). In this case the agent kills the transmitter.

4.4.3. Memory Management

One important goal for the design of the agent code is to keep resource consumption at a minimum. Therefore the memory structure is kept very simple and small. The agent's memory management routines have to handle two important data structures: the monitor objects, which are provided by the monitor factory objects, and a memory area where load data are stored until they are transmitted to a server.

Along with the definitions of the known hosts (the "machine definition"), a structure of type INFOBLOCK_DESC is created. This is the same structure used by the collection server for storing the load data. INFOBLOCK_DESC is able to store a number of load data items together with descriptions of the objects that provide this data.

The monitor objects are generated by the discovery code of the monitor factory objects and are handed to the agent. Because a monitor object has no knowledge of INFOBLOCK_DESC and does not know where to store its data, the agent provides an object to establish a connection between RETRIEVER and INFO-BLOCK_DESC: the class **OBJ_STORAGE_REL** contains a pointer to a retriever, a pointer to a machine definition and the index into an array of data items to which the retriever may write its data.

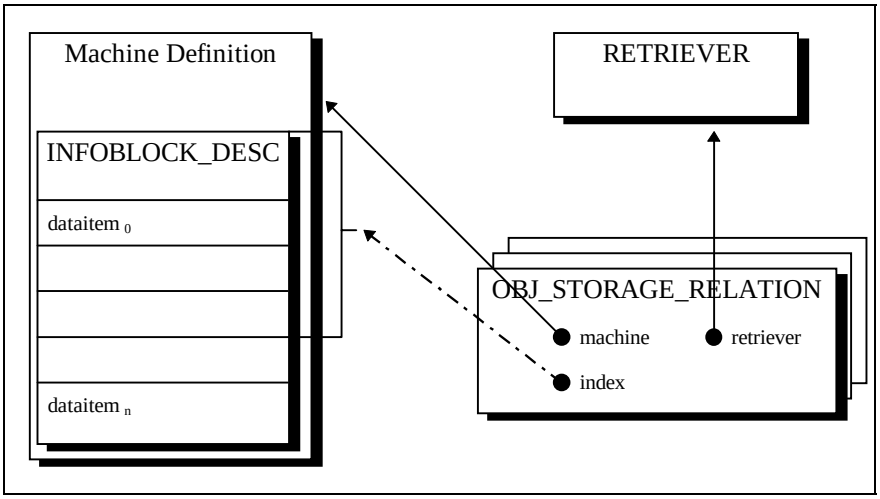


Figure 11. Relation between monitor object and machine definition

Whenever the agent receives a new monitor object it creates a storage relation object and registers the retriever at the infoblock. The agent maintains a collection of machine definitions and a collection of storage relations. All other structures depend on one of the two.

At regular intervals the agent checks the availability of monitor objects and machines. If they are no longer available for a certain amount of time the structures are removed from memory.

4.5. Monitor Module Structure

A monitor module can provide following objects for the agent:

Class	Mode	Card.	Comments
entry point (C function)	mandatory	1	returns an object of class MODULE_PRESENTATION
MODULE_ PRESENTATION	mandatory	1	core interface between agent and monitor object; provides methods to return: - a copyright statement - a number of role[21] definitions - a number of monitor factory objects - code to initialize monitoring
Role Definition	optional	n	the module must contain icons for each role which are used for display in the user interface
Monitor factory object (CLASS_DEF)	optional	n	a monitor module should return at least one monitor factory object
Retriever Class	optional	n	for each monitor factory object a retriever class

[21] Refer to chapter 6.2.5 "Machines and Roles" for more information about "roles".

Class	Mode	Card.	Comments
			must be provided

4.6. Data Transport

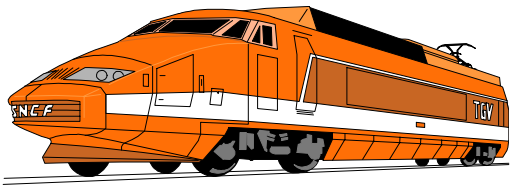
Critical to the successful operation of the agent is the efficient transmission of the retrieved data to the manager. On the one hand, it should be reliable; on the other hand, it should not put much extra load on the systems and the network.

After the agent has been started it tries to connect with a collection server which is either specified by the administrator or discovered automatically in the network. If a (new) server has been contacted after startup, in the case a connection to the known destination server being lost or object definitions changed, the agent registers itself and all (new/changed) object definitions at the server. This information is not transmitted over the network again as long as the logical connection between agent and server is not interrupted. When a data lookup has been successful and has delivered new data, the INFOBLOCK_DESC memory blocks are sent to the server. This structure contains only raw data, without extra communication overhead[22].

For more details about the protocol and the data sent across the network, see the next chapter.

[22] The information is not compressed. If necessary that task must be done by the network software or hardware.

5. Communication Infrastructure



An efficient communication infrastructure is one key for successful load monitoring in a distributed system. Therefore it is very important to understand the components which make up this infrastructure and how they can be used to satisfy the requirements of the IT shop. The implementation of these components offer great flexibility to adapt the infrastructure to the needs of the underlying network.

5.1. Introduction

5.1.1. The Problem

A great number of monitored machines has to communicate with receiving managers. That can create significant network traffic and in addition, slow WAN links and loaded networking devices may lie between them. Things become worse if there are several managers at the same or different locations.

How does an agent know where to send its data? How can a manager handle hundreds of requests per minute? Can it be an OS/2 machine or are UNIX and MVS the only suitable platforms? What transport protocol should be used or must be provided for agents and managers?

5.1.2. Common Implementations

We know of just two implementations in commercial products. The common factor between them is their reliance on a direct connection[23] between agent and manager in order to exchange data. Therefore they suffer greatly under the problems mentioned above and cannot be used for reliable monitoring in large networks of OS/2 servers.

The first implementation works like an SNMP manager: the manager initiates a data update by sending a message to the target agent. In return the agent answers and sends some data. This has to be repeated for each known agent. Broadcasts may be used to simplify the task for the manager but that delegates the work to routers and other devices and brings new problems to the network . Nevertheless, the principle remains the same.

Both the manager's host and the network must be very powerful to handle the task or the number of monitored agents will be reduced radically - the one or two machines which are in the focus of the administrator.

If more than one manager console is active the whole process is also performed. Thus they multiply the load on the network and the agents, as they have to respond to several similar requests which arrive independently from one another.

[23] That implies that both sides have to (install and) use the same network protocol.

In the second implementation variant it is the agent which becomes active and establishes a link to the manager or sends messages. E.g. *NetFinity* or the *LAN Management Utilities*, both products of IBM, use this scheme. The manager must be prepared for many requests, many of them arriving at the same time. Today products are limited to about 50 agents which may report their data to a manager in a LAN.

If several managers have to be supported the agent must be able to keep a list of destination servers and send a message to each of them. Besides the network load no known agent can do that. For obvious reasons agents may not use broadcasts to distribute their data. Therefore either each agent is set up accordingly (a difficult and expensive process), or an extra agent-to-manager protocol must be provided to register a manager at all the agents.

Because the number of agents per manager is small many managers must be used to monitor all available agents. Each manager owns a number of assigned agents. Careful planning and setup is necessary for the agents and manager. A consistent (over) view of the attached systems is lost.

5.1.3. The New Approach

In order to support monitoring of accurate data (with small sample intervals) for a great number of machines, the need for a direct session between agent and manager has been eliminated. Instead, each agent puts its data into a temporary storage area within the LAN. Many agents are able to store their data here. Independent of the activities of the agents, the information can be propagated to other temporary storage units. The information is consolidated and packed into a small number of network messages. Reducing the number of session setups and messages is much more efficient than the use of many sessions and small messages. The most appropriate communication method can be chosen to get over the network link between the storage units.

Any number of managers may access the data at temporary storage units. The agents do not have to deal with the number and location of managers. A manager does not even know where the information comes from or how many storage units were involved in the transporting of the data. Each piece of data is sent only once on a certain link. There is no need to send it again.

The communication infrastructure consists of a server program (SRVMSRVR) and code modules, which are bound to and used by every part of the tool for communication. In this chapter we will look at the solution in detail.

5.2. Scenarios

As an introduction, and in order to simplify the following technical description we will look at the three possible standard installation scenarios first before discussing the infrastructure components in detail. The three scenarios build an ascending path which is be followed by most people starting to work with the monitoring tool.

5.2.1. Basic Installation

The basic installation (see Figure 7) is the most simple way to set up the tool. It works best for a single domain in a single physical LAN. The only necessary installation is putting the monitoring tool, with all its components, on a single workstation in the network. An *administrator logon* must be done on this workstation because due to security reasons, the "IBM LAN Server" monitor code requires administrative privilege.

At this workstation three components are active:

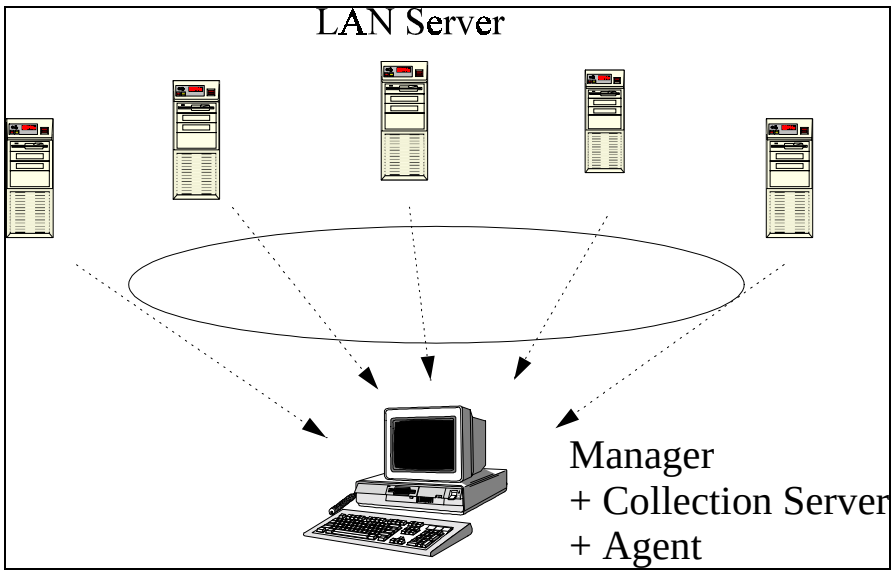


Figure 12. Basic installation within a single domain

- an agent, which accesses the data of LAN servers by remote in the domain,
- a collection server, which is always the bridge between agent and manager, and
- the monitoring manager[24], which collects and processes the data.

As all three components are executing on the same machine, no network is necessary for communication between them. Nevertheless this is transparent to the components and is handled by the network layer of the workload monitoring infrastructure.

One big advantage of remote workload data access is that the agent acts like any other "customer" of a LAN server. If the LAN server has a problem and is unable to respond to a request in time the agent will recognize this and consequently the manager will be able to trigger an alert. Therefore LAN Server information is retrieved remotely in all scenarios that are described in this chapter.

[24] Note that a manager always implies the existence of a collection server on the same machine. The manager is not allowed to communicate directly with any agent.

5.2.2. Full Local Installation

The afore described scenario is able to provide all the data which can be accessed remotely. In the current implementation all information about the IBM LAN Server product can be acquired this way. If the operator wishes to access all the available information the installation must be extended, and requires some additional effort. Note that the mechanism of the first scenario is still used: an agent at the manager accesses LAN server machines remotely. In order to do this it needs administrative privileges for the domains of the monitored machines.

Firstly the operator has to select one server which plays the role of a collection server. On this machine the collection server component (SRVMSRVR) has to be installed and activated. A possible alternative for this scenario is the use of the collection server on the manager.

On all servers (including the collection server) the agent has to be installed and activated. The operator may or may not make available the name of the collection server to the agent. The recommended method is **not** to supply the name. The agent looks for a collection server automatically (see chapter 4.4.2.1 "Data Retriever And Transport" for the processing of the agent and chapter 5.6.1 "Automatic Server Lookup" for the required low level service). This has two major advantages:

1. If the collection server machine is changed later no changes on monitored machines are necessary. This simplifies the operation of the tool.

2. In large domains the operator can activate more than one collection server on different machines. Monitoring is therefore made more robust, even in cases of server failures, and the load on the collection server itself is balanced automatically.

The only disadvantage is that, depending on the network access method, broadcast messages are propagated on the network to find a collection server and put extra load on the network. Broadcasts are used very economically to keep negative impacts at a minimum. Not all network access methods support automatic lookup; broadcasts must be made possible on the agent and the network domain. This may not be appropriate in every environment.

The collection server must be one of the machines known to the manager. In this setup all available data about the local domain are collected and displayed at the manager. In addition to LAN servers any network connected machine can send data to the collection server if it supports one of the provided network access methods.

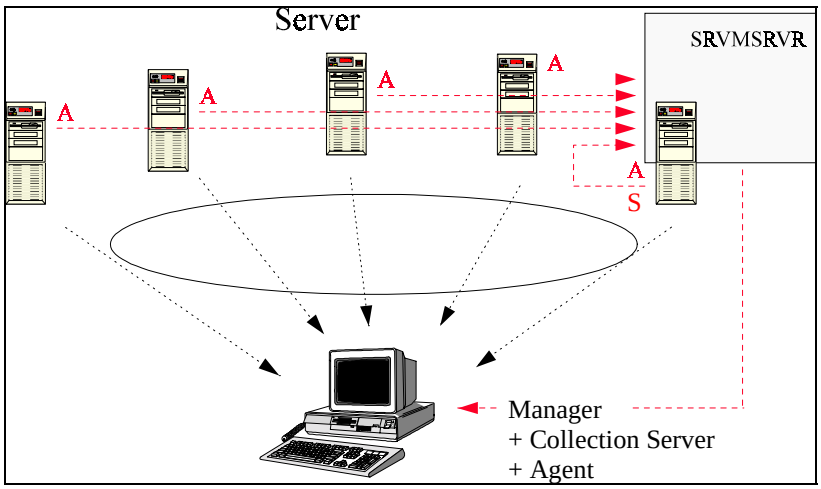


Figure 13. Advanced installation with agents on each server

Therefore other machine types like database servers or communication hardware can also be monitored.

5.2.3. Fully Distributed Installation

Both previously described installations are feasible for monitoring of machines in the same physical network where the manager is connected. Even within one physical network the number of LAN servers which can be monitored remotely from one machine in the network is limited:

- If the network is organized in several domains it may not be possible nor desirable to grant administrative privilege to one central machine (or person).
- Accessing the data remotely moves much more data over the network then are actually used. The agent processes and filters the information and hands on only a fraction of the data volume. Accessing servers over (slower or loaded) bridges can add noticeable network load.
- For the same reason it is not possible to do this over slower WAN connections.
- WAN links may not support direct LAN server access. The usage of WAN protocols must be available to the operator.

For these reasons the collection server can be used to hand on the data to another collection server.

Figure 14 shows how a remote domain collects its data and sends it via a collection server to a central collection server. It makes no difference whether the remote

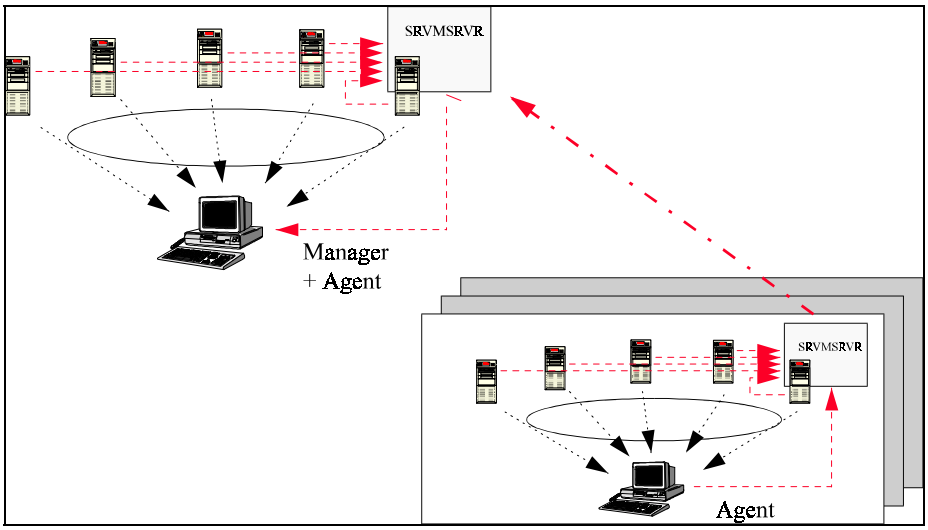


Figure 14. Full installation with remote LANs

domain is within the same physical LAN or connected via a WAN link.

At the central site there is no difference in the setup as described above. The difference in the remote LAN is that at the monitoring client workstation there is no manager and collection server active. Instead, an agent collects LAN server data and sends them to the collection server. Therefore this machine needs a different installation and preparation from the central monitoring manager PC.

Note that is possible to run a manager in the remote LAN too. The local administrator sees the data around his LAN. The collection server on the manager

must be set up in such a way that it functions as a collection server that sends its data to the central site.

5.3. Layers of Communication

To understand the way in which the tool uses a network to transport load data from a widely distributed system to a central management site we must distinguish several layers which make up the communication infrastructure.

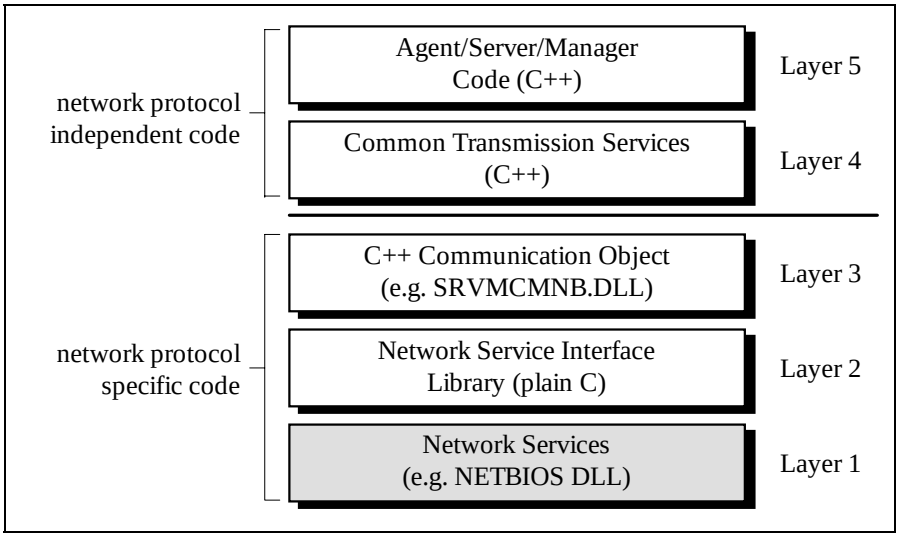


Figure 15. Layers of communication

As shown in Figure 15 there are five layers of code with different levels of abstraction used for communication:

Layer 1 is provided by the operating system or the system software, and is not part of the tool. Within this chapter the view on this component is simplified and we speak of one layer only. In reality - within the system software - there are several layers down to the network hardware. However in this context the structure of the piece of software is not of interest.

It is important that the (network) operating system provides some sort of service, which depends on the installed hardware and the network device drivers and protocols which support that hardware. The programming interface are C function(s) which are exported by a DLL. In some cases this interface is still a 16-bit interface, which makes its use a little more difficult.

Layer 2 implements specific functions based on the services provided by layer one. Basic functions for sending and receiving data are written in C using API functions of the network services. These functions are specific for the supported network layer and each *network service library* has a unique interface and special purpose data structures. One other task of this layer is to hide problems with data buffers and addressing with a 16-bit interface.

For each supported type of layer one there is a corresponding library in layer two.

Layer 3 implements a common interface based on the functions provided by layer two. This layer is very thin. Normally it does not implement any functionality but holds control information and transforms common requests into the structure required by layer two. The functionality of the transmission processing is kept in layer two.

The common interface resembles the functionality of a pipe in message mode[25] - known from IPC methods. Supported methods are:

1. open connection (preparation for transmission)
2. bind (wait for connection at the server side)
3. connect (wait for connection at client side)
4. read data
5. write data
6. disconnect (abort connection at server side)
7. close connection (abort connection at client side or end any transmission on server side)
8. lookup server (search for available servers at client side)

There is a corresponding C++ module in the third layer for each supported type of the first and second layers.

Layer 4 uses the common interface to implement the client/server protocol described below. The protocol controls load data's movement over the network and ensures that data are valid and up-to-date.

This layer exists only once. The types of supported layers below are (nearly) invisible to this layer. The only exception is that it controls registration and initialization of C++ Communication Objects.

Layer 5 contains the applications which use the other layers to communicate with one another. They only provide specific data which apply to the functions of layer 4. Network and protocols are transparent to the applications.

5.4. High Level Infrastructure Components

After reading the overview above we can now look at the technical details of the infrastructure components to understand how one can install and use the tool and what additional possibilities (of configuration) exist for special problems and situations.

First we will examine the tasks of the three major parts from the viewpoint of communication and data transmission.

5.4.1. Agent

[25] In general (named) pipes can be used in several modes (e.g. binary, stream, etc.). Because it fits best to the OO-paradigm of messages, here pipes are used in message mode only. Therefore data are sent and received in continuous blocks as messages. The pipe keeps track of messages sizes and blocking during the transmission of messages.

The agent, implemented by the process **SRVMCLNT**, has two major tasks: the **control of data collection** and **handling the network link** to a manager. In the following paragraphs we will look at the second task. In order to successfully move the data to a managing server the agent has to carry out following functions (note that each function is fully based on the service provided by layer 5 or eventually by layer 4):

1. Look up a server if no destination server has been supplied. If no server can be found the agent is suspended for about a minute. As long as there is no possibility to hand on data it does not make any sense to collect them. In case of error the agent will continue to look for a server and establish a connection.

2. If a destination server can be identified and data are ready for transmission, a link can be established with the destination server. If this fails the agent resumes with step 1.

3. If this is the first established connection to this server or in the case the definition of objects have changed since the last registration, the machine(s) and the monitored resources are registered at the server. The server acknowledges the registration. If one registration fails the agent resumes with step 1.

4. If the registration has been successful or the connection is not the first one (registration has already taken place) data are sent.

Figure 16 shows the states in which the agent can be and the method of transition between states. The following table explains the states and the possible transitions between the different states:

curr. state	meaning	new state	transition
1	no destination server known; look up takes place	2	searching for a destination server successful

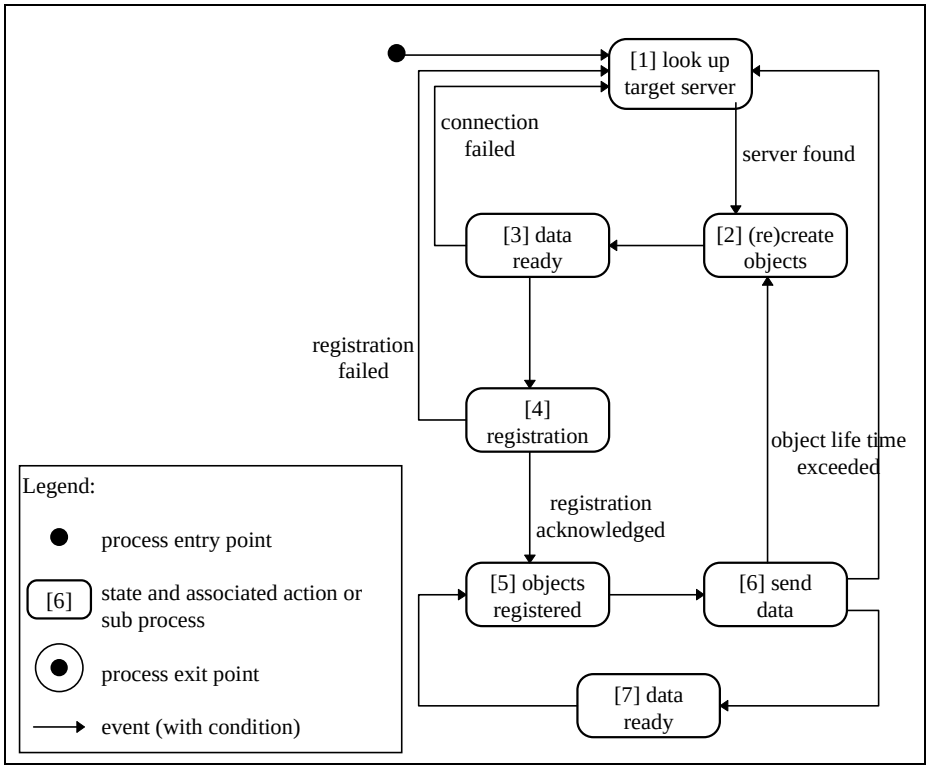


Figure 16. Agent communication state diagram

curr. state	meaning	new state	transition
2	a destination server is known, it is necessary to discover available resources for monitoring and create corresponding monitor objects	3	monitor objects created
3	an update of monitor objects taken place; data ready for transmission	4	a connection to the destination server established
		1	failed to establish a connection
4	there was no connection to the destination server, since new objects were created; the connection to the server was lost for some reason; objects must be registered	5	machine(s) and objects able to be registered
		1	destination server failed to register objects
5	objects registered at the known destination server	6	N/A
6	data able to be sent to the server	7	data able to be sent
		1	an error occurred during transmission
		2	object definition outdated
7	no error occurred before; data ready again	5	update interval elapsed

5.4.2. Collection and Intermediate Managing Server

The **collection** server, implemented by the process **SRVMSRVR**, is the backbone and most important component of the communication infrastructure. It is the bridge between agents and managers in the system. The collection server *collects* data from agents and possibly other servers and it answers requests from managers. The **intermediate managing** server is identical to the collection server but in addition to the tasks of a collection server it retransmits information to another server. If necessary a collection server may refuse to answer requests of an active manager within its network[26].

The different terms - *intermediate* and *collection server* - have been introduced to simplify the description of possible configuration scenarios (see chapter 5.2. "Scenarios).

Definition: **Monitor Domain**

is the set of all agents that report their data to the same physical collection server.

A collection server does the following tasks (that are described in detail in the following chapters):

[26] Example: the collection server handles data from several remote domains and replicates them on a central site. The local operator wants to monitor his domains only. The collection server can be active on one of his servers but does not interact with the local monitor management software.

1. registration of monitored machines and resources
2. delivery and temporary storage of load data
3. control of update intervals at connected agents
4. check of time stamps and data validity
5. consolidation of different data blocks for the same machine
6. replies to managers' requests (SRVMONPM)
7. retransmission of data to another server (intermediate managing server)

The functions provided by the process are based on layer 5 of the communication services. The process implements the server part of a true client/server application: a dynamic number of threads are active and parallel and each thread services a communication object. The communication object works like a pipe. It listens for connections, receives messages, processes data packed in the message, responds with the result, then closes the connection in order to become available and ready for new requests.

5.4.2.1. States of a Service Thread

All service threads are identical, execute simultaneously and are totally independent of one another. Therefore we can look at the functionality of a single thread. There are no side effects on other threads. The basic state diagram (Figure 12) of a service thread is very simple:

1. wait for a connection (listen)
2. connect
3. receive a message in a general „all purpose“ area
4. decipher the message header (message type/number and version) and convert the message into the correct message object (a C++ structure)
5. process the message and send a result
6. disconnect
7. continue with step 1

The service threads implement three nested levels as shown in Figure 12. The outer level manages the communication and the thread itself. If the thread can be initialized it changes to the second level, the "main service loop" state. In this state the thread is able to handle requests. It waits for incoming connection requests and handles them. If there are too many unused threads it leaves this state again. When an incoming connection request arrives it sets up the connection and changes to the "inner service loop" state. In this state it is able to handle requests that are send via the connection. When the requesting partner closes the connection it leaves this state and goes back to the "main service loop" level.

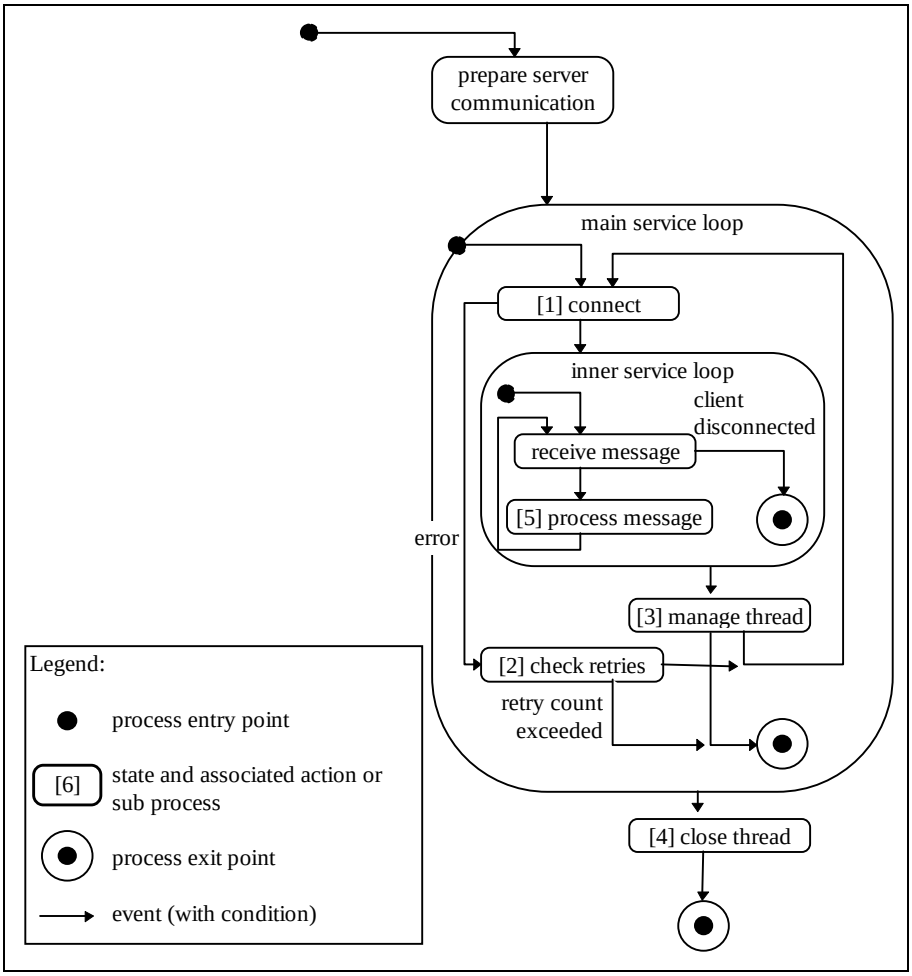


Figure 17. State diagram of the service thread of a collection server

The following table details the states of Figure 12:

state	description
[1]	the service thread is ready for a new connection; an agent may request one; if a connection is established the thread enters a loop inside which a number of messages from the agent are handled
[2]	the network report an error when the service thread tries to ready itself for connections; the thread waits a moment and attempts to connect again; a retry counter is increased for every failed connect; if it is not possible to resume operation the thread gives up and closes down; otherwise the retry counter is reset to zero
[3]	the number of threads is dynamic; when a thread leaves the message processing loop a check is made to see if it will be needed; if there are too many idle threads, it is closed down
[4]	for reasons mentioned above, the thread is about to close down; it shuts down network activities, frees its resources and ends
[5]	a message received in binary format is "transformed" into a message structure and processed by the thread (see below)

5.4.2.2. Tasks of the Collection Server

The following paragraphs describe the main tasks of a collection or intermediate managing server process.

Registration of monitored machines and resources

The collection server should know which machines (agents) it is responsible for[27]. The internal management of storage and definitions depends on this information (see below). In addition the manager has to know about the resources available on the machine. Therefore each agent has to register itself (the machine where the agent "lives"), the destination server (where the resources sit) and the monitor objects (the resources) at the server before load data can be transmitted.

The process maintains a set of registered machines. A machine is identified by its name (a 16 byte character string). For each machine up to four[28] *source agents structures* are stored. A source agent structure represents **the process which sends data to the server** and is identified by the machine name of the source agent plus a one character extension which defines the type of monitoring (remote or local data access). It contains the object (resource) definitions and a memory block for all load data items provided by the agent.

If the monitoring application is set up completely and correctly two source agent structures will be associated with the machine: one for the agent which runs locally at the target machine and does local monitoring. In the case that the target machine

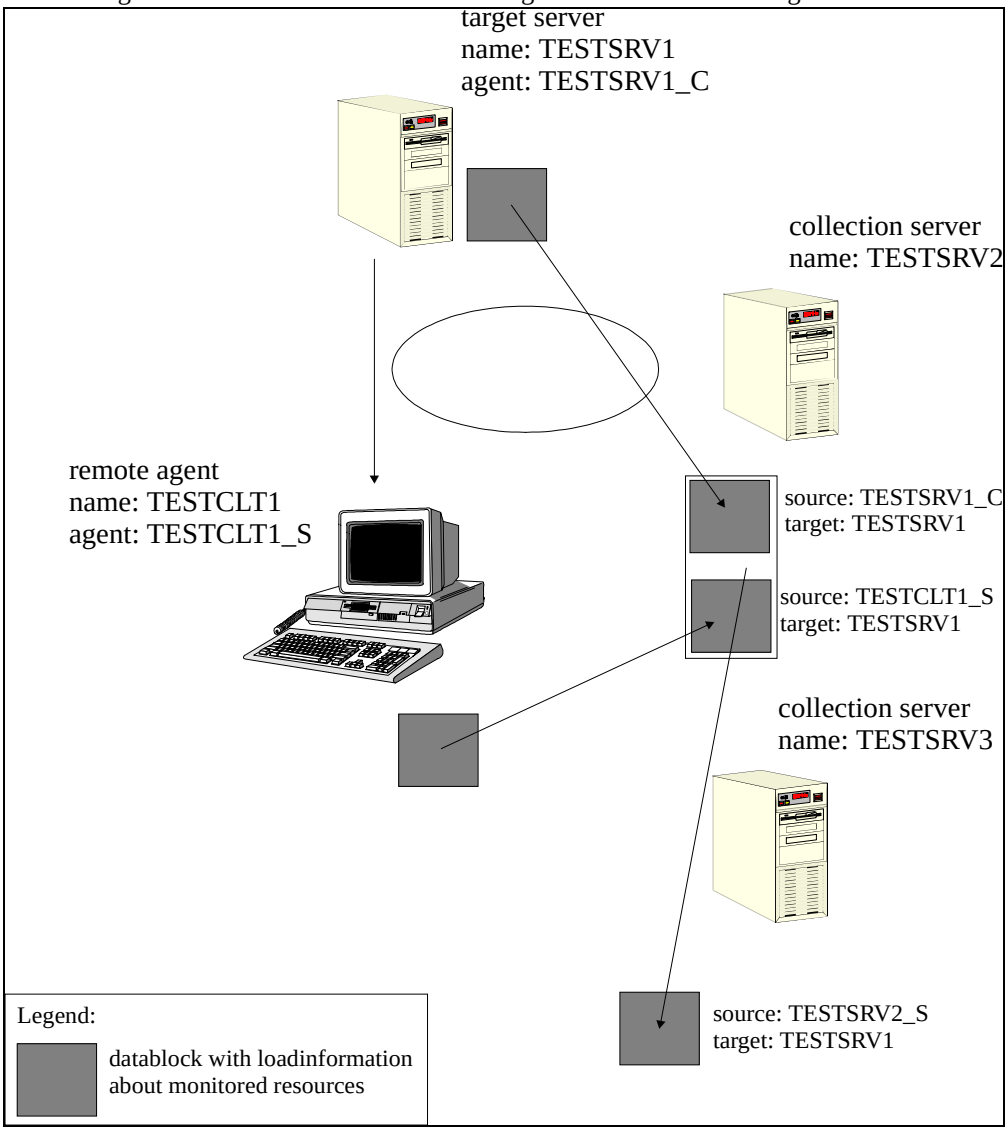


Figure 18. Replication model

[27] The same machine can register its definitions on more than one collection server. This is not very sensible but the program is robust enough to handle such a case.

[28] Normally one or at the most two source agents can be present. In cases of wrong manual configurations or due to future extensions more agents may report information on the same machine.

is a LAN server, a second agent, which runs on a different machine, does remote monitoring[29].

Figure 18. shows a simple example of a server that is monitored by both a local and a remote agent. A local agent at server TESTSRV1 monitors information about operating system resources and a remote agent at the machine TESTCLT1 monitors network related information about TESTSRV1. Both agents send data about the *target server* TESTSRV1 to the collection server TESTSRV2; the collection server stores two data blocks about TESTSRV1. Before it forwards the information about TESTSRV1 to another machine, in this example TESTSRV2, it consolidates the information and forwards only one block of information. From TESTSRV2's point of view it behaves as it were an agent that sends data and TESTSRV2 does not know anymore that there exist two sources of information for TESTSRV1.

Figure 19 shows the data structures that are used to store workload data at the server. The server maintains an array of registered machines (the registration area). Each entry in the area points to a machine definition that contains some general data about the monitored machine and a list of references to (source) agent data structure. There is one such structure for each agent that reports data about the machine. The source agent area contains a reference to the start of a list of resource definitions and the index into the machine related load data storage area. Load data arriving at the server on behalf of the agent will be copied into the data area starting at this index. A resource definition consists of the name, class, version of the resource and an offset into the load data storage area, where the manager can find the data for this object. See chapter 5.4.2.3 "Memory Management" for details on

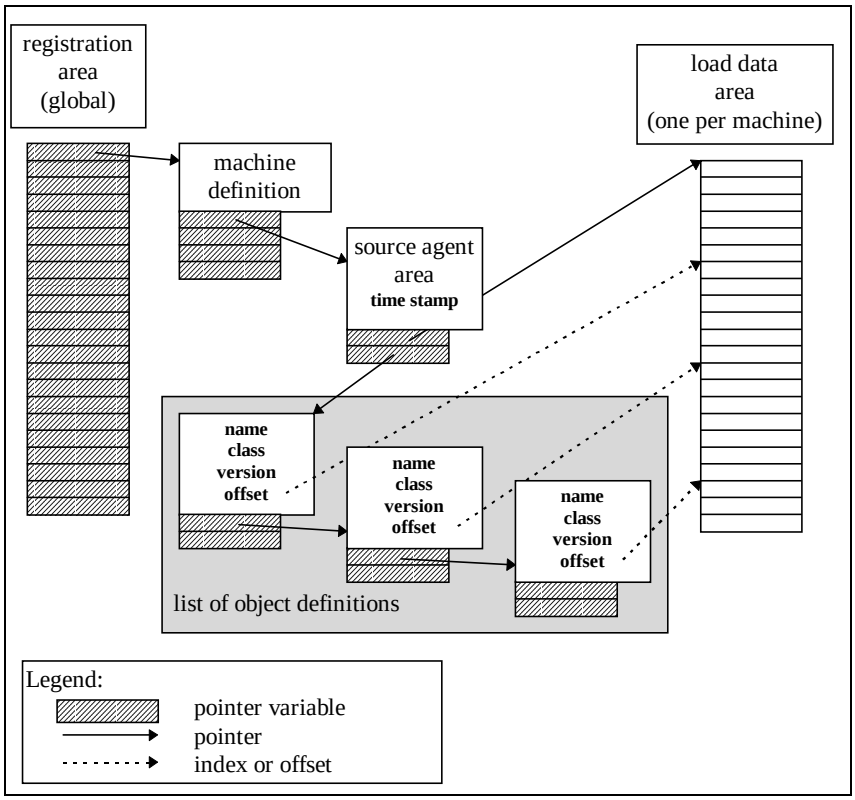


Figure 19. Storage structure for the collection server

these data structures.

[29] Theoretically a third agent could also do remote monitoring. It would be associated with the machine and it would provide the same information and data as the second agent. This error does no harm to the monitoring system but it is a waste of resources and does not gain any additional value.

If the collection server receives a registration for a machine that is already registered, the old definition is replaced by the new information. There is no special notification about this fact.

An error occurs only in those cases where the server runs out of memory or space for machine registration. This space is limited by a number allocated during compile time. It cannot be changed during later operation. In the current implementation this number is set to 1024, meaning up to 1024 machine definitions can be handled by one collection server.

A machine definition and all dependent information is removed if, for some time, no update for its data has taken place (see below). It is not possible to „unregister“ a machine or an object. Before an agent registers new object definitions for a destination server, the existing definitions are removed. Therefore unused resource object definitions are removed on behalf of the agent. For more details see chapter 0. "Service Thread Management".

Delivery and temporary storage of load data

After an agent has registered itself and the resources it monitors, it sends only the load data. When new data arrive the server looks up the machine definition and the source agent structure. If these structures are available at the server the message body is copied onto the data area associated with the agent structure. Otherwise an error is returned to the agent which indicates that registration is necessary.

This kind of error can occur if the collection server has been restarted while the agent has been stopped. Thus, it has not noticed that the server is not available and tries to send its data.

When new data are copied to the data area a time stamp is set at the machine definition to indicate that new valid data have been received and that there is still an agent in the system which sends data about the machine. If the timestamp becomes outdated the information in the load data area will be considered invalid and not worth delivering to another server or manager.

Control of update intervals

The collection server is the central means to defining the update intervals within one monitoring domain. The operator specifies the update interval for the domain on the command line when the server is started. The default update interval is 60 seconds. This interval is a good compromise between the need to minimize monitoring related load on the host computer and the network and the need to present current information to the user. Because most information is derived from system and application counters no information is lost. From the needs of long-term monitoring and data gathering longer intervals would also be suitable.

Whenever an agent registers itself at the server the server sends the defined update interval to the agent. The agent adapts its update speed to this interval. Therefore all agents reporting to a certain collection server update the load information at the same speed as the server replicates the data.

Check of time stamps and data validity

The collection server process has a passive role. All connections to the server are initiated from another process (agent or manager). It is irrelevant to the server which agents exist and whether or not they send data.

In order to guarantee data consistency time stamps are used to check that data are valid. The basic algorithm works as follows: whenever the server receives data

(registration or load data) a time stamp in the machine definition area is updated (it is set to the current system time of the machine where the server is running). If the server is asked for this data or (in the case of an intermediate managing server) data are to be retransmitted to another server the time stamp is checked against the actual system time. The difference between the system time and the time stamp stored in the machine definition may be 10% greater than the retransmission interval defined for the server (default is 60 seconds). If the difference is greater than that, data are assumed to be outdated and invalid. They are not forwarded to the requesting process.

Note: The manager does not see these timestamps. If it receives data it presumes that the information is current and valid. However it uses another time stamp to check whether the definition of each monitor object is unchanged or whether it has been recreated (see "Answers to requests by managers" and "Local Object Definition"). Do not confuse the two time stamps.

The usage of time stamps has proved to be very robust and efficient. It makes the communication infrastructure relatively invulnerable to failure of agents or collection servers. The manager does not receive old or invalid data. Due to the possibility that several agents report data about the same machine a theoretical problem has been introduced which has not been solved in the implementation described above.

As mentioned above, whenever the server receives information about a machine the time stamp in the machine definition area is updated. If two agents report data about the machine this happens two times during one retransmission interval. This is more or less correct and does not influence the functionality. If one agent fails to send data the time stamp still gets updated because of the second agent, and because the time stamp is updated, all data are assumed to be valid. Therefore the data of the failing agent are reported to the manager although they have not been updated for some time.

To solve this problem the time stamp has been moved into the source agent area and is updated for each reporting agent separately. For those checks which are performed for the machine (e.g. when the manager asks for data, or before removing the machine definition), the most recent time stamp is used. When data are prepared for transmission (see below) each individual time stamp is checked. If it is outdated the available data are replaced by the value *undefined* (0xFFFF). Thus the manager can not under any circumstances receive invalid data.

Consolidation of different data blocks

In order to keep the protocol simple it makes no difference whether an agent sends its data to a server or whether another server retransmits the information. As already mentioned, the server does not have to distinguish between the two cases. Only the fact that a process on machine A sends load data for machine B is important. Machine A and B may or may not be identical.

There are two messages in the client/server protocol which are used to handle the load data: one is used to send data from an agent (or collection server) to the server and the other to send the data from the server to the manager (on behalf of the manager). Both messages consist of a header and a data block which is attached to the message.

As depicted in Figure 15, the server maintains a separate data structure for each data source (the process which sends data). If the server receives data it only has to copy the data block, which is appended to the message, to the data area of the source agent structure.

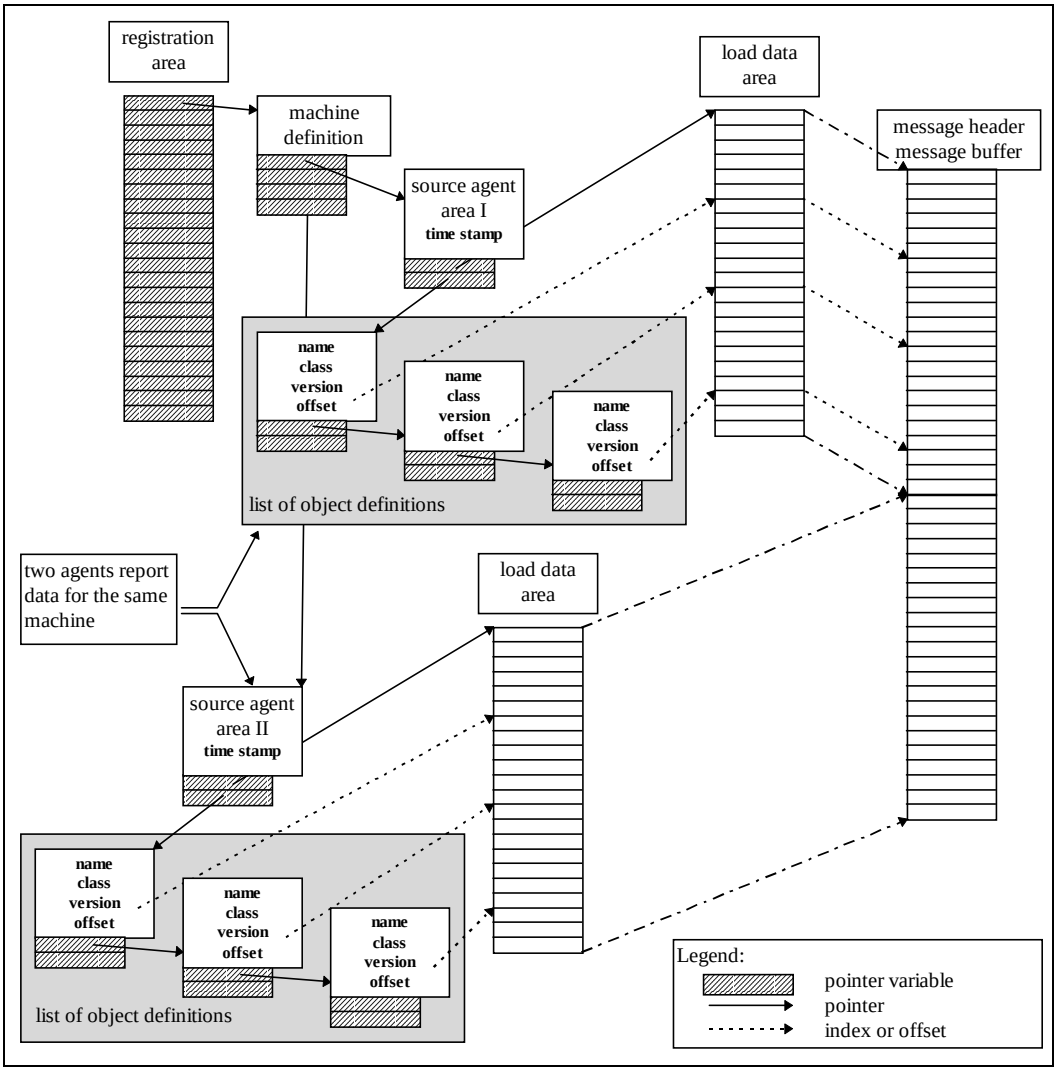


Figure 20. Network message assembly

When the manager asks for data or when the server has to retransmit the information the latter has to assemble a data block from the data areas of associated source agent structures. The assembled memory block can then be attached to the message. For other processes (server and manager), the server looks like one (simple) data source (like an agent). This keeps both the protocol and the client code rather simple but robust.

Each object definition contains an offset into the memory block which is sent after the registration has succeeded. If the server assembles a larger data block it has to modify these offsets accordingly. In the figure above this is indicated by the double dashed arrows (⇢), which point from the object offset into the new message buffer (identical to the data block of the source agent structure at the receiver of the message).

Answers to requests by managers

The manager accesses all information via a number of servers. Agents do not send their data directly to a manager. The manager makes a connection with all known servers and asks for data. The server answers with machine identification and an attached data block (assembled in the way described above). If the manager does not know the machine or if **creation time stamps** do not match, the manager asks for machine and object definitions.

The *creation time stamp* is set at the agent where the monitor objects are created. This time stamp is passed on by any server (collection or intermediate managing server) without any change. When the manager receives the resource definitions it stores the time stamp. Whenever it receives data from a server for the monitor object, the object's creation time stamp is sent along with the message, and the manager compares it with the time stamp of the local shell object. Thus the manager notices if data for a newer definition has been received. If this is the case the manager has to request the new resource definitions.

Although the *creation time stamp* is the time at the agent, for other processes on the same or different machine it is used as a key or id rather than a time stamp. The value is not compared with other system times and no time calculations are performed alongside it. Therefore it is safe to send this value over the network. Unsynchronized clocks on different machines have no effect on the function and meaning of the creation time stamp.

Retransmission of load information

If the application is used in a large network monitoring domains should be built. Within one domain an intermediate managing server collects the information from all agents in that domain, consolidates it and retransmits it from time to time to another server. There are several advantages to this method of data transport:

1. The intermediate managing server may use a different network protocol from the agents. While the agents use the most efficient communication for the local area network, retransmission may be done over a WAN link, which may require a different protocol. This protocol may not even be available to the agents.

2. A considerable amount of time and network traffic is devoted to establishing a connection. An intermediate managing server establishes one connection and transmits all machine data at once (in a series of messages; each message is already attached to an assembled data block). This prevents the need for numerous connections because agents can connect to a *local* server.

3. The update (retransmission) interval can differ for intermediate managing servers in different locations. It can be adjusted to meet network needs or special requirements of the location.

Service thread management

The collection server uses the power of OS/2 to implement high service availability while reducing resource consumption to a minimum. In order to be a thread must be ready to connect to the client at any time. If there is no available thread the client receives an error from the network software.

The collection server monitors the activities of its threads. It measures how many of them have been used at the same time during one update interval and how many have remained idle. If during one interval all threads are busy at the same time, a new thread will be started for the communication method.

One the other hand, if the maximum number of concurrently used threads, increased by two, is less than the number of available threads for about ten minutes, one thread closes down. Therefore two threads are ready in addition to the normal workload. They can handle peak loads if new agents or servers become on-line and begin data transmission.

5.4.2.3. Memory Management

"Registration of monitored machines and resources" gives an overview on the data structures allocated dynamically when a new machine or object definition is registered.

In detail the server maintains the following memory areas (see Figure 19):

1. A global, static **array of pointers to machine definitions**. This is the only memory area allocated in any case. The size of this array limits the number of machines that can be maintained by the server. In the current implementation up to 1024 machines could be referenced.

   Note that all other memory areas are allocated dynamically.

2. Each **machine definition** is allocated when a machine is registered. It contains an array of pointers to source agent structures. Up to four pointers are available. Currently only two pointers can be used during normal operation.

3. Each **source agent structure** has a pointer to a data block, where load data are stored, and a pointer to a block of object definitions. The sizes of these blocks depend on the actual available data. The blocks are resized if new information arrives.

4. A **data block** is an (dynamic) array of objects of the class DATAITEM.

5. A **block of object definitions** is a (dynamic) array of objects of class OBJECT_DESC.

6. An object of the class **DATAITEM** consists of an unsigned short which is encapsulated by several methods to ensure data consistency.

7. An object of class **OBJECT_DESC** consists of the name, the class, the version of a monitor object as it was created at the agent and the offset into the *data block* where the load data for the object can be found.

```
typedef struct OBJECT_DESC {
   char     Class[CLASS_NAME_SIZE];
   char     Name[OBJ_NAME_SIZE];
   int      Version;
   int      Index;
} *POBJECT_DESC;
```

Removing a machine

If the server has not received any data for a machine within a 24 hour period the machine definition and all associated memory areas are removed from the process.

5.4.3. The Workload Manager

The workload manager is the receiver side of the communication infrastructure where all information is collected and processed. The manager uses the services of layer four (the communication objects) to access collection (or intermediate managing) servers and to retrieve the data which are stored there.

One major difference in the way agents and servers carry out their communication is that a manager works with several (different) partners (servers). Agents and servers connect to one other machine to send data on their behalf. The manager

looks for available servers in the local network and contacts all of them during one update interval. Depending on the network, either the number of collection servers or the update interval has to be carefully chosen in such a way that the update processing will take less time than the update interval.

For each server the manager performs the following steps:

1. Ask for the next machine. During this "download" machines are referenced by a running counter (the *machine request index*) which starts from zero. The manager asks for machine *n* and the server sends the *n*-th machine definition of its internal array of machine references.

   The *machine request index* is not an attribute of the machine definition (neither on the collection server nor on the manager). The relation between the index and a machine is only valid during one operation. The index cannot be used to access the machine at a later time. Therefore the manager does not rely on this index during other operations.
   If no further machine definitions are available the server returns an "empty" reply and the processing for this server ends.
2. Look up the machine definition for the received data.
3. If the machine definition is not found, create one. At this point the manager has a reference to a machine definition.
4. Look for the source definition area (same scheme used for servers and managers). If it does not exist create one and set the creation time stamp to zero. At this point the manager has a reference to a source definition area.
5. Match the object creation time stamp in the source definition to the one passed along with the data message. If the time stamps match, continue with step 7.
6. If the time stamps do not match download the object definitions and create or update the internal data structures.
7. Copy the attached data block into a memory area associated with the source definition.
8. Update time stamps and increase the machine request index.
9. Continue with step 1.

The creation of a machine definition triggers a number of other tasks in the manager. These are described in chapter 6.2. "Internal Data Management".

5.5. Transmission Protocol

The following paragraphs describe in detail the definition and semantics of all messages and how they are used by the communication infrastructure components.

5.5.1. Message Structure

Firstly, we will look at the definition of the messages that are sent across the network. The messages are organized in a class hierarchy. The receiver of a message identifies the message by the message id. The first two bytes of the message header contain this id. The id consist of a number and a version. This is necessary to support different (newer) versions of infrastructure components during upgrading. Following messages are available:

```
#define MSG_TYPE_OBJ_DEF            0x1001
#define MSG_TYPE_DATA               0x0002
#define MSG_TYPE_ERROR              0x0003
#define MSG_TYPE_CLIENTDATA         0x0004
#define MSG_TYPE_REGISTER_OBJ       0x0005
#define MSG_TYPE_REGISTER_MACHINE   0x1006
#define MSG_TYPE_ACKNOWLEDGE        0x0007
#define MSG_TYPE_QUERYINFO          0x0008
#define MSG_TYPE_QUERYMANAGER       0x0009
#define MSG_TYPE_QUERYREMARK        0x100A
#define MSG_TYPE_QUERYUPDATEINT     0x000B
```

Not every type has a corresponding message class. Some types are used with the same structure.

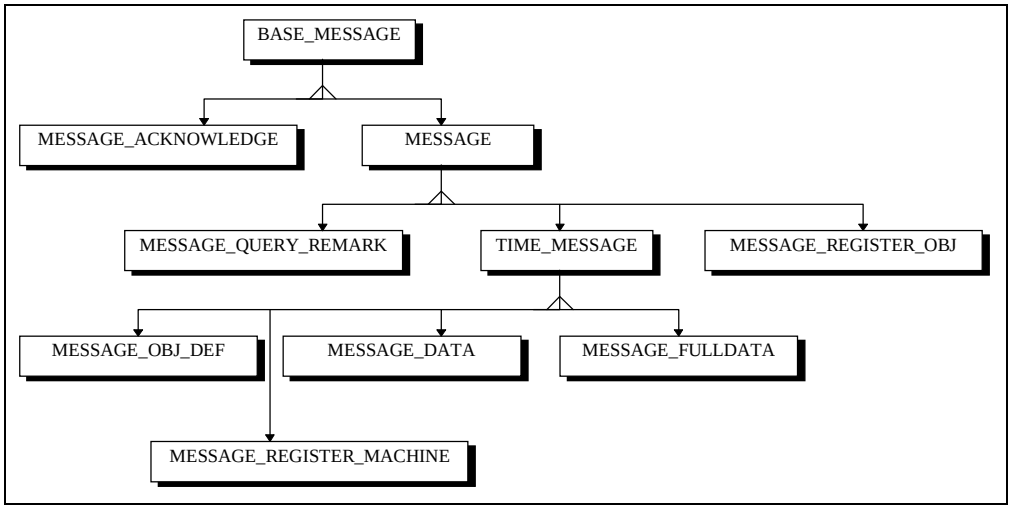


Figure 21. Message class hierarchy

Refer to chapter "Message Structure of the Transport Layer" in the appendices for more information about the messages.

5.5.2. Agent To Server

Figure 22 gives an overview of the messages exchanged during startup and normal operation of a client. After the agent has registered the machine (or the machines) and the objects monitored by the agent, only data are sent to the server.

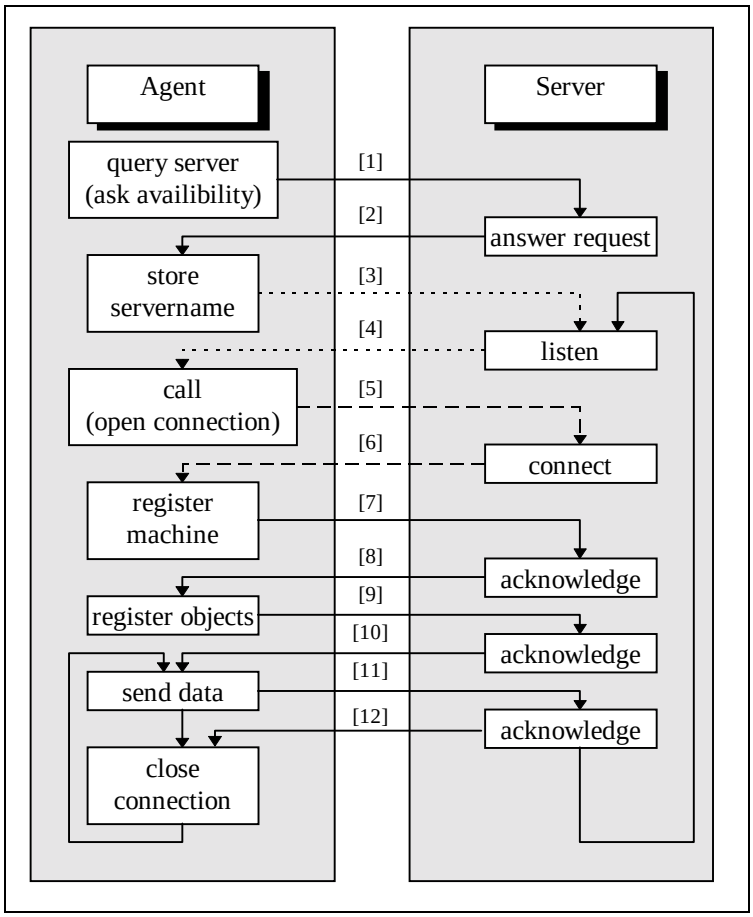


Figure 22. Overview of agent to server protocol

Explanation:

The agent has a list of potential servers. This list is either supplied by a NOS service, or the automatic lookup facility that is implemented by the tool, or it is supplied by the operator (in this case it contains only one entry, because the operator can only specify one server address).

After the first invocation or in the case of the last attempt to transmit information failing, the agent has to find an available server: it opens a connection to that server. If this fails (on the level of the network), then the server host is not available or the collection server is not active (or too busy). When a connection is able to be established, the agent asks whether the server is able to receive data. If this is acknowledged the agent marks the server as active collection server, then it registers all host machines at that server for which it will send information. For each host the monitor objects are registered that are associated with that host. The connection is then closed.

Every time the agent has to send information it opens a connection to the server and then sends one data block for each registered host. This is much faster and more efficient than sending one message per monitor object, as it is done in some systems management tools. After all data blocks have been sent the agent closes the connection.

The server has to acknowledge each message. This is the signal for the agent that the connection is up and that the server can still accept its data. After the connection has been closed by the agent the server listens and waits again for a new connection.

msg. #	Description
1	MSG_TYPE_QUERYMANAGER (BASE_MESSAGE)
2	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
3	implicit

msg. #	Description
4	implicit
5	handled by the network layer
6	handled by the network layer
7	MSG_TYPE_REGISTER_MACHINE (MESSAGE_REGISTER_MACHINE)
8	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
9	MSG_TYPE_REGISTER_OBJ (MESSAGE_REGISTER_OBJ)
10	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
11	MSG_TYPE_DATA (MESSAGE_FULLDATA)
12	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)

Whenever the agent fails to re-establish a connection to the server the whole sequence of messages is used again to find a new server and use it for data transmission.

If the operator supplies a fixed server name, server lookup does not take place and the agent will try to establish a connection to the given server again.

The method of the server lookup is not part of the protocol between agent and server. This is implemented by the communication object and is „unknown“ to the agent. After the agent has received a list of available servers it uses messages to *ask* the server whether it can receive data from an agent or not.

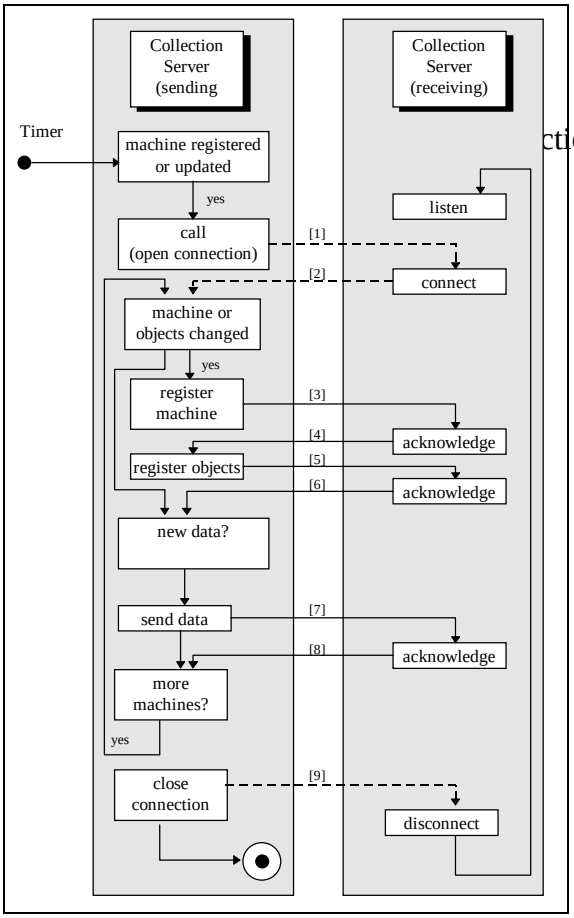


Figure 23. Overview of server to server protocol

ction server to Server

The server-to-server protocol has two tasks:

1. hand on machine registrations if a new registration has been received or an existing definition changed

2. hand on load data at regular intervals (the update interval)

Explanation:

Triggered by a timer, the collection server tests to see if some of the data that are maintained by the software have changed. If so a connection to the destination server is established.

If machine definitions or monitor objects have changed or new instances have been added since the last time the two servers communicated, the new or updated data definitions are sent (hosts and monitor objects are registered at the destination server).

Each host is examined for the availability of new data. If this is positive one data block, with all the information about the host, is transmitted to the destination server.

The destination server has to acknowledge each message.

After all data blocks have been sent, the connection is closed. The destination server starts to listen for a new connection. The collection server blocks until the next timer signal is triggered.

msg. #	Description
1	handled by the network layer
2	handled by the network layer
3	MSG_TYPE_REGISTER_MACHINE (MESSAGE_REGISTER_MACHINE)
4	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
5	MSG_TYPE_REGISTER_OBJ (MESSAGE_REGISTER_OBJ)
6	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
7	MSG_TYPE_DATA (MESSAGE_FULLDATA)
8	MSG_TYPE_ACKNOWLEDGE (MESSAGE_ACKNOWLEDGE)
9	handled by the network layer

Both tasks are time triggered. A separate thread, not used for servicing incoming requests, blocks on a timer semaphore and performs both tasks in sequential order.

5.5.4. Collection Server to Manager

The manager makes a connection with all known collection servers in the local LAN and requests machine load data. The figure below illustrates the protocol between manager and one server. This process is repeated for each server.

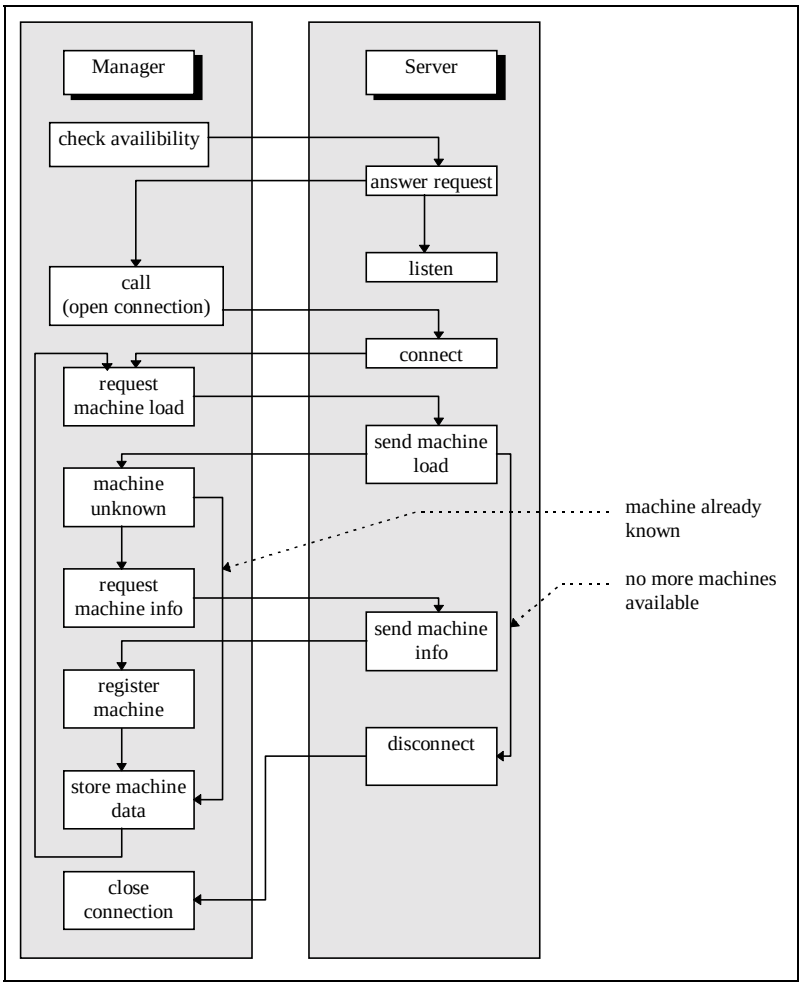


Figure 24. Overview of server to manager protocol

For each known collection server the manager checks the availability of the server. If the server is not available on the network, each attempt to connect to that server takes a lot of time due to time-outs in the network layer. Therefore the manager maintains a flag in order to remember which server is active. Every thirty minutes the manager tries to connect to an inactive server to see whether it has meanwhile become available. If the server is marked "available" the manager opens a connection and asks whether the server is allowed[30] to provide data. If the latter is not allowed to send data it is removed from the list of available servers and not further contacted during the lifetime of the manager process.

Triggered by a timer, the manager can open a connection to an available server and start to request load data about the host machines registered at that server. The server maintains an index variable for each connection that points to the next machine definition record to be sent to the manager. Each time the manager requests the next data block the server sends the data and increments the index. If no further machine is left the server sends an "empty" block. This signals to the manager all data blocks have now been received. In this case the manager closes the connection to that server and continues with contacting the next server.

On receiving a data block the manager examines its knowledge of the host and objects and the possibility of new or updated data being registered at the server. If this is positive the manager requests the machine definition and the monitor objects associated with that host.

After the manager has closed the connection the server disconnects and starts to listen for a new connection. The protocol and messages are the same as described before and are omitted at this point.

[30] The server software can be configured to not respond to a manager. This feature can be used to setup collection servers.

5.5.5. Hiding The Implementation

As mentioned above, the communication infrastructure should be transparent to the layers on top. During the work on the communication layer we learned something which became very important during later steps:

> **Finding:**
> The best and in terms of development effort most efficient way to hide the implementation and the existence of different release levels of distinct components on the same or different computers is through the **network protocol** itself.

Our original approach was to define an OO (C++) monitor class which had a generic interface for communication but no implementation. Each derived monitor class was responsible for accessing its data over the network. This evolved to the point that the monitor object is freed from the responsibility of communication. It retrieves data and hands them over to an agent. All monitor objects run in the context of an agent. The manager does not monitor anything directly. The collection server component is introduced to add more flexibility to the communication infrastructure. No direct connection between agent and manager is needed.

The protocol is "standardized". New components may choose other implementations as long as they adhere to the protocol. This makes changes to a component or to one of the class interfaces simple.

5.6. Low Level Infrastructure Components

To keep the task of communicating between agent and server simple the agent is shielded from this problem by a communication class which implements a named pipe. It consists of a server part and an agent part. Server and agent may establish connections and may exchange data packets of any size. This communication class is independent from the protocol described before; it could transport any kind of data between a server and a client process on the same or different machines in the network.

The principle interface of named pipes are used to define the interface for the communication classes. The techniques of named pipes is easy and straight forward but it became apparent that several implementation options has to be provided to fit the many different environments an agent may be used in. Following the idea of the named pipe, the communication between the components is always based on a **direct connection (session)** between two components. However, there is no session between agent and manager. A number of different sessions between different components are used to deliver the information to the manager. From a higher viewpoint this kind of communication may be compared to a simple message queuing system, but it is not intended to be one.

The following tasks must be implemented on the agent side to support efficient operation of the agent:

1. find possible server machines which run SRVMSRVR (the collection server)
2. establish a connection
3. send and receive data packets
4. close the connection

On the server side additional support must be given:

1. initiate communication
2. provide several concurrent threads for communication with different agents at the same time
3. clean up after a connection has been closed.

Three implementations are supported as part of this program: ***native named pipe***, ***TCP/IP*** and ***NETBIOS***. A developer may add new implementations that support other protocol stacks. The server must be able to support both communication channels at the same time. The agent must choose which one should be used and the selection provided by the operator via the command line.

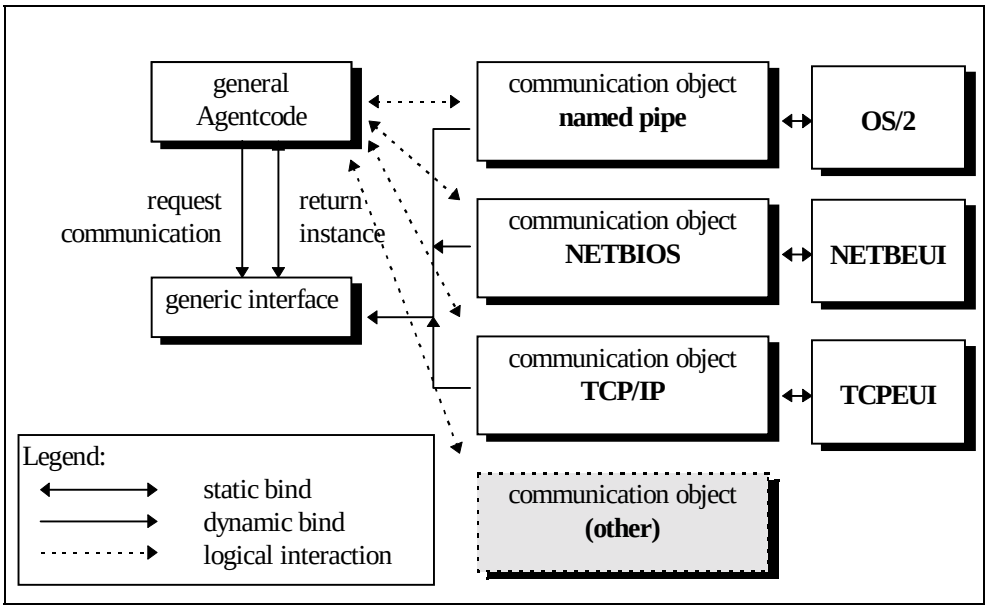


Figure 25. Interface to dynamic communication layer

The agent has, nevertheless, to be independent from the network layer it is working on. It must be functional when at least one communication method is supported by the underlying system. This follows that it must be functional if one method is **not** supported by the system. That means that no static bindings to protocol specific libraries are allowed for the kernel code of the communication layer.

E.g.: In the current situation a case may occur where NETBIOS is not available on a monitored computer. Named pipes are part of the operating system and are always available (but they must be implemented by the NOS installed on the system).

The communication objects are loaded dynamically at the request of the agent. If the code is available and can be initialized a communication object is created on behalf of the calling code. The initialization of the underlying code modules and the creation of the correct communication object are done by a small generic communication layer which has the knowledge about the available implementations (see figure below).

After a communication object has been created by this layer the agent works directly with the object. It invokes virtual methods of the communication object that implement the service. Most services of a communication object are rather simple. The implementation of the following tasks show that there are some tricky problems to solve.

5.6.1. Automatic Server Lookup

The most comprehensive service a communication object provides is the **automatic server lookup** method. It finds available servers in the (local) network. This method is important for an easy and unattended deployment of the agent. In addition, determining the destination server at runtime makes the communication more flexible and reliable. In the case of a server failing all agents who report to that server try to find another one and continue to send data if there is an active intermediate/collection server on the network.

The LAN Server product provides an API which returns the names of known LAN servers in the domains currently visible. This function needs a valid logon.

The NETBIOS implementation makes use of a *broadcast datagram*. An agent may send such a datagram with a certain id to all stations on the net that are listening to datagrams. After that the agents waits for connections, while specifying a certain time-out value. All monitor servers run a separate thread which listens to the network and receives such datagrams. If the datagram id matches the expected message id, the server establishes a connection with the calling agent. Following an established connection the agent reads the name of the partner from NETBIOS and closes down the connection, thus becoming ready for other connections.

If the agent receives a time-out signal, no other server has been listening on the net and the communication object returns the list of responding servers to the general agent code.

In certain situations - after the connection has been started or after it has been lost - the agent uses the service to find available servers and to (re)establish a connection with them. The initial purpose for this was to free an administrator from specifying target collection server names for every agent, which would have been terrible in a large network. An interesting side effect of that was that it led to an **automatic workload balancing** for the collection server processes. This is because if a server process is so busy that it cannot respond to further agents, each agent unable to request the service of that server tries to find another server (or comes back after a while).

In this case an agent has to (re)transmit all information about the resources it controls. In turn this may lead to a situation where several collection servers hold the information. Only one of them, however, is supplied with new data about these resources. As we saw above, the use of timestamps ensures the correct propagation of information.

5.6.2. Parallel Sessions On The Server

In order to be responsive to and to provide service for many agents on the network, a server must provide parallel sessions to process requests for service. OS/2 simplifies this task because of its multi-threading capabilities. Usually other system components on an OS/2 system support that, too, for example, all used network components (like named pipes or the NETBIOS interface) allow several concurrent threads to work with the network.

Nevertheless, the code of the communication objects and the layers above must be able to support multi-threading and parallel sessions. For the communication layer it is important to know whether it will be used on a server. It has to allocate and initialize network resources accordingly. Buffers, which are used in data transmission, destination and partner names and all variables in the code must be spread over various threads.

The processing in one thread must not have any influence over the processing of another thread. Each message is context free - it does not depend on an environment which has been set up by a message sent earlier. We have seen that there is a certain sequence of messages exchanged between different components, but the server will not be confused if intermittent messages come in due to other agents or servers.

6. The Workload Manager

The workload manager is the visible component in the application. It is situated at a central place and collects data which are available on the local machine and on (collection) servers in the LAN domain of the manager.

While agents collect data and hand them on without much processing and the server components stores data and handles the transport over the network, most of the data processing is done by the manager. The manager fully depends on collection servers to deliver data. It has no direct contact with any agent in the system.

6.1. Object Discovery

In order to be able to receive data the manager has to know which servers are available. There are three venues to examine:

1. the local machine;

   The manager starts a collection server process at its own machine (together with an agent which remotely collects data about LAN servers in the local domain; this functionality is used for historical reasons. It enables a novice user to start up the application and obtain a useful result. This simplicity was very important for the spreading of the application in the IBM community.).

   If the machine is configured to support NETBIOS or TCP/IP, or if the machine is a LAN server, this local collection server may also be the target for remote agents.

2. servers in the local domain;

   The manager uses a service from the NOS access layer (SRVMIMPL.DLL) to obtain a list of available servers. It keeps a list of known servers and tries to connect with a collection server process on each machine. Servers are checked for their availability before each communication attempt. This is necessary because of the long time out when that a server is down[31].

   If there is no collection server active on the LAN server the manager receives an immediate response and does not lose too much time.

3. servers from a definition file;

   The administrator may supply a list of collection servers which should be accessed by the manager. The manager reads the list and adds the machines to its list of servers mentioned above. The same rules apply for these machines.

Based on the protocol described in chapter 5, the manager accesses each server in turn. It asks for available data. If it detects information about a new server or sees that object definitions have changed (using time stamps) it downloads and updates its internal object definitions (see below).

[31] Depending on the configuration of the network software, a request for machine that does not exist can delay the calling process for several seconds before it returns an error code. When a server goes down, the monitor submits many requests (one for each resource provided by that server). All the delays can total minutes of waiting, and this breaks the whole monitoring system.

Thus, the "discovery" consists of knowing the accessible collection servers and requesting data from them.

6.2. Internal Data Management

The manager has to keep track of many different data structures and information. This chapter describes the most important areas and discusses the problems which had to be solved in the project work as well as solution concepts. For better understanding, some of the improvements, added over time, are mentioned along with these descriptions.

The chapter about monitors is of special importance to our work because the effective implementation of this class was one of the key aspects in the area of **feasibility of an OS/2 based solution**, which is one of the main questions of this work.

6.2.1. Local Object Definition

All processing of the manager is based on information retrieved by monitor objects. As mentioned above, a former version of the tool created and handled such objects within the scope of the manager itself. It became apparent that this approach was not practical and that the monitor objects could be better managed by an agent. Nevertheless the concept left an imprint on the later versions because the basic idea of the concept is still valid: it is the definition of an abstract object which provides data and shields the manager from the method of obtaining data.

With the introduction of remote objects, which exist in the scope of another process (maybe on a different machine), a subclass of the abstract monitor was created which was a shell for the real object and *simulated* the behavior of its associated monitor object. In some way it is an ***allomorphic*** object because it behaves like a monitor object but it is not. However both objects share the same abstract parent class.

To achieve better performance and simplify the task of network communication the transport of data and object definitions is handled by a general component which can be viewed as a part of the communication infrastructure. The shell objects do not directly handle the communication with their associated monitor object. Load

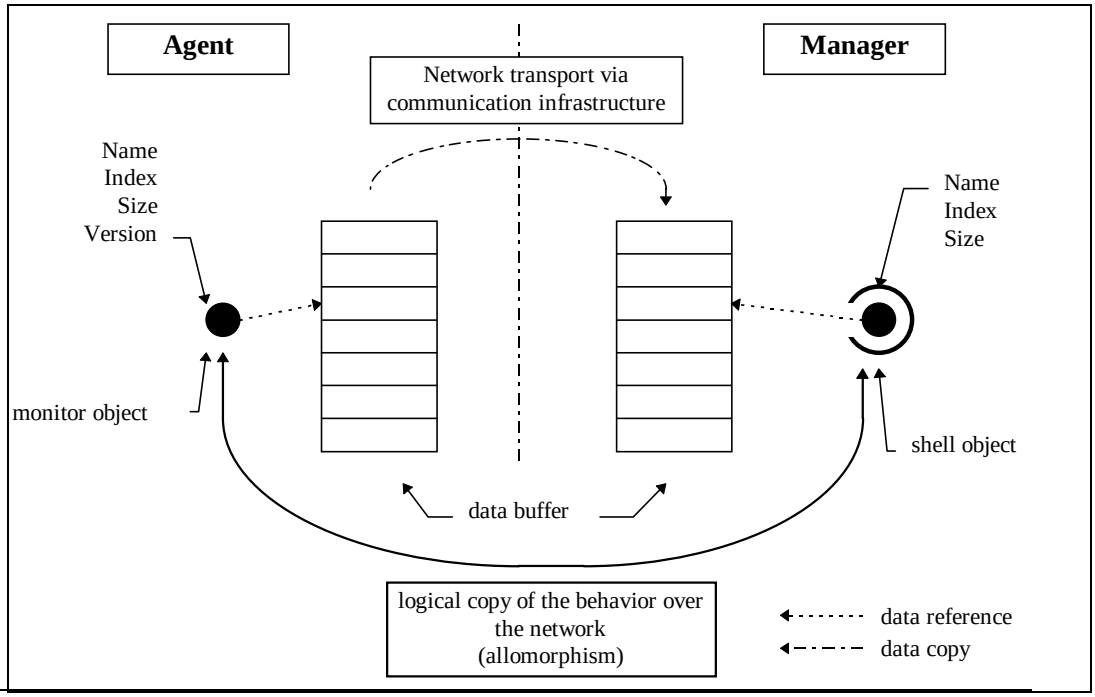


Figure 26: Shell object and monitor object

data are moved in large junks over the network and not for each individual object. In this way network overhead is minimized.

Figure 26 shows the (simplified) concept: The agent creates and handles monitor objects and data buffers. Objects have attributes and an index into the buffer where they put their data. The information is moved over the network (as discussed in chapter 5. "Communication Infrastructure").

At the manager a shell object is created from the information of the agent and it simulates the original monitor object. The communication infrastructure provides a copy of the data area from the agent. It accesses a buffer at the manager and for the next higher level of the manager there is no difference between a real monitor object and the shell object.

Ensure Object Validity

In order to ensure that the shell object "replicates" a valid object, the agent stores a timestamp in each machine definition. It is the time of the object creation (or last modification). All objects on a machine are created or updated at the same time. The timestamp is also transmitted with each data block The shell object compares the creation timestamp with the timestamp in the data buffer. If they match, the object definition is still valid. Otherwise it notifies the manager that the shell object for the referenced machine should be updated.

6.2.2. The DataStore

Each monitor object returns one set of data items at each update interval. The shell object behaves in the same way. In order to cover a period of time the data samples must be stored in a memory area. This area is called **datastore**. The data area and its associated references and methods is called a **monitor** because this objects controls monitoring activities and memory management for the workload data. In the first version of the manager datastore and monitor object was the same. It turned out to be better to distinguish logically between retrieval and storage, because they were (re)used in different contexts.

Each datastore has an associated monitor object. This object is responsible for feeding the datastore. The association is established when the monitor object is created. There are two possible monitor objects:

1. A shell object, as described above. Such objects are created whenever the manager detects and loads new data from a collection server.

2. A sum object, which is described in the next chapter. It is used to build sums of the current contents of a number of other monitors.

6.2.2.1. Properties of the DataStore

As shown in Figure 22 a datastore consists of a hierarchy of classes which are used to manage memory buffers in the size needed by the application.

class DataStore

This class forms the root of all storage related classes. It covers most of the required functionality and implements the virtual memory management algorithms. This includes backup and restore functions for the on-line data.

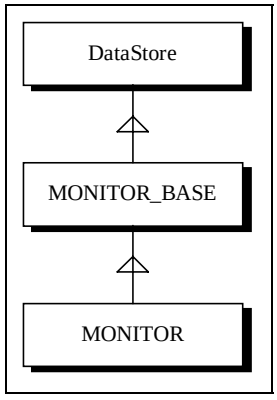


Figure 27. DataStore class hierarchy

```
typedef class DataStore {
   PDATAITEM pData;
   short     Start;
   short     SaveIndex

   static DATAITEM Placeholder;
protected:
   ULONG     flFlags;
   short     ValNum;
   short     TypeNum;

   void shift ();
public:
   DATAITEM   CurrMax;
   DATAITEM   CurrMin;

   DataStore (int t, int v);
   ~DataStore (void);
   inline DATAITEM &pValues (int Type, int ValIndex) const
          { return (pData ? *(pData + ((ValIndex + Start) % ValNum) * TypeNum +
Type) :
                            Placeholder) ; }
   inline DATAITEM &getCurrent (int Type) const
          { return pValues (Type, 0); }
   int getTypeNum (void) { return TypeNum; }
   int getValNum (void) { return ValNum; }
   void  Reset (void);
   int isValid (void) const { return pData != NULL; }
   int testExtrema (void)
       { return ((flFlags & DATA_EXTREMA) != 0); }
   void save (int fh);
   void load (int fh, int atValNum);
   int enforceLimit (DATAITEM NewLowerLimit,
                     DATAITEM NewUpperLimit);
   void enable (void);
   void disable (void);
   inline int isEnabled (void) const
          { return (pData != NULL); }
   inline ULONG getFlags (void) const { return flFlags; }
   inline void setFlags (ULONG flags, ULONG mask)
          { flFlags = (flFlags & ~mask) | (flags & mask); }
} *PDataStore;
```

The class **DataStore** handles a dynamically allocated, dynamically sized, two-dimensional array of *DATAITEMs*.

Property or Method	Description
pData	pointer to the array; the pointer may be NULL if the storage is *disabled*
Start	index into the array that marks the current value; if an outside object requests the array item with index zero, the object returns the item referenced by *Start*; if the array has to be shifted, only this index is decreased
SaveIndex	index of the last item that has been saved to disk; all items between this item and *Start* are not saved to disk and will be written when the next backup occurs
PlaceHolder	this static *DATAITEM* is used for operations involving the object when no array is allocated (*pData* is NULL); it is always set to *VALUE_UNDEFINED*
ValNum	first array dimension: number of "rows" in the array
TypeNum	second array dimension: number of types (= columns) in the array

Property or Method	Description
CurrMax	highest value in the array
CurrMin	lowest value in the array
shift ()	logical shift of the array; in fact, only *Start* is decreased and the items which are referenced by *Start* are reset to "undefined"
pValues ()	returns a reference to the item that is defined by two indices (or to *PlaceHolder*, if the object is disabled)
getCurrent ()	returns a reference to the item with the logical index zero and the physical index *Start*
save ()	saves the data to disk; after the initial backup only deltas are written to the backup file
load ()	loads the data from disk
enable ()	allocates and initializes the necessary memory for the array
disable ()	frees the memory and sets *pData* to NULL

class MONITOR_BASE

This class adds the functionality of writing log records to the workload database.

```
typedef class MONITOR_BASE: public DataStore {
protected:
   int openFile (int FileType) const;
public:
   char     Name[20];
   STRING   Key;
   time_t   LastUpdate,
            PrevUpdate;

   MONITOR_BASE (char *n, int t, int v,
                 MONITOR_SET &MonSet);
   virtual ~MONITOR_BASE (void);
           void  Update (void);
           void  ShiftData (void) { shift (); }
   virtual void  InitData (void) = 0;
   virtual void  NewValue (DATAITEM aVal[VALUE_ARRAY]) = 0;
           ULONG UpdateDelta (void);
           void  LoadData (void);
           void  SaveData (void);
   virtual void  WriteLogRecord (void) { }
   virtual char const * getServerName (void) const
           { return NULL; }
   virtual char const * getClassName  (void) const
           { return NULL; }
   virtual PCLASS_DEF getClass (void) const
           { return NULL; }
} *PMONITOR_BASE;
```

In addition to *DataStore* **MONITOR_BASE** introduces a *Name* and a *Key*. Therefore the class can be identified and is used in collections for fast lookup. The methods build a management layer on top of the services of *DataStore*.

class MONITOR

This class adds the link to a *monitor object*. When new data is requested the class can use the service of the retriever to obtain the information and put it into storage. Note that a retriever is either a shell object or a sum object, because a monitor always exists in the context of the manager.

```
typedef class MONITOR: public MONITOR_BASE {
   PRETRIEVER pObjRetriever;
public:
   MONITOR (char *n, int t, PRETRIEVER pObj, MONITOR_SET &MonSet);
           void  InitData (void)
           { LoadData (); pObjRetriever -> InitData (); }
           void  NewValue (DATAITEM aVal[VALUE_ARRAY])
           { pObjRetriever -> NewValue (aVal); }
           char *getName (void)
           { return pObjRetriever -> Name; }
   virtual PCLASS_DEF getClass (void) const
           { return pObjRetriever -> getClass(); }
   virtual char const * getClassName  (void) const
           { return pObjRetriever -> getClass() -> Name; }
   virtual char const * getServerName (void) const
           { return pObjRetriever -> pServer -> Name; }
   virtual void WriteLogRecord (void);
} *PMONITOR;
```

6.2.2.2. Virtual Memory Consumption

The amount of virtual memory required by the monitor is dynamic and depends on the number of data items provided by the monitor object and the number of samples. This number can become quiet large and therefore some sophisticated memory management techniques had to be developed. The following paragraphs give a basic estimate on how much memory is needed.

Most memory is required for storing workload data samples. Each sample is a 16-bit integer value and needs two bytes. DataStore objects come in two sizes: the small datastores, which are used for storing the *raw data*[32] from the monitor objects, contain 120 sample values times the number of items (attributes) associated with the monitor object. For a monitor factory object, which provides 10 items, this gives 1.200 items per monitor which means 2.400 bytes for one small monitor in this example.

The large datastores, which are used for views, contain 3076 sample values. For a monitor factory object which provides 10 items this creates 30.760 items, meaning 61.520 bytes for one large monitor.

Note that currently the number of items per class varies from one item to 122 items. The architectural limit is 128 items per class.

The number of monitors in the system depends on the number of monitored resources per server and on the number of defined views (see below). With the default views (which cover all available data items), and from our experience with our reference environment, we estimate about 70 large monitors and about 50 small monitors per server. Given the estimated number from above we get a memory consumption of 70 x 61.520 + 50 x 2.400 = 4.426.400 bytes which is about 4 MB of process memory for each server. In a small environment of 20 servers the manager has to deal with 80 MB worth of data. The sum of process memory of all process including the operating system and its device drivers cannot be more then

[32] The term refers to the sample values as they come in from the agent.

OS/2's upper limit of 512 MB. Given the numbers above and assuming a 10% overhead for other data structures we get a theoretical limit of about 105 servers that can be monitored from one manager. A user may significantly reduce the memory consumption by removing those views he does not need. This number will rise if a future version of OS/2 breaks the "512 MB" limit.

6.2.2.3. Monitor Memory Layout

It was one of the great challenges of the project to handle a large amount of data on a PC system. We assume that one of the main reasons why most systems management applications are only available on UNIX-like systems or mainframes is the ability of these platforms to handle large volumes of information.

Since OS/2's version 3 the virtual memory management has become sophisticated enough to support the algorithm described below. We assume that the reader has a basic knowledge about virtual memory. OS/2 V3 uses **paged** virtual memory. The size of one page is 4 kB.

An outside look at a monitor gives a two-dimensional array; one dimension is the *item index* and the second dimension is the *time index*. The item index is the reference to a certain data item within the items provided by a monitor object. The time index is a reference to the sample taken at a certain time. The index zero has the meaning of *now*. Thus, the time index for one sample is increased with each new data update.

The main requirement to make it function is for all operations (data update, calculations, display, etc.) to access data which fit into a few pages. Everything is omitted that touches the complete memory area. This would load the full monitor while other monitors would be swapped. During normal operations only the most recent samples are used. A sample consists of a number of items. Therefore the items of one sample must be placed on the same memory page and adjacent

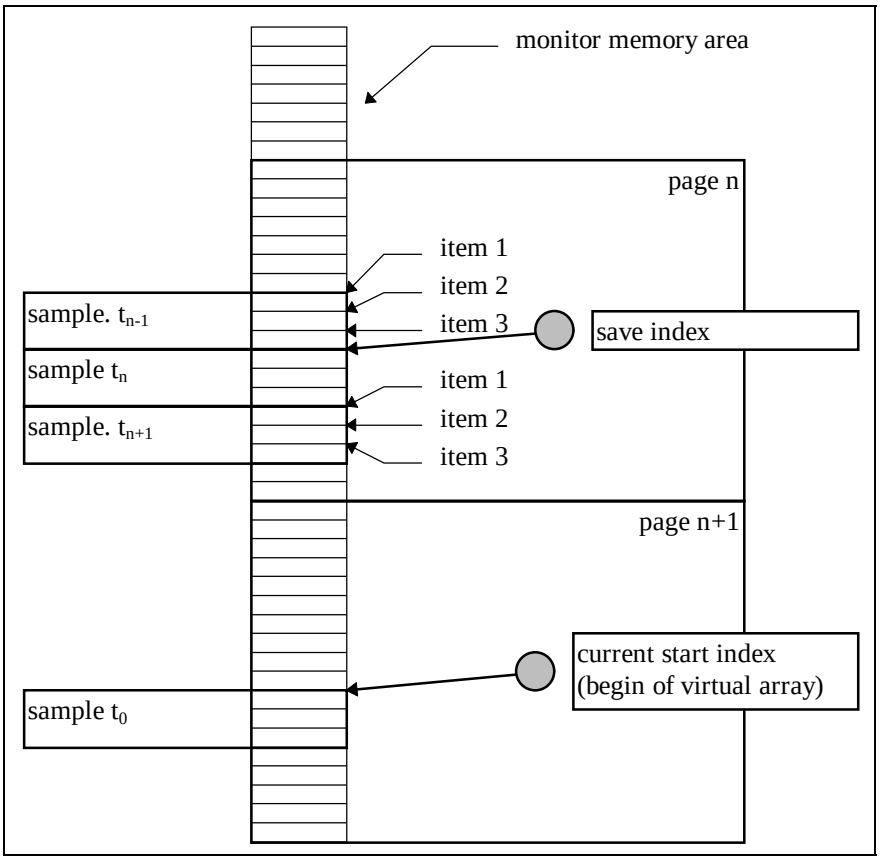


Figure 28. Memory Layout of a DataStore

samples should be next to one another. This brings the memory layout as shown in Figure 28.

It shows a datastore for a monitor factory object which provides three data items within each sample ⇒ sample size = 3. One sees two important properties of a monitor: the **start index** and the **save index** (see below).

The start index points to the most recent sample. This index is used to translate the time index, mentioned before, into the physical address in the array by adding the start index to the time index. If the result is an index greater than the size of the array, it is decreased by the array size. Before a new sample is requested from the associated datastore, the start index is decreased (by one) and new data are written to the sample.

Counting down just the index and working with the few referenced items saves the datastore from moving all its elements to make room at index zero. That operation would be expensive because it would force all memory pages of the datastore into physical memory. Doing that for all datastores during each update cycle would lead to endless swapping activities.

In the example above more than 680 samples fit into one memory page. If the manager requests one sample per minute it will spend more than 11 hours writing to that single page (11 * 60 * 3 * 2 B = 3960 B < 4096 B). The rest of the datastore can be swapped onto disk and will not be needed for a long period of time. During normal operation only one page per datastore is held in physical memory because no other parts of the memory area are touched.

Other operations may access more pages of the monitor, for example, if the user displays a view on the screen and scroll showing the data of the last two days it may be necessary to reload swapped parts of the monitor. This is handled by the operating system and will slightly degrade performance on the system.

6.2.3. Data Validation and Synchronization

When a new sample has to be requested the start index is decremented and the area that is now referenced by the index, is reset to the value "not a number". If a new sample has been received it is synchronized with the local update interval and the result is copied into the memory area. Otherwise later processing will know that a sample is missing. That way time synchronization and consistency over all datastores is guaranteed. The synchronization of update intervals is done for all values that represent deltas of a running counter between two samples. The meaning of such data depends heavily on the length of the interval. Because intervals in different monitoring domains and the manager may differ it is very important to synchronize them.

Let us consider the following example: the number of bytes written to a disk is measured. In the monitoring domain the update interval is two and at the manager the interval is one minutes. The manager will load new data every minute but the information at the server is updated only every two minutes. Because the information is still valid the manager will receive the same information two times. This is no problem for absolute values but it would lead to wrong results for a number like "bytes written to disk". Because the manager knows the update interval in the monitoring domain it multiplies the value with the factor (localInterval / domaininterval) which is 0.5 in our example. Thus, a value of 100 kB written to disk during two minutes would appear as 50 kB written in one minute in two consecutive samples.

6.2.3.1. Backup and Restore

A large monitor holds data containing more than two days worth of information. Because all information is kept in virtual memory of the manager process, the information is lost if the process ends. Naturally, an operator does not want to lose the data. On the other hand, the manager may be used on machines which are not turned on 24 hours a day. In many situations only office hours are of interest.

For this reason the application is able to store monitor data in the background. From time to time the application starts a secondary thread which saves all monitors on disk files, and from which they can be later reloaded. The time interval can be adjusted. The default is 20 minutes. For each monitor a file in the load database (see below) is created, in which the contents of the monitor are stored together with a file ID and a time stamp.

The ID is used to check the correct version of the file format, and the time stamp is needed to reposition the data in the monitor during reloading. Remember that the logical position of a sample in the data area represents a certain moment in time. Therefore it is very important to place the loaded information in the right index. When the backup information is reloaded into a newly created monitor, the *DataStore* object calculates the new position of items from the timestamp in the backup file, and the current time (in a new monitor, the physical array index zero corresponds to the current system time).

Two aspects had to be considered: the backup thread must not touch more pages than absolutely necessary, and the simple structure of the backup file had to be retained.

In order to fulfill the first aspect, the monitor had to remember which samples had already been written onto disk and which new samples had arrived since the last backup. For this purpose the save index was introduced. It pointed to the sample, where backup had started the previous time. That sample was the first which was written onto disk during the previous backup operation. All samples within the current sample (referenced by the start index) and the one referenced by the save index are new and must be saved for the next time.

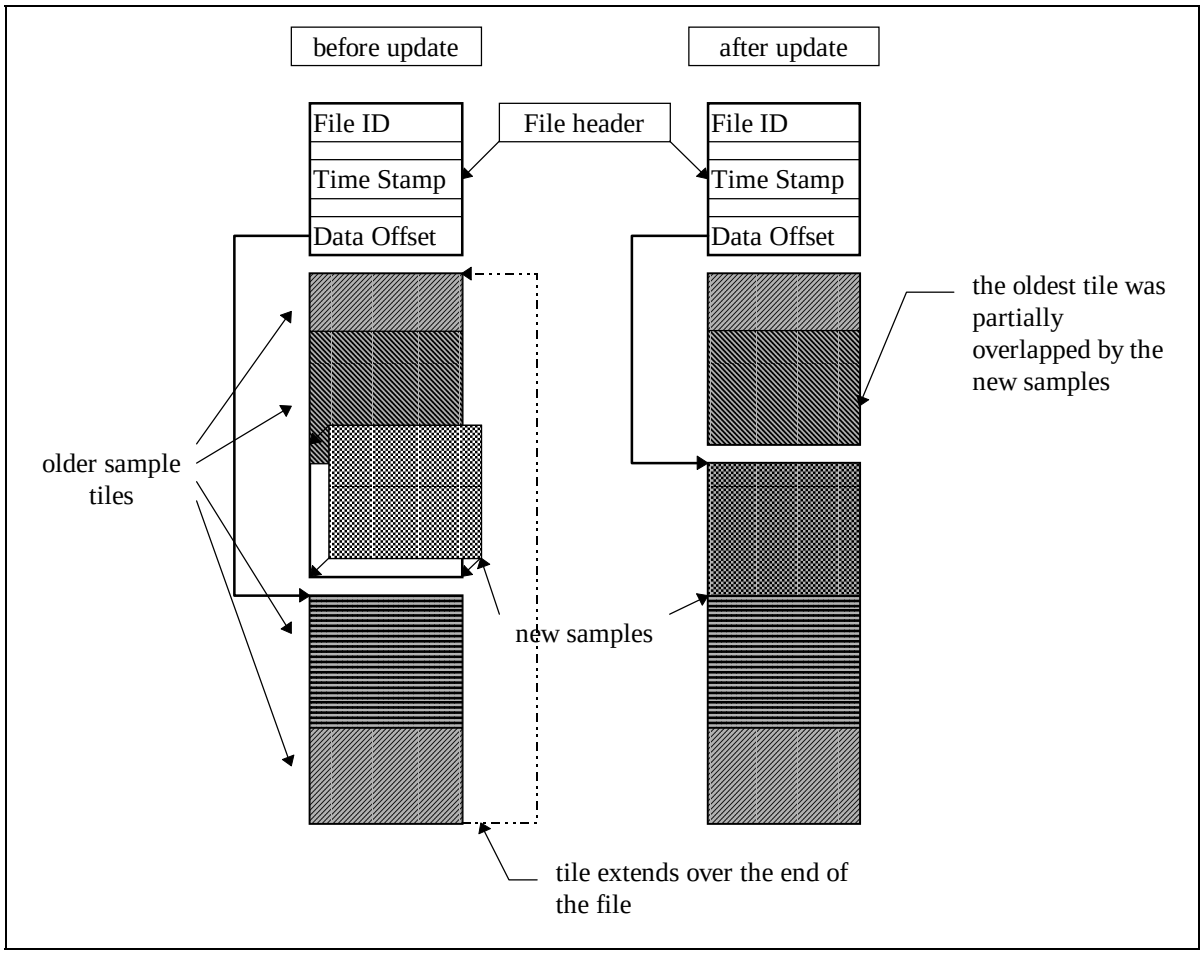


Figure 29. Structure of Monitor Backup File

In the backup file the new samples replace old samples in a way that the backup file appears as one continuous stream of samples. Figure 29 shows the structure created by the backup thread in a backup file. New load data that arrived since the last backup run replaces outdated information in the backup file.

The backup process writes new samples like tiles into the file. When the datastore is asked to backup its contents it calculates the number of samples since the last backup. Only these sample values are written to disk. Using the "data offset" that points to the beginning of the data in the backup file the datastore calculates the position for the new data and updates the data offset value.

It is likely that the size of the backup file, which is identical to the size of the monitor, is not a multiple of the size of a tile. Therefore the oldest tile is partially replaced by the newest tile. If placed at the end of the file, tiles may be split into two parts: one filling the remainder of the file and the rest is placed at the beginning of the data area (behind the file header).

Loading the backup data is very simple: two read operations are necessary. The first starts at the data offset and reads to the end of file. The second reads from the beginning of the data area up to the data offset - theoretically. In reality a monitor does not read the whole file, because the gap between the current time and the time of the last update of the backup file is left empty.

Example: the administrator stops the manager for some reason and lets it save the data. After twenty minutes he starts it again. We assume that the update interval is one minute. Therefore the first twenty samples in the monitor are missing and are put initially to *undefined*. Starting from the 21st sample (in memory) the monitor loads the data from the backup file (starting at the data offset noted in the header) and omits the last twenty samples from the file (there is no space to put them into).

6.2.4. Subclasses of Class Monitor

As we saw above, a monitor is a general concept to store load data over a period of time. The monitor application uses three specializations of this class:

1. the **raw data monitor** is used to store the data as they are loaded from a collection server;

2. the **view** is used to store data which are prepared for displaying to the user;

3. the **sum monitor** is a special raw data monitor and is used to build sums (totals) or averages (for normalized information) over other data.

6.2.4.1. Raw Data Monitor

The raw data monitor contains 120 sample[33] values per item. The associated monitor object is a shell object, as described above. This kind of monitor stores the load data as they originate from the agents. Each monitor object at any agent has a corresponding raw data monitor at the manager.

This class builds the basis of all other data processing at the manager. In most cases other objects access the data from the raw data monitor. **Data logging** is done only by the raw data monitor. Each raw data monitor has its own log file. Every hour a background thread is started which calls a log method for each raw data monitor. The monitor collects information about the number of samples received during the previous hour and appends a log record to the corresponding log file. For more information about data logging and the log-file format see chapter 6.6 "External Data Management".

6.2.4.2. View and Lens

A view contains 3076 sample values per data item. The associated monitor object uses a **lens** to "look" at a raw data monitor. The transformed data are stored in the view. Thus, the user has the option of preprocessing the data shown on the screen. Each view is coupled with a raw data monitor. A view cannot contain transformed data from several raw data monitors[34].

A lens is a simple object that processes a number of samples from a monitor and applies some mathematical function to them. Predefined lenses are:

[33] In the default configuration these are the data samples for two hours of monitoring. The logging mechanism needs the data from the last hour. The shortest possible update interval is 30 seconds. In this case the raw data monitor can hold one hour's worth of information. This is the reason why it has a size of 120 samples.
[34] However, a raw data monitor can be accessed (and displayed) from many views.

- raw data — data remain unchanged
- average — build an average over an hour
- minimum — find minimum of the last hour
- maximum — find maximum of the last hour
- hide — do not return the data; always returns the constant VALUE_UNDEFINED
- average per minute — average (see below)
- minimum per minute — find minimum of the last hour
- maximum per minute — find maximum of the last hour

The left part of Figure 25 depicts how a view access data of a raw data monitor through the use of a lens. Each value in the view passed the calculation of the lens object.

The difference between *average* (etc.) and *average per minute* is defined by the way the update interval at the manager is taken into account. The meaning of many values depends on the length of the update interval, because they are deltas between the value of a running counter at different times, for example, the number of bytes transferred over the network. For understanding this number it is necessary the know the time frame of the measurement. In the case of the default update interval of one minute there is no difference between the lenses.

Usually it is simple to think in terms of *units per minute* (or per second). Therefore lenses are provided which do the calculations and which take the actual update interval at the agent into account. Example: the current data transfer rate between a server and its clients is given as 10 kB. If one does not know the interval for which the number has been measured, the information is worthless. One would expect to see x KB/min. If the update interval has been set to two minutes, the number means 10 kB in two minutes (or 5 kB/min). Therefore, if one does not use the default update interval particularly if update intervals are different in the remote LANs, the "per minute" lenses have to be used for views. They perform the necessary calculations.

Views are defined dynamically by the user. **View definitions** are used to obtain the information about the construction of a view.

6.2.4.3. Sum Monitor

A sum monitor is a raw data monitor. It differs from the parent class in that

- the sum monitor is associated with a group of views (based on the same view definition) and
- its monitor object is a **sum monitor object**. It is used to build totals over the values of the views.

A special monitor and a special monitor object are vital because additional logic is needed to collect and manage groups of views. In addition, the building of totals is not simply summing up values. Depending on the meaning of the data items a different processing may be necessary:

- in the case of deltas where the sum retriever actually calculates sums
- in the case of absolute or normalized values where the sum retriever calculates the average of the values

Another problem is that a view contains transformed data (see "View and Lens" earlier). Therefore the sum monitor object must use extra information to access the *raw data* which are the basis for the view. They are used for all calculations.

Figure 30 shows a sum retriever object and a sum view and their relation to normal views and raw data monitor objects. Each sum monitor is associated with a (normal) view which holds values for display. The sum retriever access the raw data that are the source for the views and accumulates the data. Then the sum view uses the same set of lenses to generate the same effects for the display of the

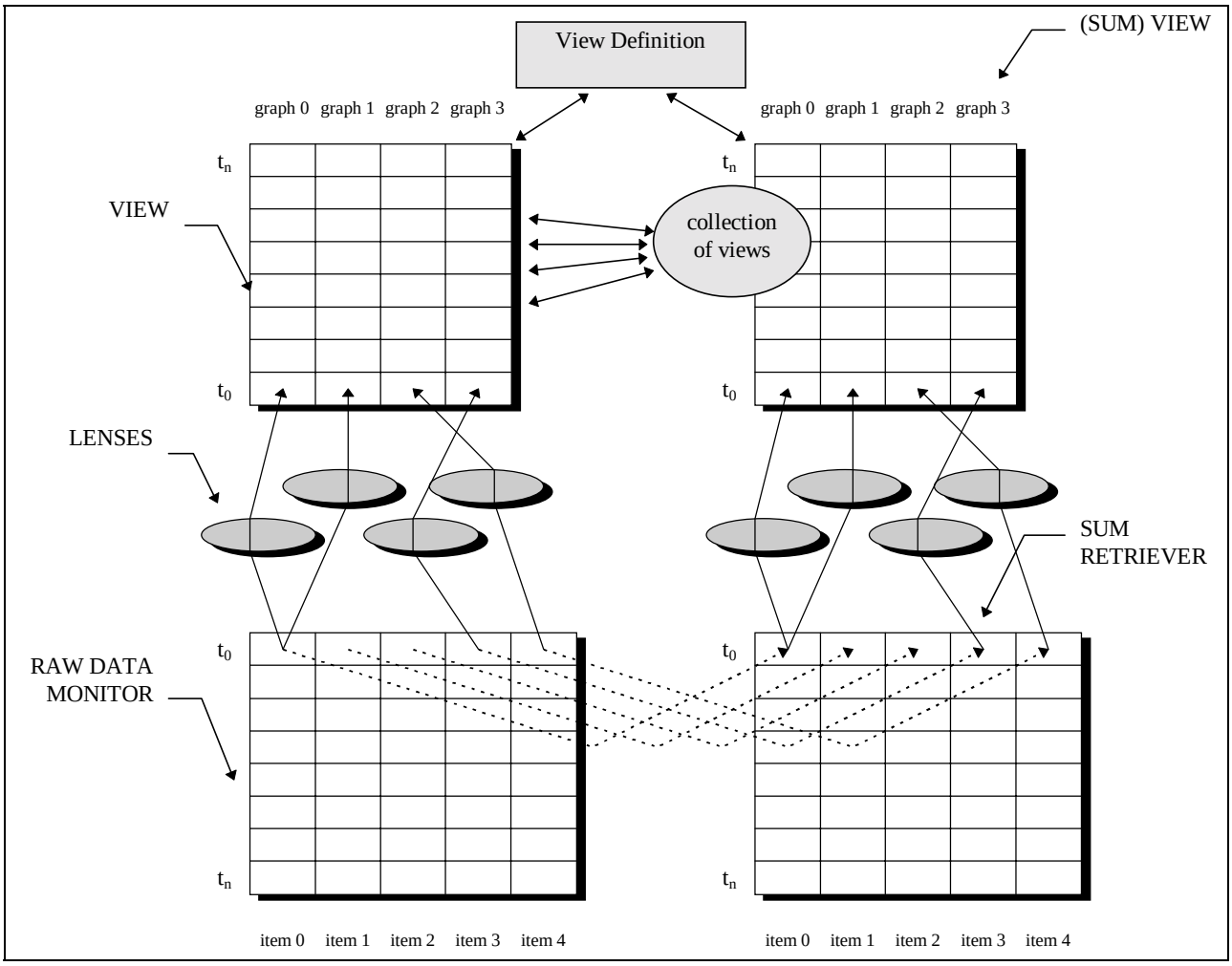


Figure 30. Raw Data, Lenses, Views, Sum Retriever

information.

6.2.5. Machines and Roles

In order to work with the great number of monitor objects (and their views) the access to the data must be managed for the user. For this purpose two dimensions are used to arrange monitors:

- the **machine** with which the monitor is associated (the machine for which it retrieves data)
- the **role** associated with the class of the monitor, for example "IBM LAN Server, "DB2/2 (Database) Server", etc.

Both entities are determined at runtime: the monitoring application does not know which machines and roles may appear during execution. It is able to accept new instances of the entities.

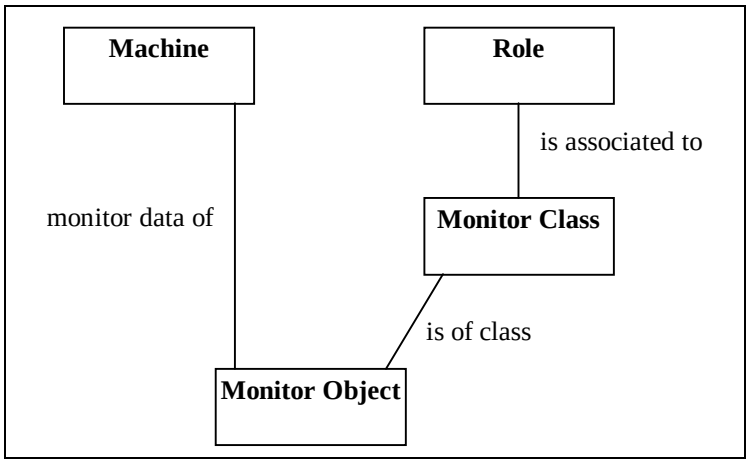


Figure 31. Relation between machine and role

Machines are defined by the network and identified by the agents. When a new monitor object arrives it is linked to its machine. If no record for the machine exists it is created. Machines are not *registered* at the manager (as an agent registers a machine at a collection server), but created as a "side effect" of monitor object creation.

Roles can be defined by monitor modules. They are important for the presentation of the data because views are grouped together according to roles. They are used to organize views and sums of monitor objects (as illustrated below). Examples of roles are "IBM LAN Server", "DB2/2 Server", "OS/2 PC", etc. Each physical machine can take on one or more roles. Each monitor factory object is associated with exactly one role. Later we will see how roles are used to handle monitor objects. Figure 31 shows the dependencies between monitor objects, monitor classes, machine and role. A monitor object is an instance of a monitor class that monitors a resource at a particular machine. The relation between monitor object and machine is established when the monitor object is created. The monitor class is statically associated with a role. This relation cannot change. Via its class a monitor object is also associated with a role (the role of its monitor class). Roles have their own icons in the graphical representation to distinguish them. The monitor selection window in appendix G contains an example for the use of different roles.

6.3. Multithreading and Background-processing

The monitoring front end application makes much use of OS/2's multithreading capabilities. On one hand this is necessary to provide a responsive user interface. On the other hand the program can make better use of computer resources by processing data in the background (invisible to the user) while the user works with interactive applications in the foreground.

The following table describes the threads and processes used by the manager:

Task	Type	Activation	Description
initialization, message queue processor, postprocessing	thread (ID 1)	permanent	The thread is used to initiate the application and handle all messages for the window system; this thread must not be blocked for more than one tenth of a second (see notes below); in addition, the thread uses PM timer functions for controlling background functions.

Task	Type	Activation	Description
background queue	thread	permanent	Regular background tasks like data update or saving of monitor data are done within this background thread; thread 1 pushes requests onto a queue handled by the thread.
data logging	thread	on demand every hour	The thread is created if and whenever log records are written into the monitor database.
database clean up	thread	once	Half an hour after the application has started it carries out database clean up. It checks the database directory for old or unreferenced entries and removes them together with associated files.
database migration	thread	once	If the application detects changes in the structure of log records or in the version of database files it initiates the thread which performs automatic migration of database records (see below).
local collection server	process	once	For communication with the local agent a collection server is started; remote agents may send their data to this instance of the collection server.
Local Agent	process	once	In order to stay consistent with earlier versions of the application, an instance of the agent is started which monitors LAN server information about server in the local domain remotely.

Note:

A. The thread with ID 1 is created when OS/2 starts a process. For OS/2 this thread has several special meanings:

- within its context DLLs are loaded and initialized
- the data segments of the executable are initialized
- signals are handled by thread 1
- when the thread terminates the process is terminated even if other threads are active
- within its context process clean-up is done

Usually this thread is used for the user interface (message handling), although this is just a common guideline and not an OS/2 limitation.

B. All threads execute with the default priority (*normal*).

6.4. Alerts and Threshold Values

Soon after an early version became operational, it was obvious that the application must be able to react to certain conditions automatically. This would then free the operator from constantly having to check load data. Therefore alerts were introduced.

In general, an alert is the definition of a condition that is constantly tested by the system. If the condition comes into being an action is triggered. The systems continues to test the condition. If the condition becomes 'false' again it is assumed that the alert has ended and a second action is triggered.

Note:
The introduction of two separate actions, one for the beginning of an alert and one for the ending, allows the operator to "put a certain condition into parentheses". This opens the door for event automation and the control of remote systems from the central monitoring site.

6.4.1. General

The following terms can be distinguished:

The **alert class** is the technical implementation of an alert. It is C++ code that handles all essential aspects of an alert.

An **alert object** is an instance of an alert class. The object is used to check for alert conditions. It has a reference to a monitor object which is checked by the alert object.

An **alert definition** refers to the data which are needed to create an instance of an alert object. E.g. it contains the reference to the data item and the threshold values that should be tested. The user can add, remove or edit alert definitions.

For each monitor object in the system a number of suitable alert objects are created. An alert object is *suitable* if it is defined for the same monitor factory object as the monitor object. If several alert definitions exist for this class several alert objects may be generated.

From the application's viewpoint alerts are objects which are derived from the class **ALERT_DEF**. New alert classes, which inherit from ALERT_DEF, can be defined and used in the system. At present, this is not supported in the user interface. Therefore only one alert class can be used by the enduser: **ALERT_DEF_EXEC**.

Appendix G presents some examples of the GUI components for specification of alerts.

6.4.2. Conditions

The alert class ALERT_DEF_EXEC can react to two types of conditions:

1. The current value of the referenced monitor object exceeds a threshold value.

   Two threshold values can be defined and the user may select whether the alert should be triggered depending on if the actual value is greater than the upper limit and/or lower than the lower limit. It is possible to specify just an upper limit or lower limit or both. In addition condition testing can be limited to certain machines and certain resources.

   Many alerts can be defined for the same attribute. The user can select different conditions in each alert. This offers great flexibility in controlling the whole alerting mechanism.

2. The last five values of the monitor object were undefined; thus it is assumed that the resource is no longer available or the connection to the monitor object at the agent is lost.

Two separate operations can be defined which are triggered when the condition is recognized and respectively when the condition ends.

6.4.3. Reaction Operation

The *operation* is a text string and the alert class is able to perform a combination of the following actions with the text:

1. Display the text as a message window and wait for a user response.

2. Write the text into the log-file - together with a time stamp for later analysis.

3. Execute the text as a command at operating system level. This is the most important action. The most obvious usage here is the notification of an administrator via a FFST[35], SNMP or NetView alert, an e-mail or a pager.

Actions are processed in a separate thread (thus they will not interrupt other processing) but they are handled sequentially. Until one action is finished, no other action can be performed. Triggered actions must wait in a queue until they can be processed.

6.4.4. Alert Variables

In order to support event automation and to create informative messages and event logs, the application provides variables that can be used in the action text. Adhering to OS/2 conventions, variable names are written within a pair of percent signs (%). The following variables are available for action text assembly:

Name	Meaning
%SERVER%	the name of the server for which the action is triggered
%RESOURCE%	the name of the resource for which the action is triggered; for monitor objects of global classes, this name is identical to the server name
%VALUE%	the current value which triggered the action
%ALERTNAME%	the name of the alert definition
%ALERTTYPE%	the type of the alert; this is always "EXEC"
%CLASS%	the name of the alert definition class(= the class of the monitor object)

At the time the action is triggered variable names are replaced with the actual values, then the defined actions are performed.

6.5. Dealing with Monitor Modules

In the same way that the agent dynamically loads monitor modules with the code to retrieve workload data, the manager loads modules with class definitions. Both kinds of modules come in the form of DLLs and are generated from the same source code. The definitions for the manager provide information about the nature of the data and the way they should be handled. Figure 27 presents the major elements and their relation. Appendix D contains the definition and explanation of important classes and class properties that are used in this context. The attribute definition (FIELD_DEF) and monitor class definition (CLASS_DEF) are essential to interpret the workload data received from the agents.

[35] First Failure Support Technology

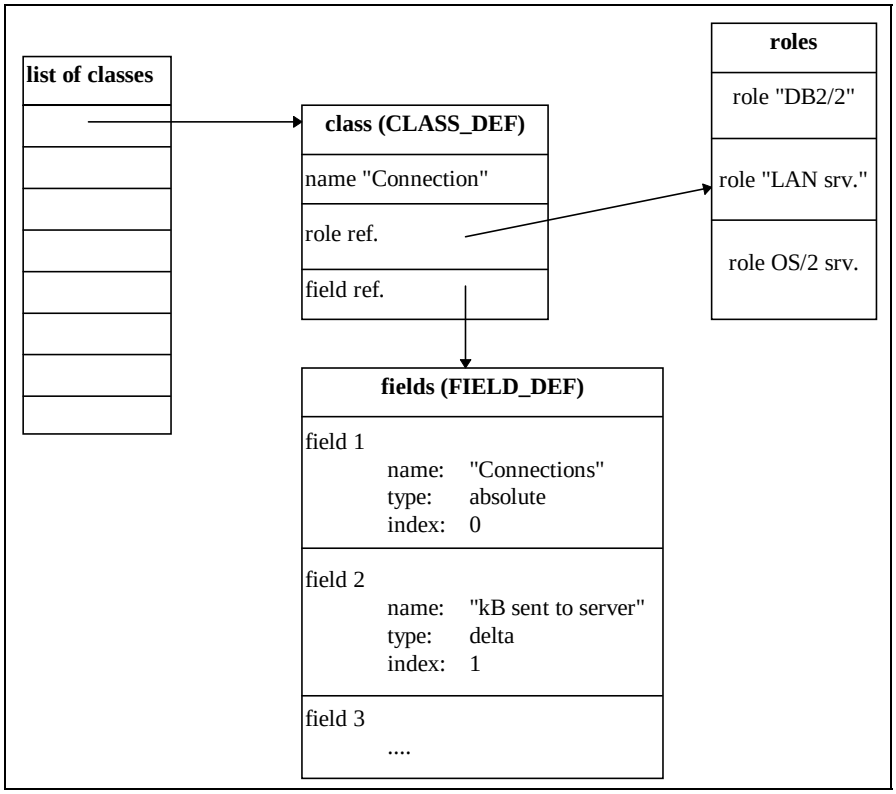


Figure 32. Data definition of monitor modules at the manager

The structure of the module is similar to the monitor module because it is intended that both types of loadable modules be generated from the same source code. The important differences are:

- The module may provide role definitions. A role definition links a name, a key and two icons together (these items form the "role"). One icon represents global server attributes (like "available process memory") and the other is used to represent individual server instances where specific resources can be provided to the user (like information about "logical disks"). Given that server and resources with that role are registered at the manager for each role one global icon and zero or more server instance icons are displayed. The user can open a menu at each icon to open a window with specific workload data.

- The module provides classes like a monitor module but the class lacks the retriever code. Thus, it does not have to be linked to any system or server application specific code that may not be available on the manager.

- For each class **default view definitions** must be provided. When the class is loaded for the first time these definitions are added to the existing view definitions.

At the manager these modules and objects are mainly used to control the graphical presentation and the numerical processing of the workload information.

6.6. External Data Management

Besides the graphical representation of the incoming data, the other main task of the manager application is the processing and storing of the data for later use. For this purpose a (kind of a) database is created and maintained which is optimized for the

stream of time-stamped load data. This chapter describes the data that are written onto disk and their preprocessing before being permanently stored.

The database is transparent to the user. The different parts of the application access and maintain the same data while the user does not have to bother about the placement and the structure of the information. For advanced users there is a way to define/change the placement of the data: to possibly move them to a shared disk drive in order to let several users access the data.

6.6.1. The Load Database

The database is a very simple means of storing several kinds of information about monitor objects. Basically it consists of an index and a large number of sequential data files (*data streams*). For each index entry a number of related files may exist. In the current implementation two types of files may be generated: a (log) data stream, which contains a sequence of log records, and monitor backup files, which were discussed in chapter 6.2.3.1 "Backup and Restore".

This structure is selected because it is very suitable for the requirements of the tool. In general there are two types of operations: the manager adds records to the database. The information is sequential by their nature because all records are ordered by their timestamp. The on-line part appends records at the end of the logical sequence of existing records. Sequential keys, especially timestamps, can be a problem for relational database as they target at random access to records. Appending a record at the end of a sequential file can be done much faster and more efficient than with a relational database. The second major access type is sequential read of all records of one data stream during analysis. Again it is much faster to read through all records of a sequential file than to select records in an index, build a result set and read one records after the other from a database.

There is no need for random access to a specific record in a data stream. Therefore the cost for a sophisticated database system can easily be omitted.

Another important consideration that is reflected in the structure of the database: the compression of the information to be able to store long-term information for many servers with a number of different applications on PC disks. To better understand the requirement let us consider the attempt to log similar long-term information with a tool like the NT Performance Monitor. We use this monitoring tool because there is no comparable tool for OS/2 and NT is more similar to OS/2 than UNIX. With logging of all attributes enabled on a single machine one sample is about 35 kB in size. This number may vary as it depends on the number of actual resources like logical disks or network adapters at the monitored machine. With a sample interval of one minute and a logging period of 365 days we get the total logfile size for one machine of

35 kB * 60 * 24 * 365 = 18.396.000 kB or **18,4 GB**

This number includes only the information about the operating system of one machine. Just that would be too much for PCs or workstations. Assuming the requirement to monitor 50 servers with two server applications[36] each and assuming that applications add only 40% of the amount of data compared to the operating system we would come to a figure of about 1.655.640.000 kB or **1.655,6 GB**. Neither would somebody be willing to spend that much space for monitoring nor would it be possible to perform analyzes. That example illustrates the need to transform the data somehow and to create algorithms to handle long-term data.

[36] This is theoretical because this tool is not able to include other workload data.

6.6.2. Database Index

The index is a collection of records which establish a relation between monitors and a unique identification number. The monitors are referenced by their key. The name of their server, the resource name and the class key are stored as text in each record.

When the application has been started, the index is completely loaded into memory. All operations are done within the in-memory collection. When an application part requests an entry for a monitor not yet registered, a new entry is created automatically and the collection is marked as changed. At certain points in time the collection is written back onto disk. Index entries are requested (indirectly[37]) by several parts of the application whenever it tries to access stored information for a monitor object.

When a new operation begins, which may cause changes to the collection, the application checks whether the disk image of the index has changed. If not, the in-memory image can be used. Otherwise, the existing collection is released and the index is re-read from disk. This enables several processes to work with the same database[38].

The above steps occur whenever index records are "deleted". In fact, such records are marked as deleted. When the index collection is written onto disk these records are omitted. Thus they disappear from the index. The *database clean-up routine* is able to detect and remove unused index records (see below).

6.6.3. Log-records

Every hour the application generates log entries in the database. Each record represents one hour of detailed load data. Due to the size and the volumes of detailed load data it is not possible to store all the information for a longer time period. The data has to be "compacted" in order to be able to provide material for long-term analysis.

When the application writes log records onto disk the following format is used for each record:

Timestamp	**4-byte timestamp** **# of available items**
Datagroup 1	**sum of available items** **minimum** **maximum** **standard deviation**
....	**....**
Datagroup n	**sum of available items** **minimum** **maximum** **standard deviation**

For each data item (attribute) of a monitor factory object a *data*-group is put into the record. Within one sequential datafile (a *data stream*) all records are of the same size while the size of records vary between different streams because the number of attributes can be different among different monitor classes. New records are appended at the end of the *data stream*. A data stream grows continually. The

[37] The manipulation of the index is transparent to the rest of the application.
[38] Of course this is an optimistic approach works all the time. If many different processes change the index at the same time some index entries may be lost.

user has to use the database compacting function of the data analysis tool to remove out-dated records.

If no (or too few) data samples have been collected during the last hour, no log record is written for the corresponding monitor. Later the analyzer detects missing records and uses that information to calculate resource and server availibility.

6.6.4. Record Description

Because of the structure of the log record, with its data groups, the size of the record depends on the class definition (the monitor factory object defines the number of data-items provided by the class). These definitions may change over time (as the developer adds new or removes out-dated items). Therefore the size of the records - as they are used in the data streams - has to be recorded and is used when the data are accessed. An extra file contains a table holding the number of data items for each class.

6.6.5. Automatic Migration

While the index is loaded the database access layer checks the following things and automatically corrects/adapts all files:

6.6.5.1. Check Index File ID

The first bytes of the index file resemble a database ID. The ID is used to check versions and structure of the current database. If the ID does not match the expected ID, which is defined by the database code, the ID is used to lookup an *auto migration function* in a table and this function is executed in a background thread. A progress indicator is displayed while the function is active.

Whenever the structure of one of the files is changed a migration function has to be supplied. A function has to transform the data from the last release to its current release. Over the course of migration several functions may be used in turn.

Example:
Assume the following table of auto migration functions:

Function	**migrates from**	**migrates to**
FUNC A	Rel 1.0	Rel 2.0
FUNC B	Rel 2.0	Rel 2.2
FUNC C	Rel 2.2	Rel 3.1
FUNC D	Rel 3.1	Rel 4.0

Note that not every release may change the format of the data. Therefore some releases may not be covered by the table.

If the application comes across data from Rel 2.0 it will first call FUNC B. To the database code it is transparent what the function does. It does not know which version id the database migrates to. After the function is completed the database manager restarts its processing. Now it detects that the database has the format of release 2.2. and will call function FUNC C. The data still do not have the correct format and therefore it will call FUNC D. This function changes the data to the latest format and the application can continue to load the index.

The big advantage of this method is that it is much easier to upgrade the database to a new format. The migration function only has to reflect the changes of one release. By adding it to the table the rest of the migration processing is done automatically.

6.6.5.2. Check Record Description

As mentioned above, the size of a log record depends on the number of data items provided by the monitor factory object. One major requirement is an **open interface** for third parties. The idea is that the provider of a server application provides a suitable monitor module together with the server code. Because of the long-term nature of the application the tool must be able to handle future changes in the application. Changes in the monitored application most likely will be reflected in the number of attributes that can be monitored. Consequently, when a monitor module is loaded the tool checks the number of attributes and compares this with the record definition in the database.

If the number of data items is changed by the provider of the class, the database has to be altered to reflect this change. If the monitor application finds one or more classes with different descriptions, all log files (data streams) are processed and the size of each log record is adjusted to the new size. If the new number of attributes is greater than the current number, new data groups are created with undefined values ("not a number").

It is not supported to remove attributes because that would destroy valuable information. New code may set such attributes to "not a number" or may reuse it for a similar attribute. To be save the case that the new size is smaller than the current size is also handled: the last *n* data groups of each record are removed so that all records fit to the new size. This rule is not fully accurate because the database does not know which data items have been removed from the class.

As a rule of thumb a developer should not remove a data item from the class definition because this may corrupt the meaning of all old log records. In most cases it is better to keep the definition and just drop the code querying and filling the data item.

6.6.6. Database Index Cleanup Routine

Over time many index entries are created, and to avoid unrestricted growing of the index and the whole database, an automatic clean-up routine is necessary. It detects unused index entries and out-dated data files in the database and removes them.

An unused index entry is an entry which does not have reference to existing files in the database. An out-dated data file is not touched for a certain amount of time. The allowed time period is 365 days for data streams (log records), and three days for monitor backup files. After this time it is very likely that the corresponding system resource no longer exists and therefore no new data are received by the monitor application. For some time the available log data may be used for analysis but after one year[39] the data are useless and will be removed.

The cleanup routine is begun in the background at regular intervals and runs as a separate process, invisible to other activities on the machine. This is totally invisible to the user of the application. The index cleanup is only scheduled by the manager. This function does not perform data cleanup as described in chapter 7.

[39] The current study brought evidence that one year is already too long. The data become outdated very quickly.

Making Use Of The Workload Monitor Information

7. Working with the Log Database

Chapter 6.6 discussed how the manager generates log records. This chapter describes the structure of the workload database and the algorithms and programs that were developed to get information out of the data. Special focus was on the problem of data consistency and the volume of data that accumulated over time.

7.1. Data Structure and Access

An important task of the manager component is to maintain a load database and add log records to it. The structure of the database is very simple:

- An index file contains the keys and references for all known resource proxies. For each resource that was detected by an agent and for which the database still contains information one index entry exists. The index is used to navigate through the database and to find log and backup data for a resource's datastore object.
- The structure of each monitoring class is described. This information is used to upgrade the database when the structure of classes changes.
- For each resource proxy, a number of data files can exist. In the current implementation, up to two files may exist: a sequential file with the log information and a (structured) file[40] with a backup of the on-line data.

The manager will only write a log record when at least half of the possible sample values are valid. Valid records contain data values other than "not a number". Because the sample values stored in a datastore object are reset in the sampling interval independent from actual arrival of new data the monitor makes sure that records are in a well defined state (see "Data Validation and Synchronization").

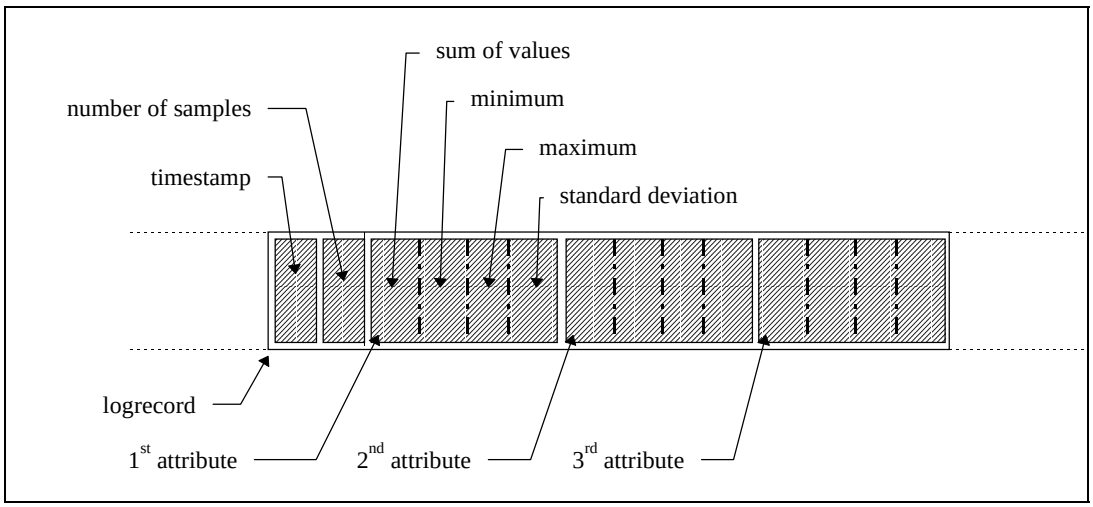


Figure 33. Structure of a log record

[40] The information in this file is used to restore the contents of the views in cases where the manager terminates unexpectedly (see "Backup and Recovery").

Resource proxies that have not receive new data for over half an hour will not generate log records.

The structure of the logfile is also very simple (see chapter 6.6). It is a sequential set of log records. Figure 33 describes the structure of a log record. Each record consists of the timestamp of creation, the number of sample values that have contributed to the log and a set of log record items (one for each resource attribute of the associated resource proxy). The log record item contains

- the sum of all sample values; the average can be calculated when needed
- the lowest sample value (minimum)
- the largest sample value (maximum)
- the standard deviation of the sample values

A database layer handles all requests to add or read database entries. This layer implements some additional functions:

- **detection and omission of invalid/destroyed log records**

  Due to different problems in the course of data retrieval, transportation, log processing and database upgrades/manipulations it is possible for a log record to be destroyed or the data of the record become invalid. The log record is eliminated in cases where the number of contributing samples is invalid or where the time stamp does not fit into the sequential flow of the other records (e.g. is older than the last record before). The "number of users" stream is used as a reference to check the number of expected samples. In general the number of samples cannot be higher than 120 because 30 seconds is the lowest supported sample interval resulting in a maximum of 120 samples per hour. Records representing more than 120 samples are invalid. In addition a record in a data stream is compared to a record from the "number of users" stream from the same sample interval. It must have the sample number of samples (or less if there has been an outage at the monitored resource).
  Using overall averages and thresholds, "out-of-bound" records are also removed. This is essential in enabling the calculation of meaningful statistical reports later on.

- **omission of out-dated log records**

  Because of the dynamics in host configurations and limitations in the amount of data that can be handled all records older than one year are ignored. Practice showed that older records are seldom of interest (because important parameters change meanwhile).

The database layer hides this processing. During read operations invalid records are omitted but not removed from the database. A separate maintenance process can be scheduled that takes advantage of the functionality of the database layer (omission of invalid records) and writes back the corrected data. If a datafile is empty it is removed from disk and the index file.

Whenever the database is opened by a process the database code first checks the version of the database and then migrates the database automatically. For more details on automatic database migration see chapter 6.6.5 "Automatic Migration".

The low level processing of each data stream is already done in memory. When a data stream is opened for reading the database code reads the complete file and creates the structure for the file in memory. It replaces missing records with special records containing "not a number" values. This structure makes it possible to access

the data of each hour of the year directly by a simple index calculation. During the read from files some checks like timestamp matches are performed. Based on the memory structure out-of-bound calculations are done. This results in a "clean" memory structure which then represents the data stream. All following data access operations are done in memory and are very fast. This contributes to the high performance of all processing functions. The **data cleanup function** writes back the "cleaned" file to disk. That way old or destroyed records are removed from the file.

7.2. Statistical Processing

The workload monitoring tool offers a user-friendly way to work with the workload database. This chapter briefly covers the elements and windows of the tool. A navigation tool is provided to make use of the objects in the database. Standard reports can be generated. They are presented in a graphical way with a number of customization options. The following pages briefly describe this tool and what can be performed by an end-user.

With the analysis tool it is possible to work with parts or the whole of the log database. After commencement it presents an "object tree" of available information and a number of available tools in a tool bar at the right side of the window to the user (see Figure 34). The user applies a tool by taking an object from the tree and

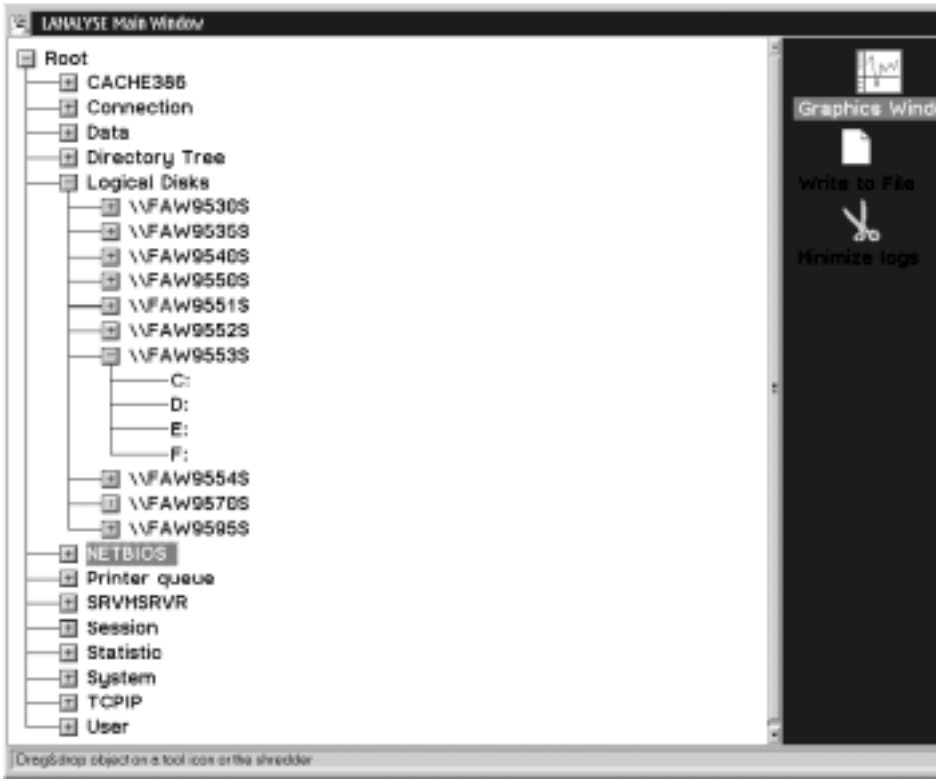


Figure 34. Main menu of the analysis tool

dropping it on a tool icon. The user can chose to select a single resource (the leaves in the tree) or a group of resources by selecting a node up to the root node. In this case all resources depending on the selected nodes are processed together. The graphics window tool will only accept groups up to monitor class level because up to that level all objects share the same class attributes, while it would not make sense to merge resources from different classes into a condensed view.

The first level of nodes in the tree are the workload classes. Data exist for these in the database. The second level consists of the names of the host machines. The last level includes the names of the devices or resources from which some information has been monitored and recorded.

The user can perform database maintenance activities. It is possible to remove data for a certain resource proxy by dragging a node to the shredder icon (which is provided by OS/2). The scissors icon can be used to remove all out-dated or invalid records from the database (*data cleanup function*).

Another helpful function of the analysis tool is the export function. "Raw" log records, or the result of the statistical calculations may be exported to an ASCII file in a character delimited format (CSV), that can be read by a spreadsheet program in

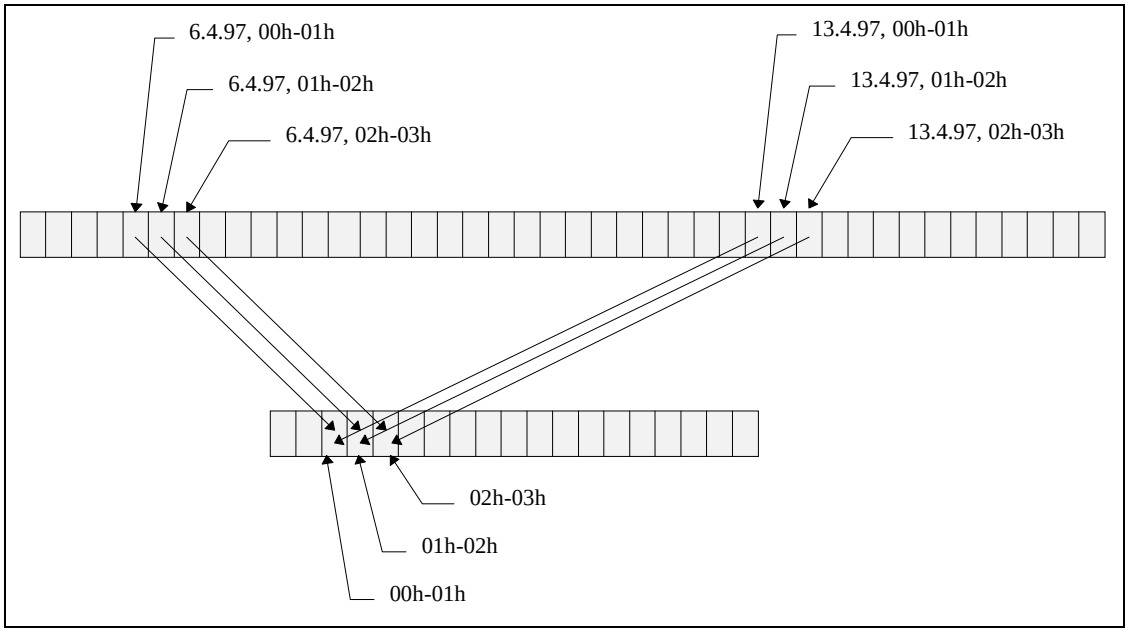


Figure 35. Mapping of daily workload records into a virtual week

order to do further work.

The most important function is the calculation and display of statistical reports. This is done by dragging one node of the object tree to the icon labeled "graphics window".

The essential data is read from the log database and the report window appears for the requested object (or group of objects). Four report pages are available within the window. The first report is the "week report" (see Figure 4).

All views have in common that the granularity of the picture is one hour. This is the granularity of the information in the workload database. On element in the picture (cell) represents a number of samples that have been measured during that hour.

The week report reveals a compressed view of the data. Each hour of the week is represented by one column. Within one data stream all records from the database are mapped to the corresponding hour (as symbolized in Figure 35). Mapping means that an average (for sample sums respectively the normalized sample value in case of absolute or normalized resource attributes) is calculated.

If a group of resources (members of the same branch in the object tree) are processed together - depending on the type of data - the tool builds a sum (delta

values like "kB sent to server") or an average (absolute values like "currently logged on user" or normalized values like "% CPU used") over the log records from different resources. For normalized information there is little point in building a sum. In this case the average is calculated.

The report shows the average value, the average minimum and the average maximum within each hour. If information is available, it also shows the lowest potential minimum (minimum of all minima) and the highest potential maximum (maximum of all maxima). If a linear trend is significant in a certain hour it is displayed as a line with a legend text. Normally the user can see whether there is an upward or downward trend, or whether the resource attribute has little changed over the last months. Figure 4 depicts an example of a graphics window and its elements.

A legend window shows the meaning of the different lines and areas found there. In the sample pictures the legend is placed at the left top of the reports. The user can move the legend window with the mouse, if it is hiding an interesting area of the report. A control window can be used to work with the functions of the report

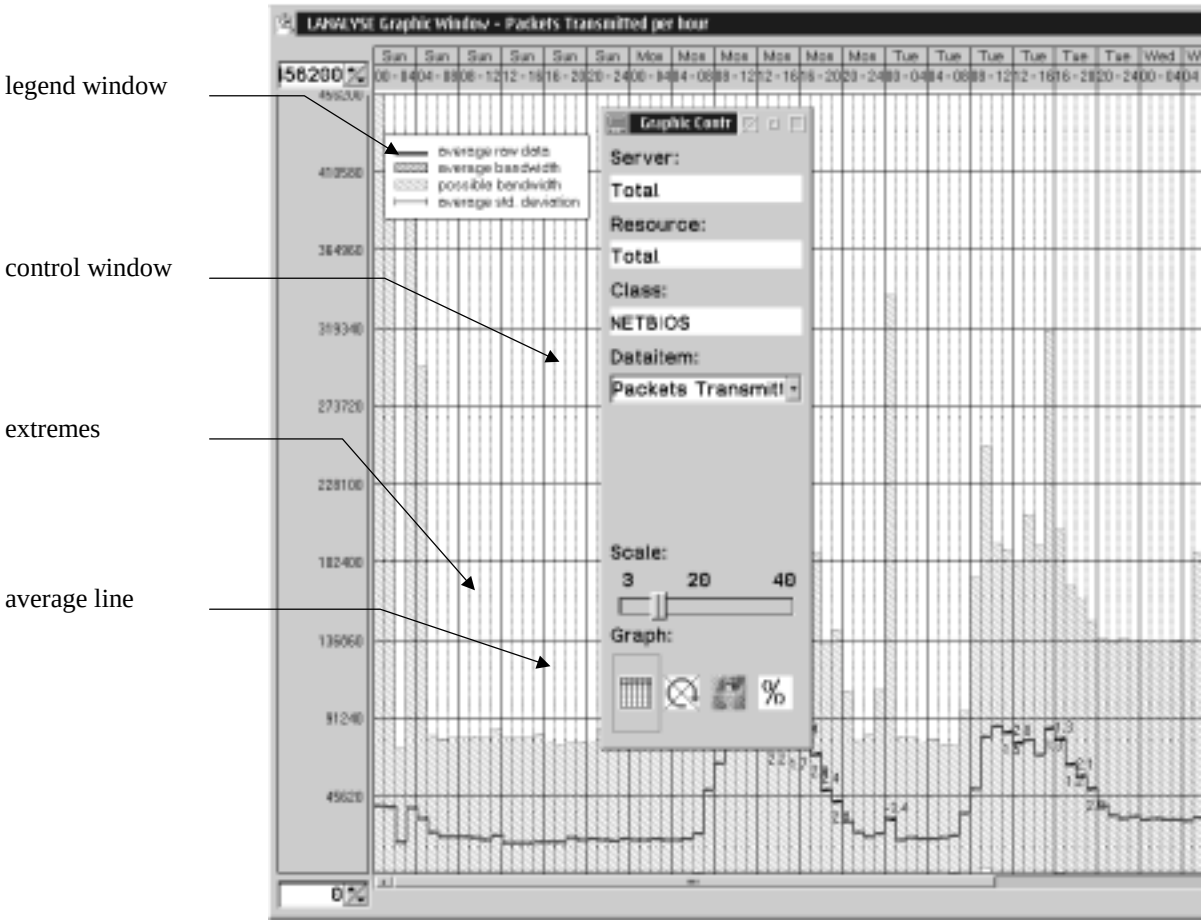


Figure 36. The week report

window.

The control window is used to select from the display options for the window. It displays the name of the server, the resource and the class for the one or more objects that are shown. The "Server" or "Resource" textbox contain the term "Total" when a group of resources has been processed and the graph displays the results of that operation. Under the entry "Dataitem" the user may select which attribute he wants to see. The window presents the list of available attributes of the

selected class. The control window "Scale" can be used to select the screen space that is used for each hour in the graphic. The last control element is the list of available views. Each view represents a different statistical evaluation of the information. They are explained later in this chapter. The user may switch to one of the views by selecting the corresponding symbol from the icon bar at the bottom of the control window. With the horizontal scrollbar below the graphics area the user is able to move the visible viewport to any area of the report. The vertical viewport (representing the visible value range) of the display can be adjusted, too, in order to magnify a certain part of the picture.

This "week report" is a condensed summary that has proved to give the most useful view. It presents a powerful means of understanding the load put on a system quickly. Without the techniques for long-term monitoring described in this paper it would be very hard and expensive to generate reports like this.

The second report is the "year details report" (see Figure 37). It shows all log records of the last year as they are stored in the database. For delta (accumulated) data, only the sum of the sample values is displayed (for example, the number of new print jobs during that hour). For absolute values the average, minimum and maximum are shown (for example, the number of active users on the system) if a

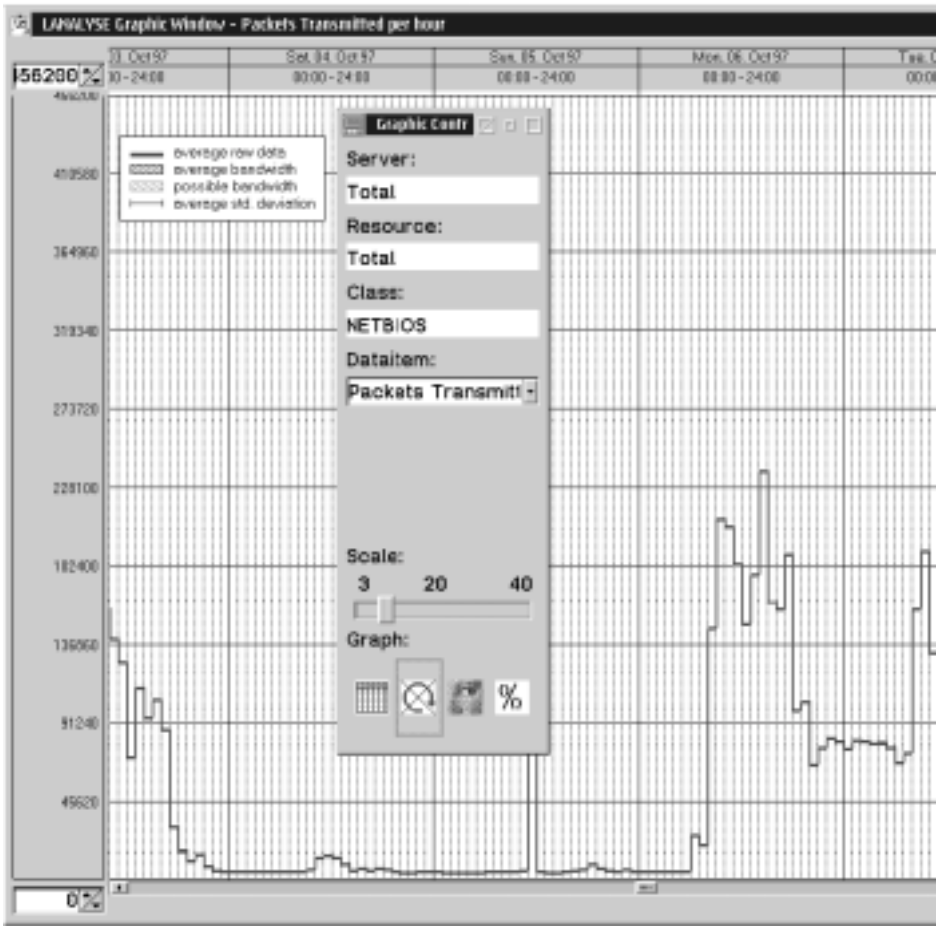


Figure 37. The year details report

group of resources (for example, all servers of a domain) are processed together.

The most obvious question an analyst asks is whether there is an association between the number of users and a resource attribute under consideration.

Therefore the third report shows the relation between the number of users logged onto the system and the attribute of the resource (see Figure 6).

The picture contains a *scatter graph*. The horizontal axes represents the number of users that were logged on to the system at the time the log record was created. The vertical axes represents the value range of the attribute under consideration. Now the tool draws a marker symbol (♦) for each record in the log file. The x position of the marker is calculated from the number of users at the time the sample for the

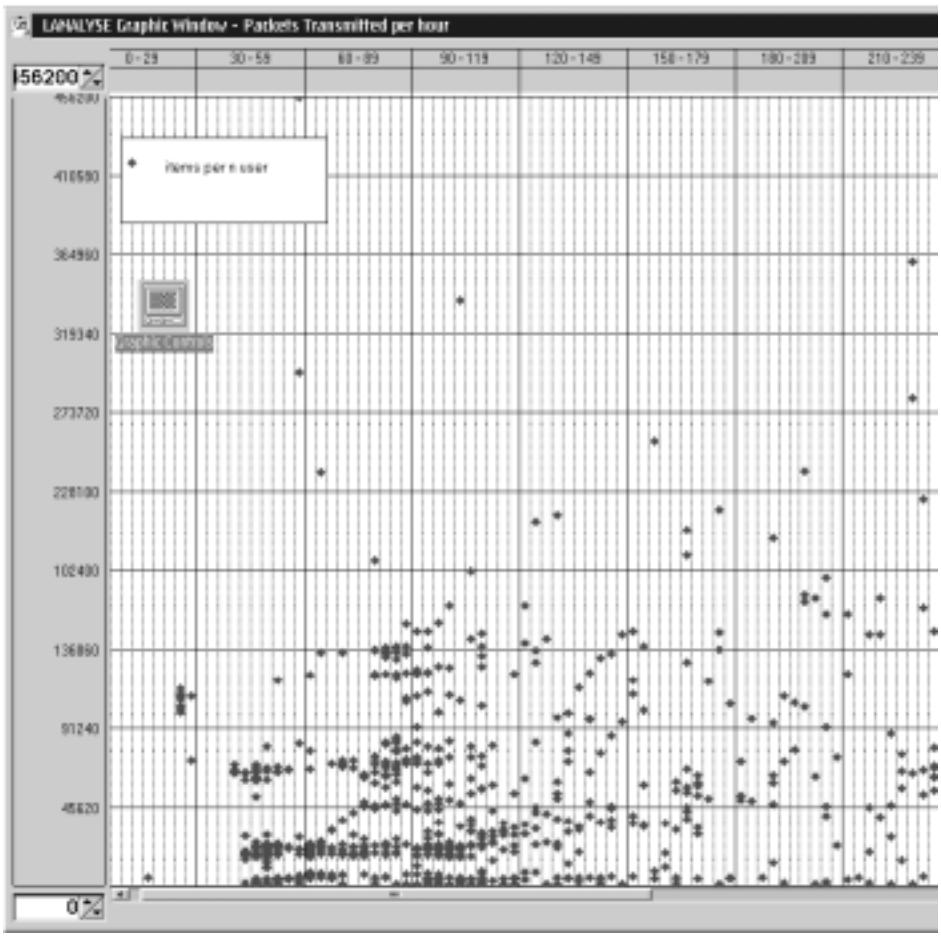


Figure 38. Association between an attribute and the number of users

resource has been taken. The y position is taken from the resource attribute.

A scatter graph is a common mean to display the relation between two data series that are connected by a common attribute, a matching timestamp in this case. The tool for detecting associations between resource attributes described in chapter 7.3 makes also heavy use of scatter graphs.

The last report shows the **availability** of the resource as can be calculated from the existing and missing samples in the log records. This window uses the number of samples in the "user log file" as a reference and compares that to the number of samples that is stored in log record of the resource under consideration. If the resource was available at a 100% there should be the same number of samples that contributed to the log record. For each log record in the file this comparison is performed for just one hour (one log record), a period of one day (24 log records) and a period of one week (24 * 7 = 168 log records). The result is expressed as a percentage of availability and is drawn in the window area (see Figure 39).

For this report it is assumed that monitoring has been enabled without interruption. If there are samples missing in the database this can have several reasons

- the resource has not existed (for example the host machine has been turned off)
- the resource has been intentionally out of service (for example, an administrator stopped sharing of a device)
- the resource has been malfunctioning (for example, memory shortage lead to an unrecoverable loss of connections)
- the network somewhere on the way between agent and manager has been interrupted; in this case it is very likely that no other (network related) services worked

In general the agent has the same "customer view" as any other user of a service. If the agent cannot access a service nobody else can. Therefore the information in the database gives a very good picture of the availability of a resource or a server machine.

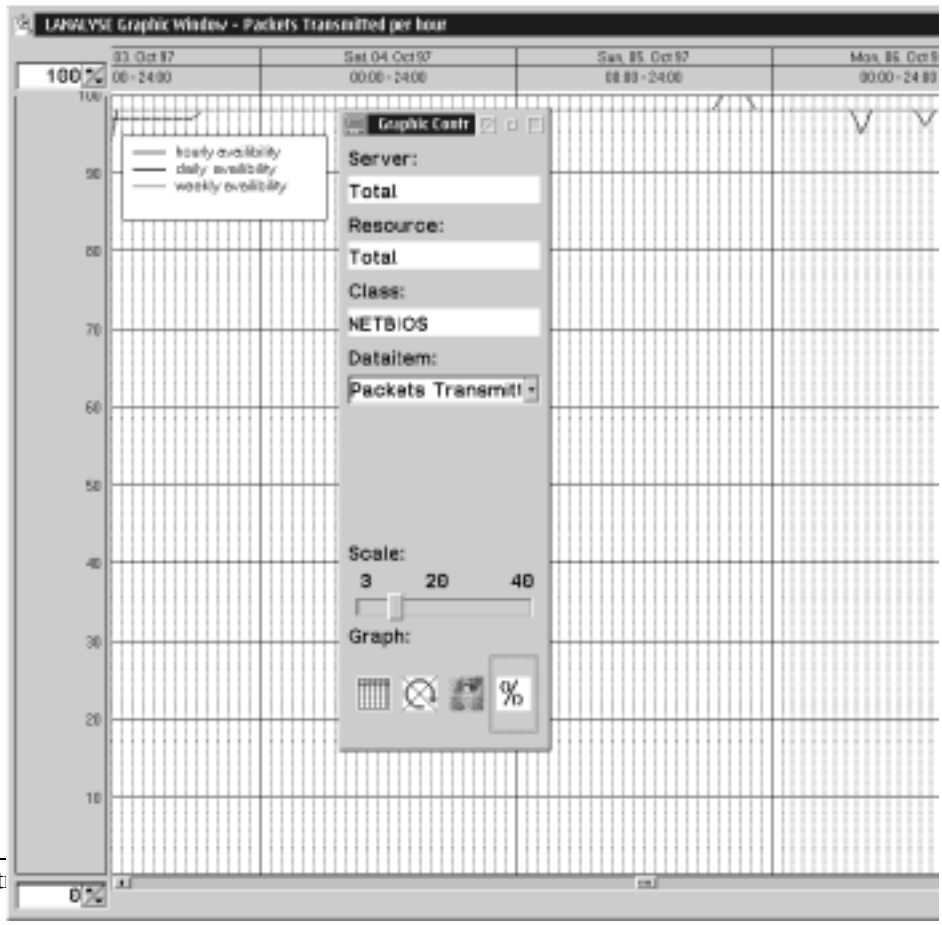


Figure 39. Availability report

7.3. Automatic Detection of Associations

An interesting followon work on monitoring and analyzing load data, which was perhaps the most difficult and challenging, was the automatic detection and presentation of possible *associations* between available resource attributes. A *correlation* describes the strength of an association between variables. An association between variables means that the value of one variable can be predicted, to some extend, by the value of the other. At the beginning of the project work it was one of our fundamental assumptions that such dependencies exist and that it should be possible to detect them after the monitoring tool has collected enough data and put them into a database. This chapter describes the algorithm that was developed for that purpose.

7.3.1. Problem Description

One important idea behind the work has been to find dependencies between different resource attributes to be able to give clues to the nature of the system. From that some information should be derived that could be used as input for a model in a specific type of server environment. Dependencies would be expressed as "good" associations between a number of workload data series as they are stored in the database. An association describes the projection of one or more (independent) source variables onto a (dependent) result variable for all possible value sets able to be taken. The correlation is calculated with the assumption that both variables are stochastic. Let us consider the following ideal example:

time (index)	x	y
0	1	4
1	2	7
2	3	12
3	1	4
4	2	7
5	3	12
6	1	4
...	...	...
8759	2	7

Figure 40. Selection of two resource attributes from the database

Let x be the source variable; the value set is {1, 2, 3}.

Let y be the result variable. Let us assume that there is a direct dependency of y in the form of $y = x^2 + 3$. x and y are resource attributes for which information is stored in the database. After a full monitoring period we obtain a table with up to 8.760 records like in Figure 40. Sample values from different resource attributes that are connected to the same time (index) form a record. Note, that the database contains time stamps which are mapped to an index when data are loaded into memory.

The question is whether an association between the two variables exist. A common method to test for an assosiation is the use of linear correlation and regression. We were looking for a procedure that could detect non-linear assosiations as well and that would be fast enough to do the calculations on a standard PC. When designing a suitable procedure we had the following goals in mind:

- design an algorithm that can detect any reasonable association, independent of assumptions about a model function (for example, linear association)
- the algorithm should be controllable by use of parameters and rules; it should be adjustable from the outside, possibly via the GUI
- it should be able to process the complete database associated with a case study within reasonable time
- it should be efficient and fast enough to execute on the monitor front end (an OS/2 based PC)

- it should be possible to generalize the algorithms to extend their application to more than two variables

Without any preassumptions about the data the program should take any combination of attributes and test them for an association. Three factors influence the overall complexity of that task:

- the large number of attributes as the number of tests increase with a power of the number of attributes
- the number of model functions that could be the base (hypothesis) of each test
- the large number of data samples that has to be processed for each attribute

In order to reach these goals and to address the three compexity factors we developed an algorithm that is based on statistical approaches that go beyond the common linear regression calculation. Usually to find an association an analyst selects a "model" to test the association. Most often a linear model in the form "y = a * x + b" is used (linear regression) but other models can be used as well [6]. Then the parameter in the model (like a and b in the linear model) are calculated in a way that the sum of the squares of the distance between actual data points and the model function is minimized. Than in the case of linear regression the correlation coefficient *r* can be calculated. This coefficient is a number between plus one and minus one. If the absolute value of r is one this means an exact match between data points and model function and zero means no association. For non-linear data samples a transformation function can be applied to the data that will allow the use of linear regression on the transformed data. The correlation coefficient is invariant against any transformation of the data.

The problem here is that we have a great number of data series to compare and we do not know which model function to use. Existing techniques for automated detection use a list of predefined model functions and apply each of them to the problem. The one with the best correlation coefficient (or equivalent measure) is taken as the most valid model. Such approaches not only test for an association but identify the model function as well. But this is extremely expensive in terms of processing power and elapsed time. It would not be possible to apply this in our situation.

To address the problems mentioned before the program

- eliminates useless attributes by a number of criteria as soon as possible to reduce the number of attributes for the later processing (the number N of all available attributes is reduced to n the number of attributes with meaningful information, e.g. non constant values)

- eliminates useless attribute pairs as soon as possible (a pair may contain too few valid data pairs)

- further compresses the data to reduce and simplify the number of math operations

- does not rely on any model function

- is devided into several steps that can act in parallel in a pipeline mode

Another example for association and association testing without a model function like the "Rank association coefficient" see [35].

Accumulated Information

Sometimes the accumulation (sum) of several resource attributes is an important information in itself. For example, each server has an attribute "number of users currently logged on". This attribute will be always zero except if the server is configured as a domain controller or backup domain controller. The number of users who are logged on to the domain and who may use services from any server in their domain, is the sum of the attribute values from all servers. In most cases the accumulation of resource attributes is of interest for the human analyst in order to obtain an overview (for example, the sum of packets on the network or the total of accesses on a server disks).

Therefore for each class of resource attributes accumulated information over all servers is generated by the algorithm and the summary items are processed like “normal” resource attributes.

7.3.2. Data Preprocessing

What we get as a result of the logging facility of the workload front-end is a set of resource attributes. Each resource attribute contains load (or status) information about a certain resource attribute on a certain server for each hour of the last year (or less if the resource or the monitor has not been available for some time). Each record p_i in the resource attribute is associated with a timestamp t_i. i is the (time) index which references an hour of the last year. Each timestamp t_i can be mapped to an index i. If the timestamp is outside the range (older than one year) the sample value is ignored. Thus we know when the data have been sampled. We may define the data series of one resource attribute as

$X := \{ p_i / 0 \leq i < 8760 \}$ and t_i is the timestamp associated with p_i

Note that a resource attribute could be empty. For better understanding of the problem we illustrate the number of times an association calculation has to be performed. In the following paragraphs the selection of a number of resource attributes from the database is called a **tuple**. In our work we use a tuple size of two. That means that for each tuple two resource attributes are selected. The algorithms could be generalized to any tuple size.

First all available attributes have to be preprocessed to generate a useful basis for the algorithm. All attributes are loaded from database (which includes some error handling and out-of-bound omission). All records are adjusted in a way that their time stamps matches and that values from the same time interval are compared to oneanother. Virtually a table like in Figure 41 is created. Each cell in this table contains the data sample of the corresponding attribute at the time that is represented by the line index in the table.

time	a_1	a_2	a_3	...	a_{n-1}	a_n
0						
1						
2						
3						
...						
8756						
8757						
8758						
8759						

Figure 41. Preprocessed raw data

Now we build pairs for all i $(p_i, q_i / t_i)$ with $p_i \in X$ and $q_i \in Y$ (that have a matching timestamp t_i; this does not mean that the timestamps are identical, but they have to map to the same time index i and that means that they refer to the same hour of the last year). The number of pairs in (X, Y) cannot be greater than the minimum number of records in each resource attribute, and may be smaller if some p_i or q_i are missing for certain time indices.

This means that all permutations of two resource attributes are created. Each pair builds a tuple. Each tuple is feed to the association algorithm. The biggest challenge then is the sheer number of tuples. Within each tuple both resource attributes are used as the dependent variable (y) and it must be tested whether the other resource attribute is associated with it. *For our algorithm the order of x and y is important*. We obtain for the number of tuples:

$$Num_{Tuples} = \binom{n}{2} * 2$$

or

$$Num_{Tuples} = \frac{n! * 2}{(n-2)!\ 2!} =$$

$$Num_{Tuples} = \frac{n!}{(n-2)!} =$$

$$Num_{Tuples} = n * (n-1)$$

with n is the number of existing resource attributes (after the filtering). Note that each monitor class contains many (up to 256 attributes) and that n is not the number of resources but the number of resources times the number of attributes of their monitor classes.

Example: If a case study contains 1.000 resource attributes the algorithm has to check 999.000 tuples. To better illustrate that number imagine that the association processing of each tuple takes only 15 seconds (including reading the data from the database and writing back results), then the whole process would take more than 173 days.

Scatter graph

A scatter graph is a common means to display data series of two variables. It can make it easy for the observer to assess a potential association between the variables. Figure 42 and Figure 11 show examples of scatter graphs.

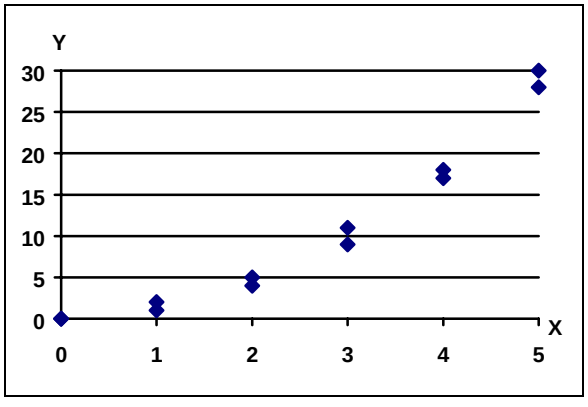


Figure 42. (non-linear) association

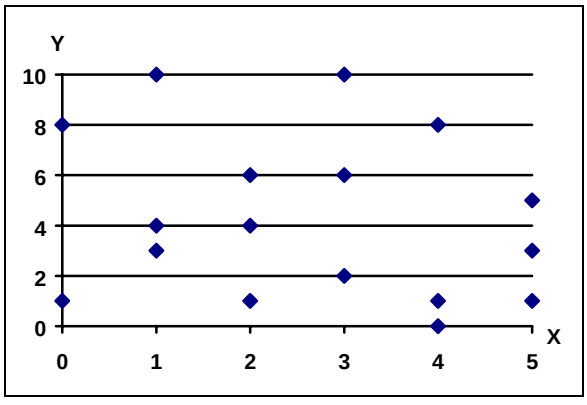


Figure 43. no association

Scatter graphs are used later to show associations that have been found in the databases of the case studies. The user interface of the detection tool extends this notion of a scatter graph. One problem with scatter graphs is that the number of identical data pairs is not visible. To circumvent this problem the color of the

marker expresses the number of (x, y) pairs that exist for each coordinate (see Figure 16).

7.3.3. Summary of the Algorithm

The detection algorithm consists of five stages. Each stage is implemented as a separate thread and acts in parallel to other stages. It receives the results of the previous stage, filters useful data and generates input for the next stage. The stages are called

1. Initialization - Database Extraction
1. Attribute preparation
1. Tuple Building
1. Tuple Processing
1. Association Probability Assessment

The following paragraphs briefly describe the tasks of each stage.

Initialization - Database Extraction

The first stage traverses all entries in the load database. Each entry represents the data a monitoring class retrieved for a resource. An entry contains the data of one or many resource attributes. The following steps are performed for each entry:

1. The thread filters resource records relevant to the actual detection process (the user may constrain the detection to certain hosts, for example, all machines of the domain FAW9500D)

1. Then the thread loads the data for the entry. Underneath the database layer removes invalid or out-dated information. Time stamps are mapped into a time index which is used to store data into an array in memory. Time stamp or time index are not stored in memory.

1. It allocates and initializes the data structures for storing the information. Resource attributes are processed independent from one another.

1. Finally it creates and totals up the accumulated information. These data appear like data loaded from the database. Later stages do not distinguish between data from the database and accumulated data.

Note that the load database consists of resource records and load data records which contain all resource attributes that are part of a class (see the description of data structures in chapter 6). The database may contain information about many domains and servers. The analyst can focus the detection on a certain part of the information by specifying a *server name filter*. This filter can contain wild card characters (‘*’, and ‘?’).

The load data are converted into the format used for further processing. For each item an array of 8.760 elements of type *unsigned long* (one for each hour of the last year) is allocated and that structure is initialized with the data from the database, or the value “no data” if there is no sample available at the time. After this process the program knows how many “hits” (= samples) have been found during the last year. Note that this data structure is later discarded and replaced by a more efficient and smaller memory structure.

As part of the elimination of useless attributes, a filter rule is applied (elimination step I): if there are less hits than a certain percentage of the maximum possible number of entries (8.760), the information for that resource is not processed any

further. With this filter very new, too old, or unreliable resource data are eliminated from the detection process very early.

Otherwise the resource is added to the summary information of the server (if several class resource entries for the server are possible), and to the overall summary for the class of the resource.

After this the resulting data structure is handed onto the next stage.

Attribute Preparation

In this stage a compressed version of the information is produced because this structure will be needed until the end of the detection process and even longer - for on-line presentation. The sample values are linearly transformed from their original value range of zero to $(2^{16}-1) * n$ (with n is the number of samples accumulated during one hour; the default value of n is 60, the maxium value possible is 120) to a value range of zero to 254. Sometimes such a transformation of the raw data is refered to as *classification*.

A new array of 8.760 elements of type *byte* (or *unsigned char*) is allocated. The program looks for the minimum and maximum values within all records of the resource attribute. The next rule is applied: the difference between minimum and maximum must be greater than a certain parameter (the difference must be at least greater than zero). This rule (elimination step II) eliminates resource attributes that never change (for example, the number of print jobs on a server that is not a print server). Such attributes do not contain information and would only confuse the results of the algorithm: any value from source resource attribute X would "perfectly" map to Y.

Now, for each defined element with index i of the resource attribute the projection

$$t_{size} = (v_{max} - v_{min}) / 255 + 1$$
$$v_{new}[i] = (v_{old}[i] - v_{min}) / t_{size}$$

is applied. v_{min} is the lowest value of the resource attribute and v_{max} is the greatest value. Note that this algorithm similar to the calculation of the correlation coefficient is invariant against transformations applied on the input data.

All v_{new} are in the range $0 \le v_{new} \le 254$ (because integer arithmetic is used). The value 255 is used for undefined elements (not a number).

The value range associated with a value between 0 and 254 is called *a tile*, because one may imagine the original value range covered with a number of "tiles". t_{size} is the size of the tile (relative to the original value range). It must not be zero and is therefore increased by one. If the difference between minimum and maximum is less than 255 the tile size is still one. An original value range of a resource attribute can be covered by n tiles with $1 < n < 255$.

The value range (r_{min}, r_{max}) of a certain tile t_i is defined by

$$r_{min} = v_{min} + i * t_{size} \quad \text{and}$$
$$r_{max} = v_{min} + (i + 1) * t_{size} - 1$$

The use of the notion of tiles eliminates small variations in the original values.

The next rules are applied (elimination step III): The resource attribute must cover at least a certain number of tiles, otherwise it will be eliminated. If the rules eliminate a resource attribute because it does not cover enough tiles, this may be due to an out-of-bound value distorting the transformation, for example, assume that the normal bandwidth of a resource attribute is between 0 and 500, but for an unknown reason, there is one record with 500.000. Now, the size of one tile is

1.969. All values within the normal bandwidth are transformed into tile 0, the only extreme value into tile 254. Therefore only two tiles are in use (0 and 254).

In order to handle this case, the program tries to eliminate extreme values in a more sophisticated way than the database layer does. Both from the lower and from the upper margin tiles with less hits than CONSTRAINT_IGNORE_TILE (a parameter specified as 2) are removed. In other words: at least three hits must exist for a tile to stop the further restriction of the valid value range. New minimum and maximum figures (v_{min} and v_{max}) for the original values of the resource attributes are calculated. If there are tiles that can be removed, the attribute preparation starts right from the beginning. Values that lie outside the new boundaries are ignored. Theoretically, this process can be repeated several times until all out-of-bound values are removed (or until the number of available records left is too small to continue).

In the example the algorithm would detect that only tile zero would contain more than two hits. The new minimum would be 0, the new (initial) maximum would be 1.969. In the next iteration, the value 500.000 would be ignored. 500 would be detected as the maximum of the remaining records. The new tile size would be 2 and the resource attribute could be used for further processing.

The result of this stage is that for each resource attribute that passes a number of filters the raw data have transformed in a compressed structure. Time stamps have been mapped into a time index and sample values have been mapped from a four byte integer into a one byte integer to be able to keep all resource attributes in memory.

Tuple Building

Until now suitable attributes have been prepared and the elimination steps have reduced the amount of attributes for processing. This stage selects all combinations of two attributes. When a new attribute arrives, the program builds a new tuple for each element in the set of processed attributes. A number of tuples (x, y) are created and pushed to the next stage. After all attributes have been processed, the new attribute is added to the set of processed attributes. One can imagine that this stages selects all possible combinations of two columns from the table in Figure 41 and hands each pair over to the next stage.

The attributes x and y are examined to see if one of them is an accumulated item and the other is an element of that accumulation. It would not make sense to check for association, because if x is a contributor to y the association is implicit.

Tuple Processing

In order to test for a potential dependency between two attributes a suitable statistical algorithm would be the use of *contingency tables* and the calculation of the χ^2 value [38]. We will give a short overview of this method and some reasons why we did not use this method but implemented a simplified approach instead.

Contingency Table Method

A contingency table is a r x s table (r columns and s rows). r and s are the number of classes for each of the two attributes. The transformation describe before can be seen as a classification into 255 classes. Each cell of the table contains the number of data pairs where the value of x fits into the class within r and the value of y fits into the class within s (see Figure 44).

	x_1	x_2	x_3	...	x_r	Σ
y_1	h_{11}	h_{21}	...	...	h_{r1}	$h_{.1}$
y_2	h_{12}	h_{22}	...	...	h_{r2}	$h_{.2}$
y_3	h_{13}	h_{23}	...	...	h_{r3}	$h_{.3}$
...	...	...	...	...	...	
y_s	h_{1s}	h_{2s}	...	...	h_{rs}	$h_{.s}$
Σ	$h_{1.}$	$h_{2.}$	$h_{3.}$	...	$h_{r.}$	

Figure 44. Contingency table

In our case h_{ij} is the number of data pairs where the value of x is (i-1) and the value of y is (j-1). r and s would be 255 except for attributes with a reduced value range. After the number of occurencies are found for each cell the column sums and row sums are calculated:

$$h_{i.} = \sum_{j=1}^{r} h_{ij} \quad \text{and,}$$

$$h_{.j} = \sum_{i=1}^{s} h_{ij}$$

After the contingency table is created for each cell an expectation value is calculated. This is the number of expected data samples under the assumption that both attributes are independent. The expected value $\hat{e}_{ij}$ is

$$\hat{e}_{ij} = h_{i.}\, h_{.j} / n$$

n is the total number of data pairs (up to 8760). In order to get a non-zero expectation value both column sums and row sums must not contains any zeros. The square sum of the differences between actual occurencies and expected values is a χ^2-distribution. We accumulate

$$X = \sum_{i=1}^{r} \sum_{j=1}^{s} (h_{ij} - \hat{e}_{ij})^2 / \hat{e}_{ij}$$

We compare the result with the quantile of the χ^2-distribution given the significance level and (r-1)(s-1) degrees of freedom. A program may calculate or lookup the quantile in a table. If the X is significantly greater than the quantile then the initial hypothesis of independence between the two variables is wrong. On top of that the C-value (Pearsons contingency coefficient) can be calculated as

$$C = \sqrt{(X / (X+n))}$$

C = 0 means total independence and C = 1 means total dependence between the two variables.

The main reason we did not use this method was because it is relatively computational intensive. All calculations have to be done as floating point calculations, including the transformation of the integer numbers to floating point

representation, and a huge number of operations is required for each step. Another problem is that the expected value for each cell must not be zero and that means that there must not be empty classes. However given the raw data from the database that cannot be guaranteed. In reality it is very common that for a certain value no (x,y) pair exists.

In order to achieve optimum performance we want to keep pure integer arithmetics, keep the number of required operations low and the algorithm should work with empty classes. And, as mentioned before, using rules and parameters an analyst should be better able to influence and control the detection process.

Association Table Method

This approach tries to further reduce the number of values that must be processed. The downside is that it works "directionally", which means that the alogrithm has to check x⇒y and y⇒x. That is not necessary with the chi-square test. An examination of all 8.760 possible records within the attributes is performed to see whether a valid data value does in fact exist in both items. For each tile in x the number of hits to the respective tile, average and variation of y is calculated for all valid data pairs x and y. This is done in both directions (x ⇒ y and y ⇒ x)

The next rule filters pairs with low significance: if the number of existing record pairs (x_i ,y_i / t_i) for all i: $0 \le i \le 8.760$ is less than a certain parameter, the tuple is not processed any further.

x	average of y	number of records	std. dev. of y
1	4	2593	0
2	7	3964	0
3	12	2979	0

Figure 45. Association table

The algorithm generates an association table (Figure 45 shows a very simple example) and then checks, whether there is an association between x and y or between y and x. We examine the first case; the second case is calculated in the same way.

In order to ascertain the likelihood of there being an association from x to y, we have to assume that for a certain value of x_n in x all possible values y_k are more or less the same. For each tile in x, the program calculates the average and the variation of all y_k associated with that tile.

The following picture gives a graphical representation of the association table that is now available for x. The same table exists for y → x.

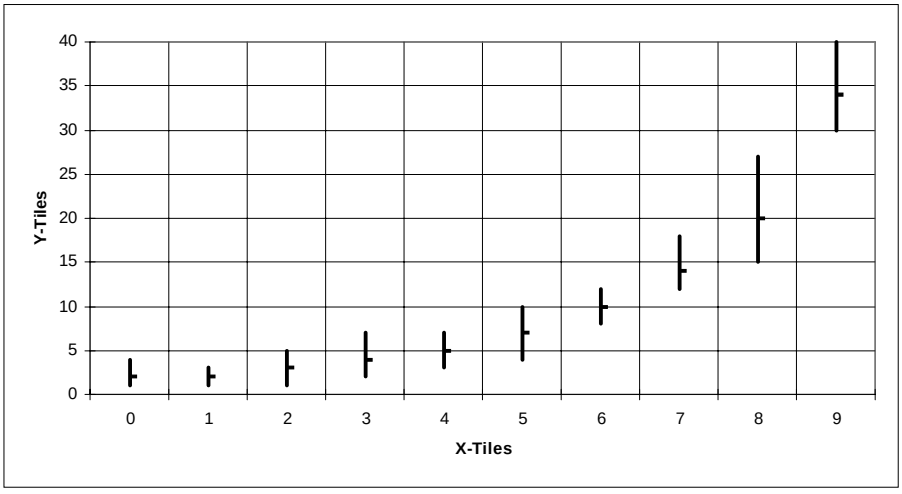


Figure 46. Graphical presentation of an association table

The association table is handed onto the next stage.

Association Probability Assessment

The task of the last stage is to assess whether the relative position of the average of all tiles and the variation within each tile makes it likely that an association between x and y in the form $x \rightarrow y$ or $y \rightarrow x$ (or both) exists.

To make this assessment, a system of “penalty points” is used. Starting with zero points, each tile is examined with a number of rules applied to it and each rule may add penalty points depending on the conditions below. The parameter CLASSIFY-_MAX_POINTS defines the number of penalty points that each tile may contribute on average, with the tuple still being considered as a "good" association. The rules and the penalty system are calibrated with a number of artificial test data sets that are generated with several functions (for example trigonometric functions), in conjunction with random variations that supersede the function values.

Finally, each tuple has two sums of penalty points: one for $x \rightarrow y$ and one for $y \rightarrow x$. If the number of penalty points is greater than an adjustable limit the tuple is discarded. In this case there is no association between the two attributes. Otherwise, the tuple is added to the set of result objects, which can be viewed and analyzed by an end-user (see “User interface” below).

Penalty points are added due to the following rules.

- If there are no hits to the tile, no points are added because there is no information in it.

  (The following conditions assume that there are hits to the tile under consideration. This tile has index i and will be noted as x_i.)

- Points are added if index i is greater than zero and if the previous tile x_{i-1} does not contain a hit,. This is to ensure steadiness.

- Relative to the standard deviation more points are added. The higher the variation, the more penalty points are given for that tile, with an upper limit.

- If $x_{i-2} < x_{i-1}$ and $x_{i-1} > x_i$ or $x_{i-2} > x_{i-1}$ and $x_{i-1} < x_i$, points are added. This rule supports monotonic relations.

- The average of the actual tile is compared with the last five tiles (if these exist and contain more than one hit). Points are added if the average value of the last five tiles is equal to the actual value. This is to eliminate “constant” attributes (and which escaped the eliminiation steps).

- If the difference between the average value of the last five tiles and the actual tile is greater than a certain parameter value, points are added. This is again to enforce steadiness of the potential projection function.

- Points are added if their are too few records in a certain tile. This rule eliminates pairs with too low significance in the distribution of their values.

Summary

The algorithm avoids any assumption about the potential association between x and y. It does not contain complex mathematical functions and is very fast and efficient. It is suitable for processing a great number of resource attributes on a standard PC. It can also detect complex projection functions (e.g. trigonometric functions), which are overlaid by natural, coincidental variations. The algorithm and all the

control parameters in particular, were tuned by test runs that included complex functions plus random number variations.

7.3.4. Result Index

It has already been mentioned that the load database "lives" and changes constantly. Every hour new records are added to the database and other records become outdated. Therefore analysis results can change every day. The association analysis writes its results into a separate file - the *result index*. This file contains all result tuples that have been found in an invocation of the analysis tool. A counter is part of each result entry. With this it is possible to see how often an association has been found. A user can use the counter together with the penalty score to decide whether to use the record.

7.3.5. User Interface

List of Detected Results

The window in Figure 15 contains all tuples likely representing an association. The result tuples are sorted by the likeliness of a good association. The user may choose another sort criteria.

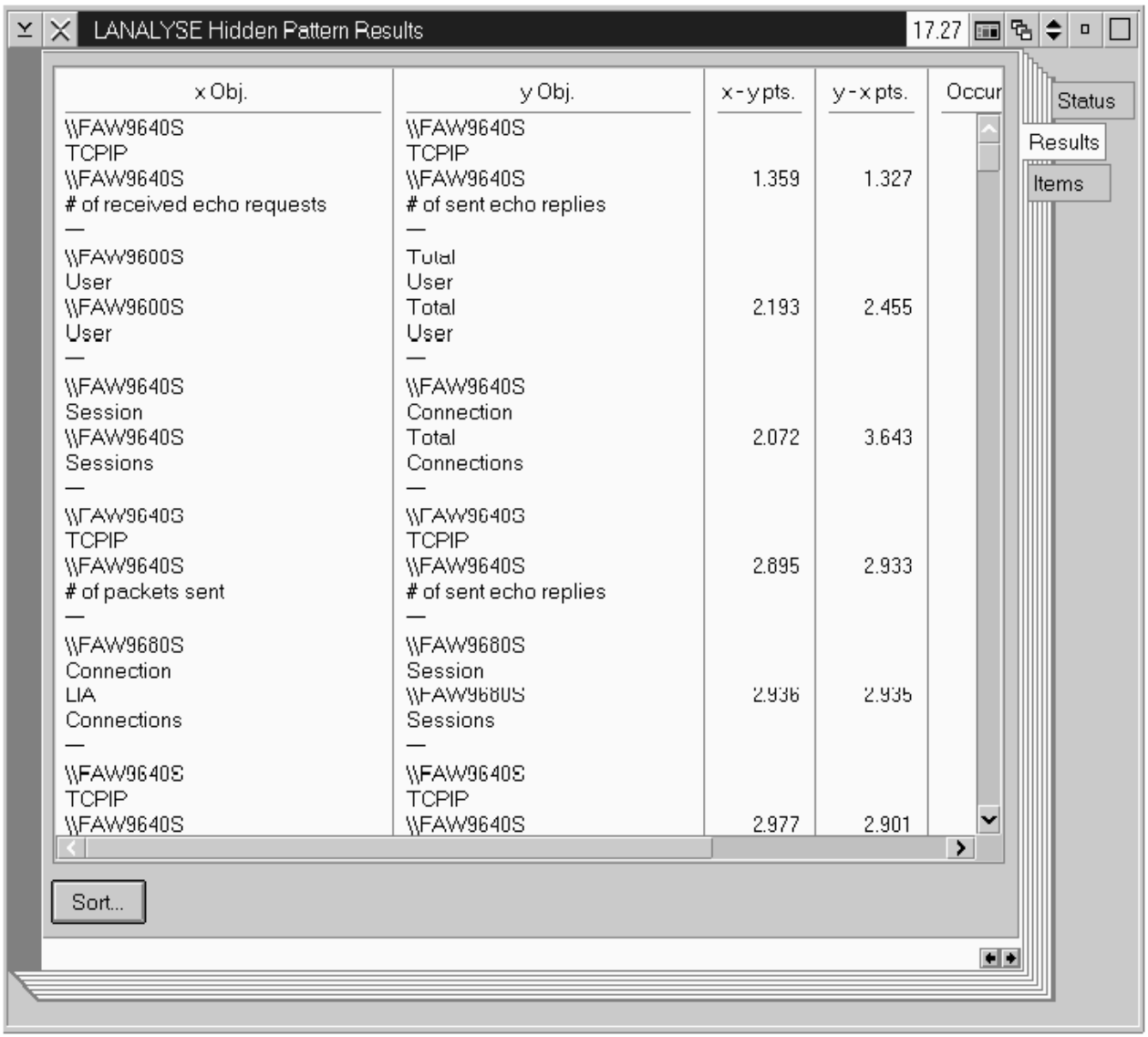


Figure 47. List of Detected Associations

Scatter graph of an association

When the user "opens" a result record, a graphic window like the one in Figure 16 is displayed. It shows the number of hits for each pair x, y. The user can additionally turn on figures for the average and standard deviation of all values.

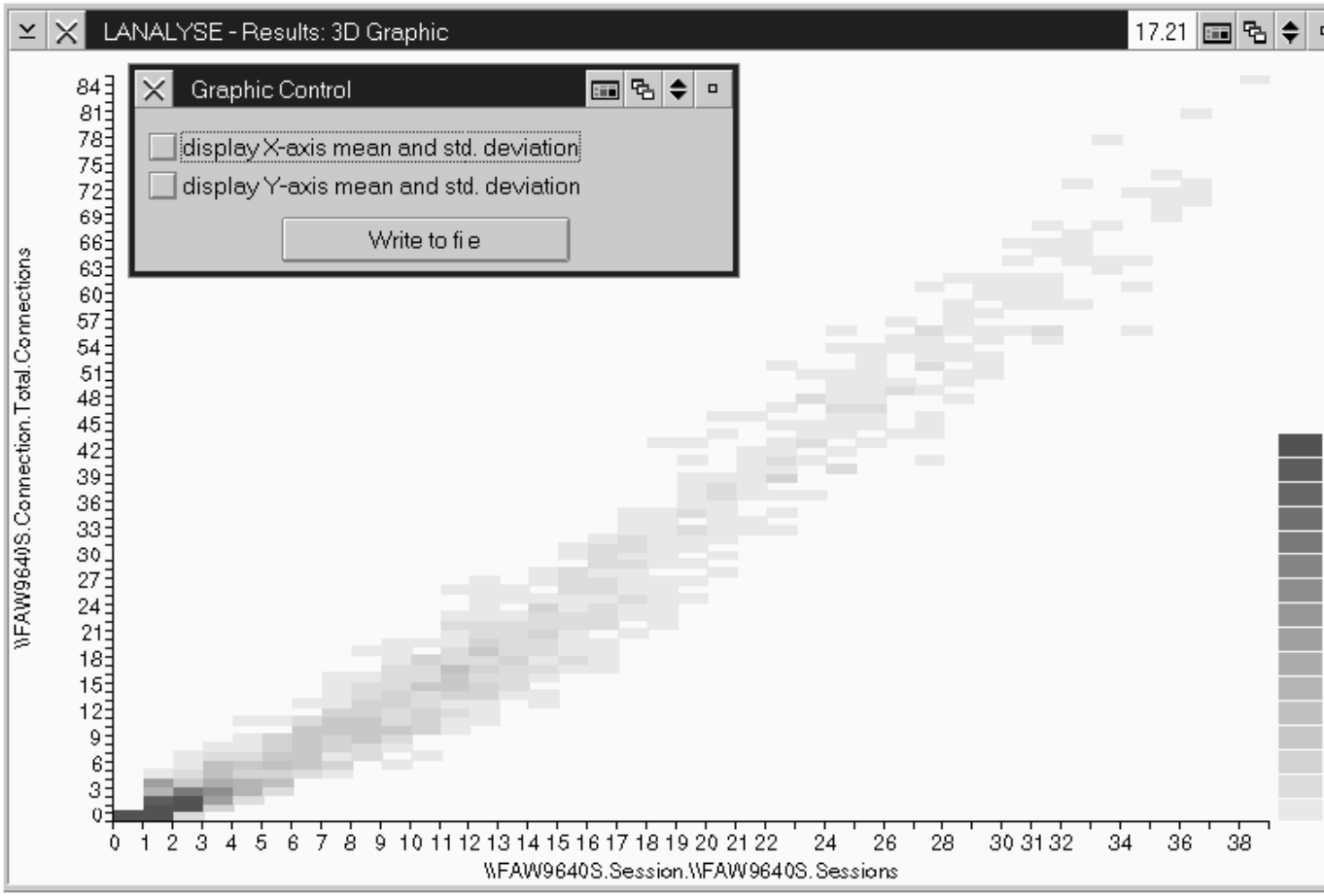


Figure 48. Association scatter graph

The picture above shows the good association between the number of sessions to a server and the number of existing connections. Besides the fact that there really is a dependency between the two attributes an analyst can draw further conclusions from this example:

1. Most of the time there are between zero and three sessions with zero to three connections.
1. There is a maximum of 38 sessions and 86 connections on that server
1. The number of connections rises faster than the number of sessions; it seems to be a non-linear relation between the two attributes.

Information like this could be very useful and valuable for a simulation model of a system under consideration.

7.3.6. Data Export

To enable association results to be included in a document (like this one), the data for a certain result can be exported to a file. The user can do this from the "Association Scatter Graph Window". The resulting file, that contains the data, has the form of $x_i;y_i$ (the first line contains the names of item x and y):

```
Total.TCPIP.Total.# of unicast packets sent;Total.TCPIP.Total.# of packets
sent
48;10
48;12
48;16
49;9
52;9
53;19
55;9
57;12
55;13
60;12
59;14
61;14
62;17
69;11
...
```

A spreadsheet application like Microsoft Excel can import the data and display a graph which can be exported to a word processor.

7.3.7. Constructing a Tabular Model

The idea to construct a tabular model is based on the assumption that with a database of detected associations a network of related resource attributes would emerge. Each entry would represent the projection from one item to another in the form of a table. Figure 49 depicts a very simple model. If there is a relationship it could be expressed as an equation in the form

$$y = f(x) + r(t)$$

with f(x) is the technical relation (due to the implementation)
r(t) is the time dependent random variation that overlays the data

The term { f(x)+r(t) } would be replaced by an association table, in which it would be possible to look up any value of y given the value for x. The random variation is already covered in the table, as average values of y were calculated over the observation period. Moreover it would be possible to work with the standard deviation, if this is important for the analyst.

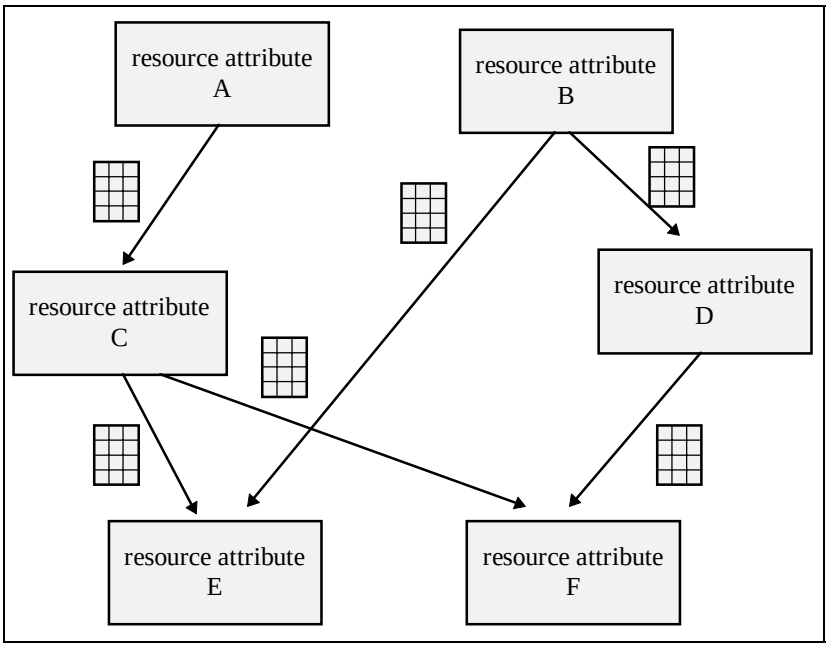


Figure 49. Model build from projections

It could occur that not every value of x would form part of the table. Inter- and extrapolation algorithms could however be used to calculate missing values in the table. That way it would be possible to use the tables to predict the behavior of the system when parameters change.

Figure 49 shows a network of associated resource attributes. A simulation engine could feed parameters into the network and observe how parameters change for different resources. Analytical processing could be applied as well. Questions in the form "What happens when attribute x takes the value of n at time t?" could be answered this way.

The tabular model would perfectly represent the observed system. First of all, it would it make possible to understand the internals of the system. By using the individual tables, statistical methods could be used to remove random variations and find an association function, for those interested in that. Furthermore, they could be used to analyze its behavior under different conditions, for example, if the number of users increased.

An intention to change (parts of) the system, would have to be reflected by adapting the tables manually or by replacing or adding relation functions or artificially generated tables. The processing engine necessary to work with the tables would have to be able to accept them from different sources. It would also have to be possible to add tables to the database.

Conclusion

We believe that the approach described in this chapter could lead to an understanding and model that matches the complexity of today's client/server environments. While it is not intended to replace analytical methods and simulations, the method of automated modeling a system from measured resource data would be a valuable addition to classical techniques.

In practice, with the currently available software products evaluated and connected to the monitor, the idea of tabular models has not worked as intended. No complete "network" of attributes could be found. The major reasons are

- that server applications do not provide information about their internal resources or even their external resources.
- that only simple associations (one source to one result item) could be considered. While these type of dependencies form the majority of cases, more complex dependencies still remain undiscovered.
- that a measurable change in a dependent resource attribute can appear with a significant time latency. Resources like a cache can keep up near optimum throughput for some time and than suddenly change their behavior.
- and that detected associations can represent where "local" conditions that cannot be generalized and would not fit into a model.

As a consequence, a very incomplete "network" remains today that cannot be used as a model.

Results of the analyses of the long-term load data of real client/server environments

8. Case Studies

This chapter contains the results of the case studies which were performed over the period of four years in several locations. We provide this massive information as a reference because we experienced a lack of actual data from real environments. It would be beyond the scope of this paper to find an explanation for every chart. We think that it may be useful in certain circumstances to check the range of certain parameters and how they change over time.

Each log database, that resulted from the deployment of the tool, contains millions of records and each record represents 60 samples for one hour of measurements. In order to represent this huge amount of information in this document only compressed information can be printed. Each observed environment consists of a number of servers. In the pictures the data from all the servers are summarized. That means that each picture represent the sum of the data for a certain attribute for all the servers in the observation group. The data covers a period of one year and is collapsed into an "**average week**" (see chapter 7 for a description of this data mapping).

That means that for each hour of the week the average for minimum, average and maximum values of all records that fall into that hour have been calculated. This values may represent absolute values (like "number of users connected to the domain") or cumulative values (like "kB sent to server during one hour"). The context and naming of the data item usually makes clear what is meant. In some cases the meaning of the values is explained explicitly.

Where possible a short comment or explanation is added to each graph. If there is nothing to say about a picture a comment is omitted. Chapter 2 contains the summary of the case studies which includes major observations and some hypotheses to explain the measurements.

8.1. Case I

The first scenario is a domain of ten server machines which support up to 90 user in an backoffice type of work group. This domain was studied from the beginning of our work (starting 1995) and was used to test and verify assumptions and research the basics of distributed monitoring.

Because this domain was studied over several years the server roles and usage profile is very well known to us. That information helps to understand or interpret the results of the monitoring process.

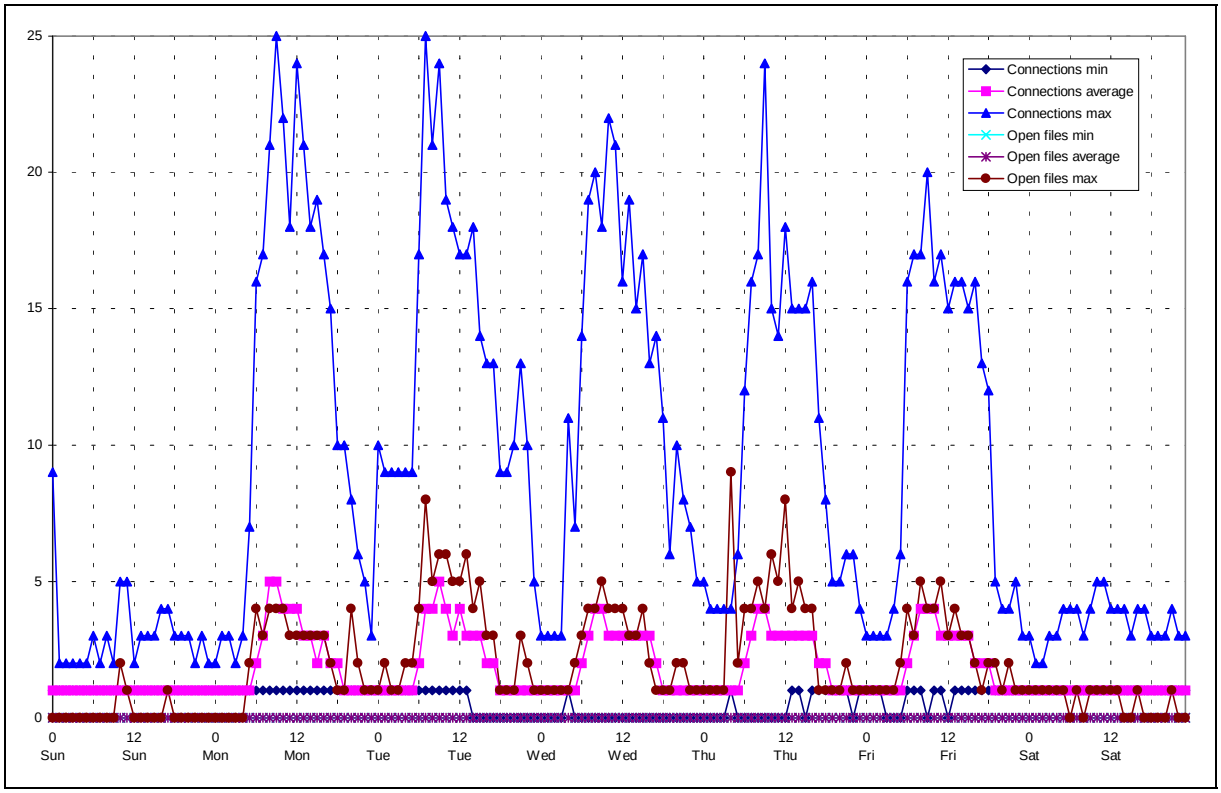
8.1.1. Server Characteristics

To better understand the usage of the different servers the number of connections and open files is used. These two figures give a good idea about the number of users who actually make use of the services of each machine.

FAW3200S - Domaincontroller

The main task of the domain controller is to handle the user's logon process, manage the security database and replicate it with the other servers, handle resource alias names and propagate them to clients in the network. A server could handle up to a thousand active logons. The actual number of up to 25 connections is not a real problem for the machine - it is not heavily loaded.

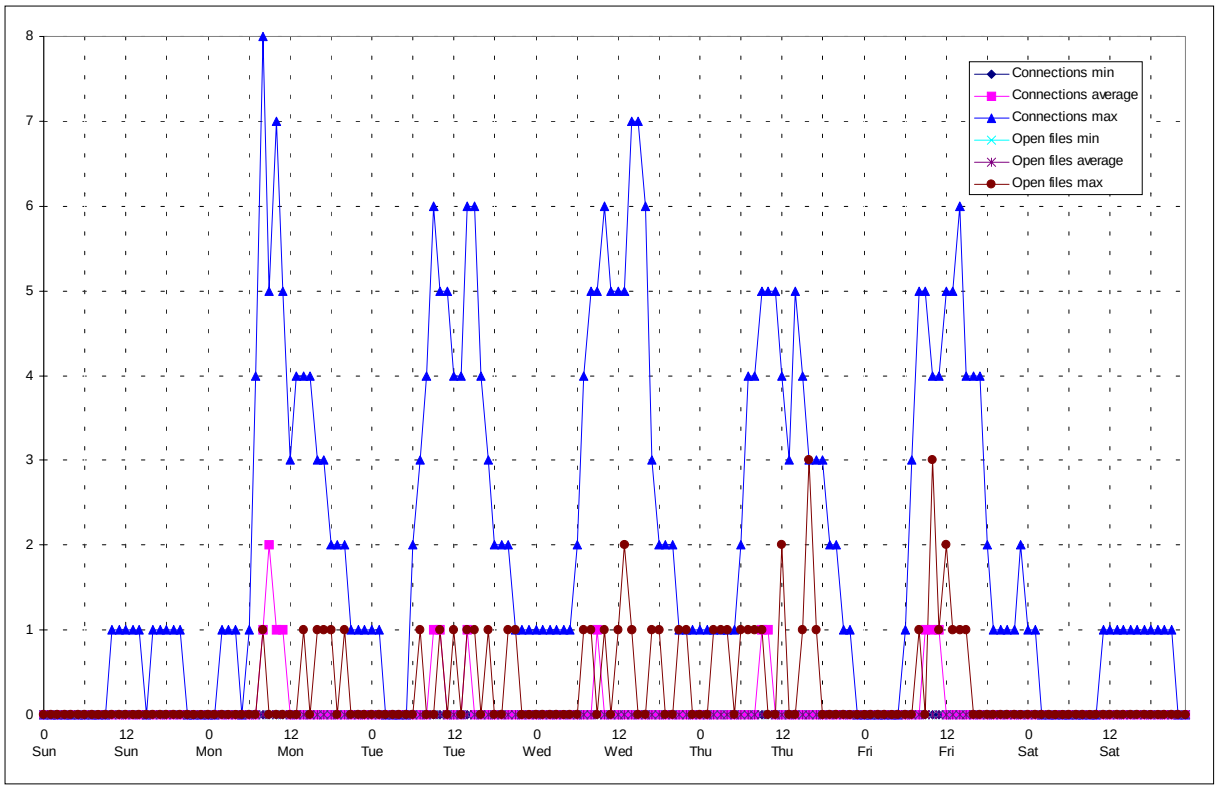
Note that a good deal of the existing load is covered by the domain backup controller. Logon-load is balanced between the main controller and the backup controller. In this scenario the main task of the backup controller is to keep the domain running in the case of a failure of the main controller. In other scenarios load balancing may be a more serious task.



Each active user has a connection to the domain controller. As the controller does not provide much more the number of connections is identical to the number of users which are handled by that machine. Normally there are no open files because the machine is not used as a file server.

FAW3210S

This machine is used for internal reasons and does not provide services to the normal user of the domain. Thus the number of connections is very low.



FAW3220S - Print Server & Backup Domain Controller

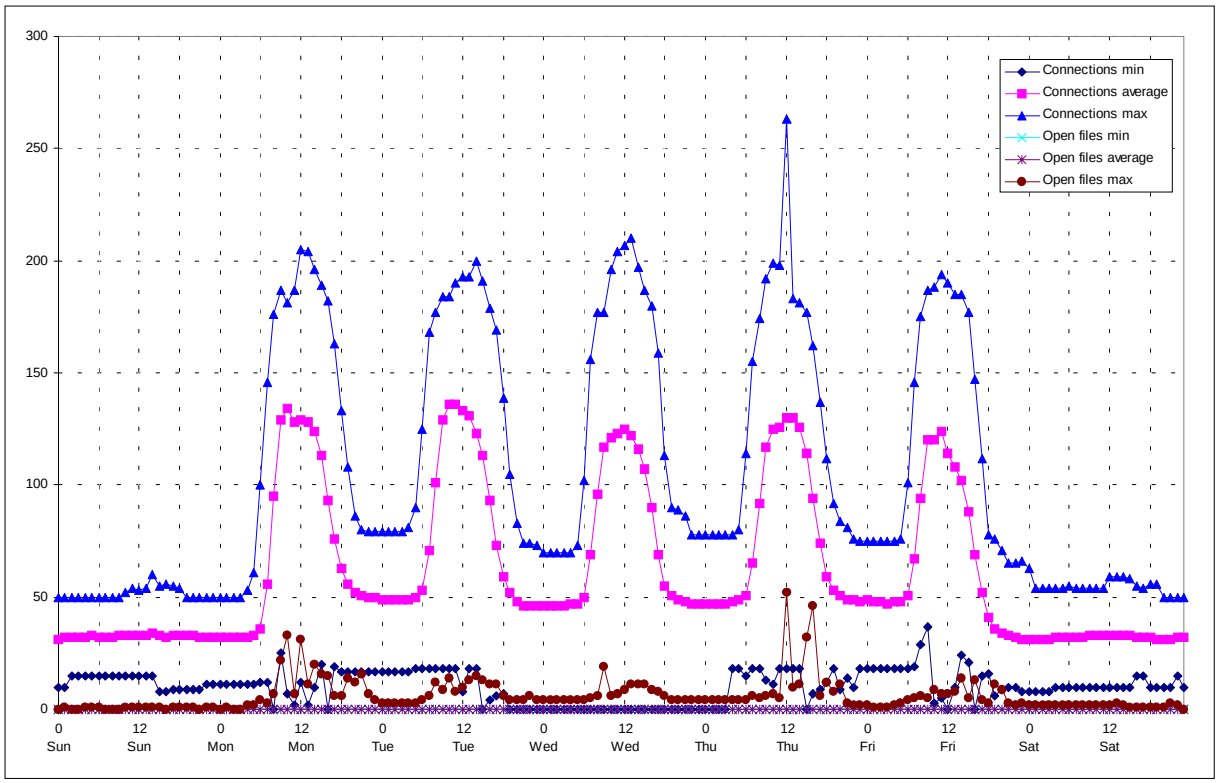
This machine is one of the most important machines in the domain. It serves two purposes: it is used as a backup domain controller, which shares the load of logon processes, and it is the only print server in the domain.

A print server has several tasks that may produce some load on the machine:

1. It receives and stores the print output from the clients.

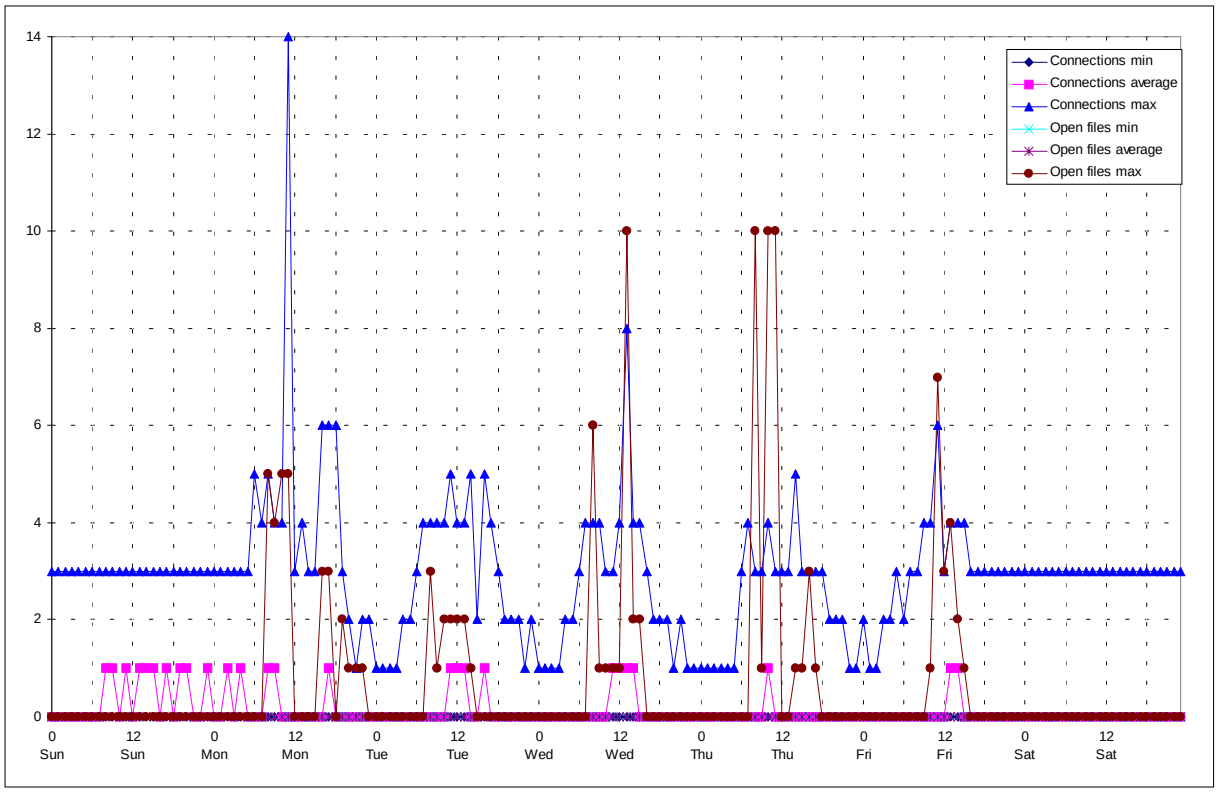
1. It manages the access to the spooler queues. Queues can be either shared or serial. The first one, typically a printer queue, can be used by several client at the same time. All output is stored temporarily. If a spool job has finished spooling it has to wait until the printer (port) is available and then the job contents is sent to the printer. That is transparent to the client.
   A serial queue - typically a fax or modem - can only be used by one client at the time. If the queue is already in use the client has to wait or it will receive a (time out) error.

1. In the case of OS/2 printer output, the server does the actual processing and calculation during the conversion to the actual printer device output. The clients only sends a standardized meta file.
   In fact most of the print output is produced by Microsoft-Windows applications. In this case the client has to perform the conversion/generation of printer specific print output. Therefore the server load from such conversion tasks is low.


300
250
200
150
100
50
0
Connections min
Connections average
Connections max
Open files min
Open files average
Open files max
0 Sun
12 Sun
0 Mon
12 Mon
0 Tue
12 Tue
0 Wed
12 Wed
0 Thu
12 Thu
0 Fri
12 Fri
0 Sat
12 Sat

FAW3240S - CD-ROM

This server houses a shared CD-ROM drive and another removable media. Today, it may sound strange, but at the time, this domain was monitored, CD-drives were still expensive and usually not part of common desktop systems. As the price of such devices dropped and the availability of CD-ROM drives in desktop and laptop PCs increased dramatically, the usage of this server decreased.

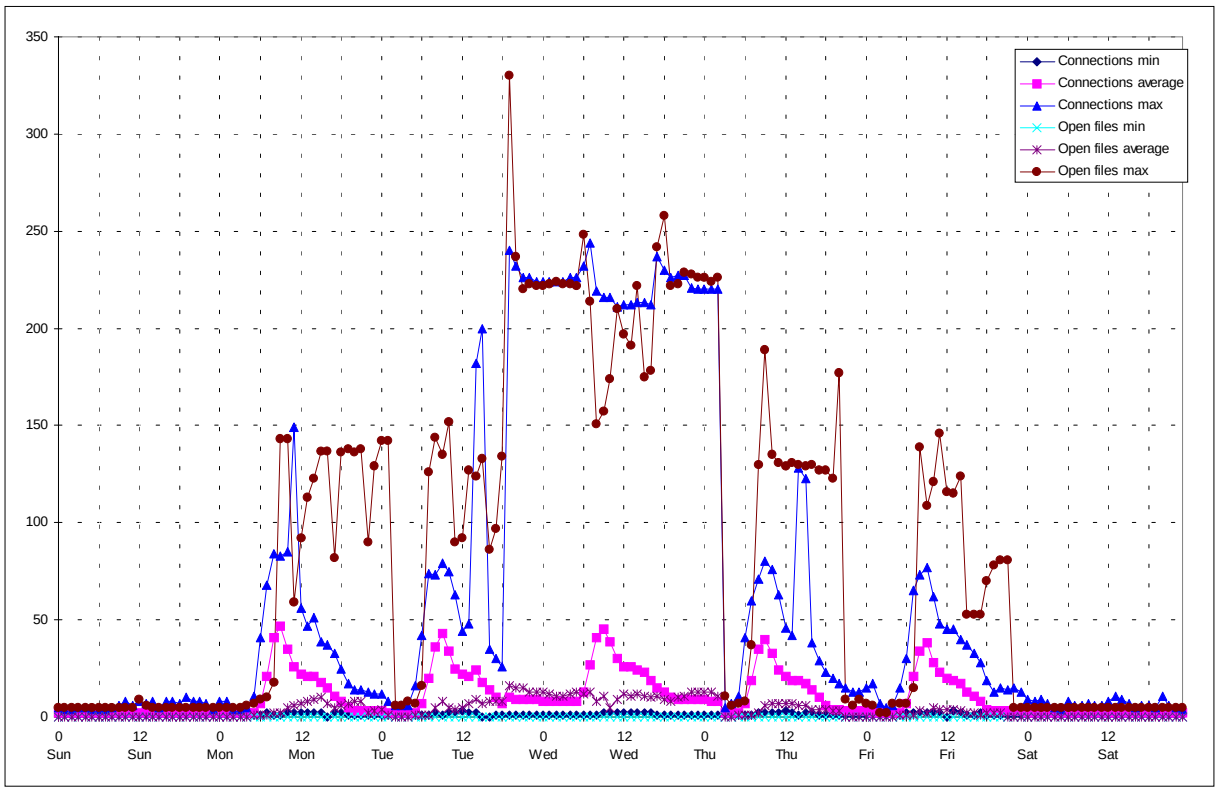


Usually only one user wants to use a certain CD. Therefore the average number of connections is either zero or one.

FAW3250S - Home Directories

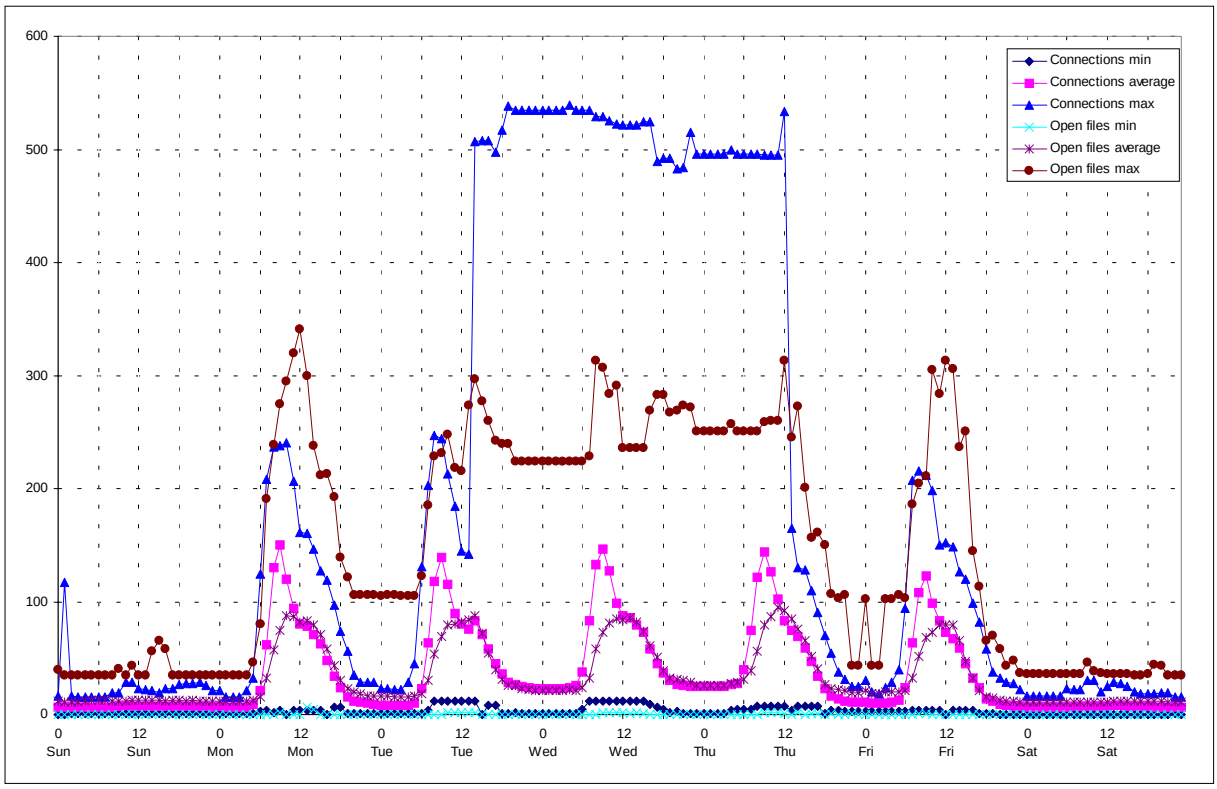
The main task of this server is to maintain home directories of LAN users and shared areas for teams. Home directories should be used for backup purposes only. Over time - with rising capacity needs - home directories where distributed over a number of servers. Team disks are usually controlled by an library management software. Therefore the files on team disks cannot be accessed directly (with several exceptions).

Looking at the picture below it can be seen that during peak hours the number of open files is high. That indicates that people work directly with their files on the server (number of open files).



FAW3260S - Database and Application Server

The server houses several DB2/2 databases and most shared applications that can be installed or used from the server. The number of connections and open files indicate a heavy usage (up to 90 user have up to 550 connections) of this machine.

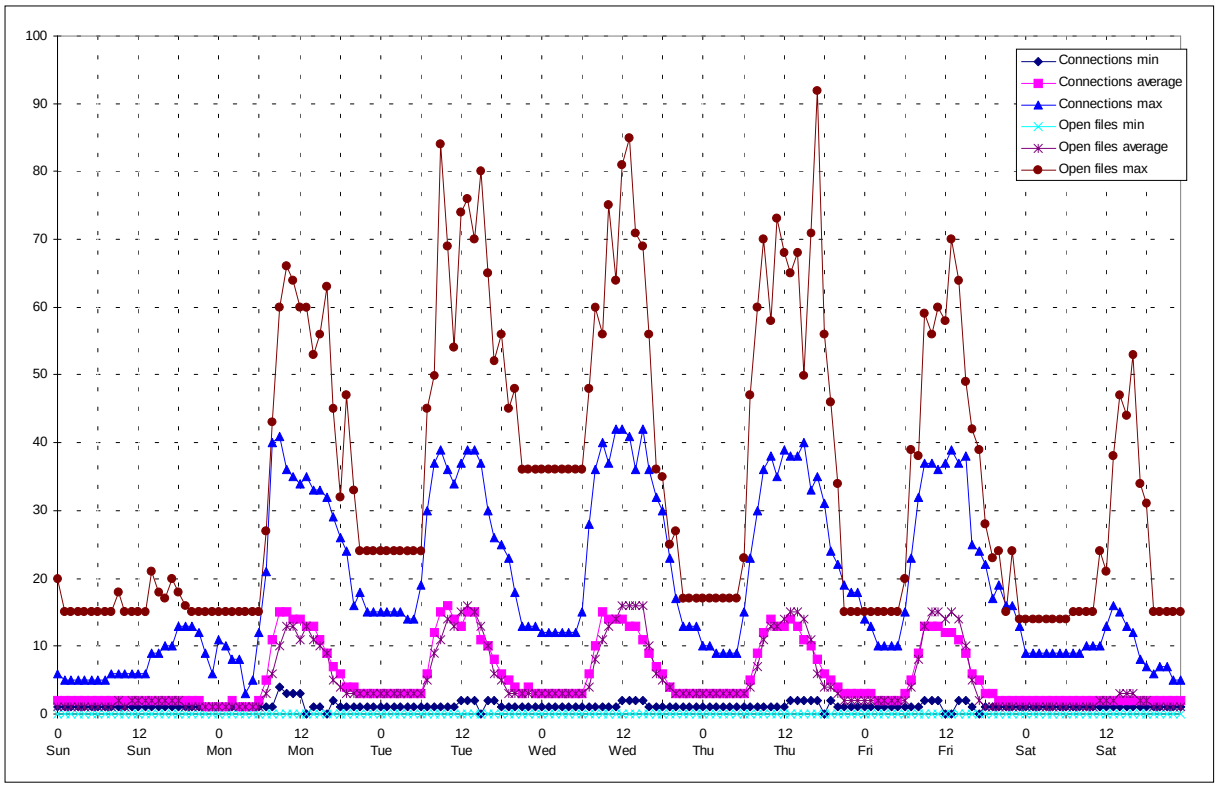


Over the time it could be noticed that the usage of the server decreased. One reason for that was the increasing number of laptops which replaced desktop machines. Application software has to be installed on the laptop and therefore fewer people are using shared applications from a server[41].

[41] That creates a lot of other problems e.g. software distribution and service which were addressed by a shared resource before.

FAW3270S - Application and Tools Server

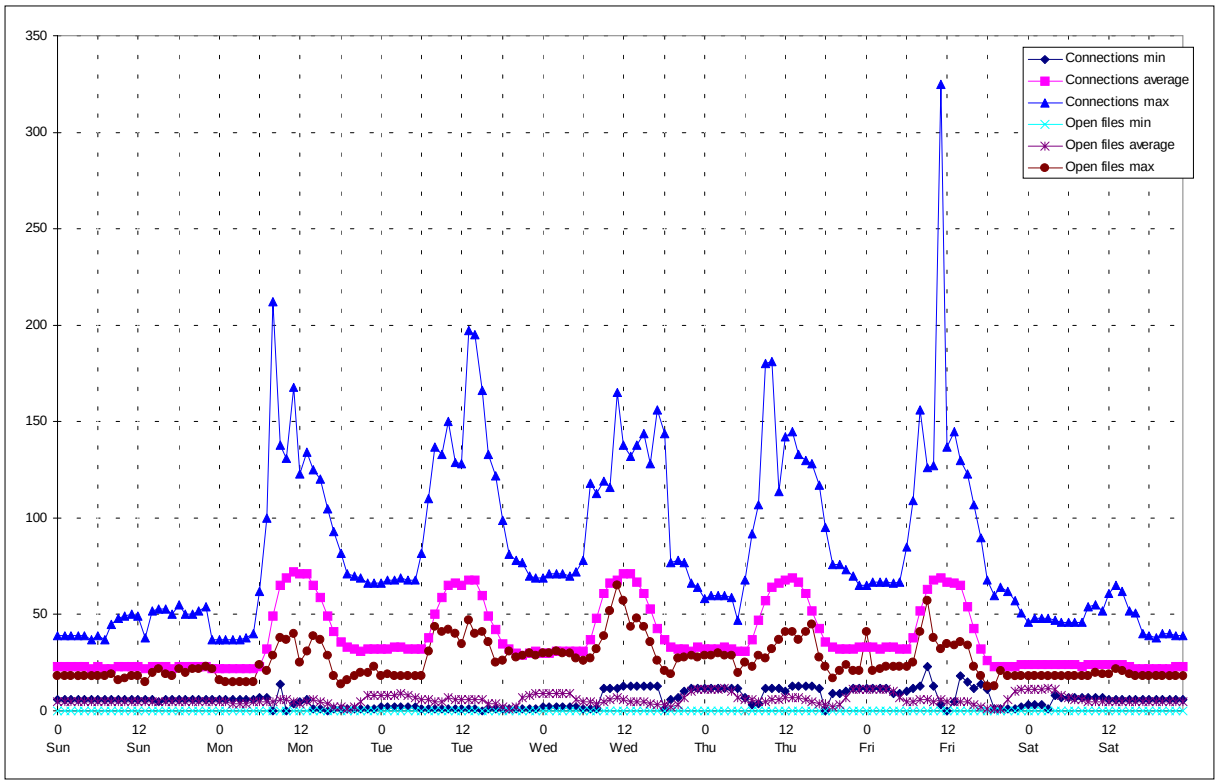
Most of the (workstation) development tools of the department were installed and maintained on the machine. The number of (workstation) developers is about 40% of the user community. Common tools stay active after their first activation and they consist of many files (programs and dynamic link libraries). That is the reason why the number of open files is rather high compared to the number of connections.



Interesting is that even during weekends many machines seem to be running and the tools stay active most of the time.

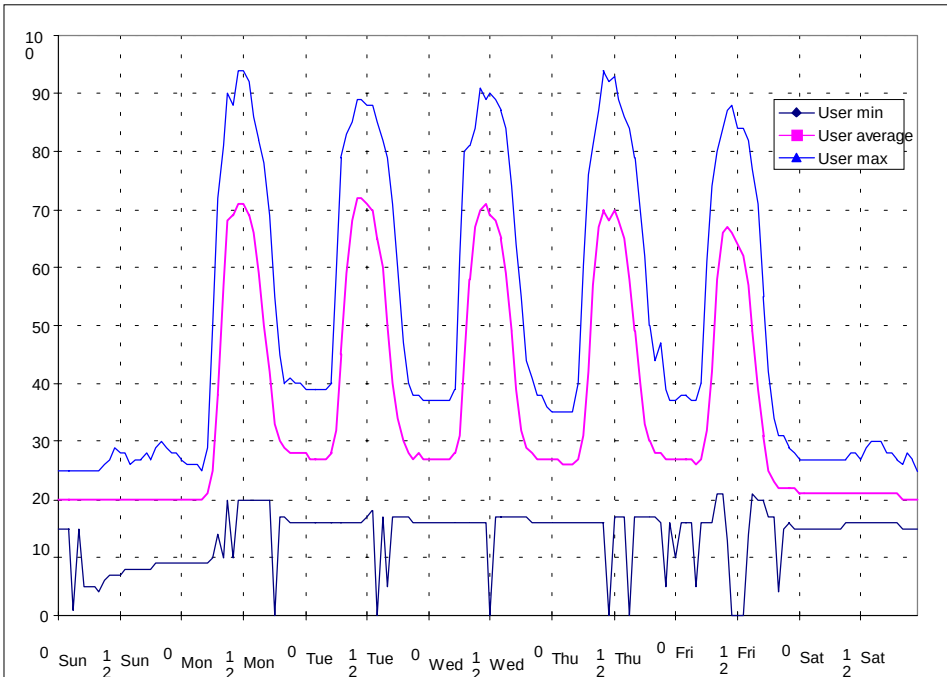
FAW3280S - Development Code & Data Server

Most of the teams use this server to store code and data of their projects. The shared resources are controlled by a library management software are not accessed directly on the server. Still a user needs a number of connections to different areas of a projects and has to transfer or receive files from staging areas. The activities in this areas are rather high (compared to the number of users).



8.1.2. Active User

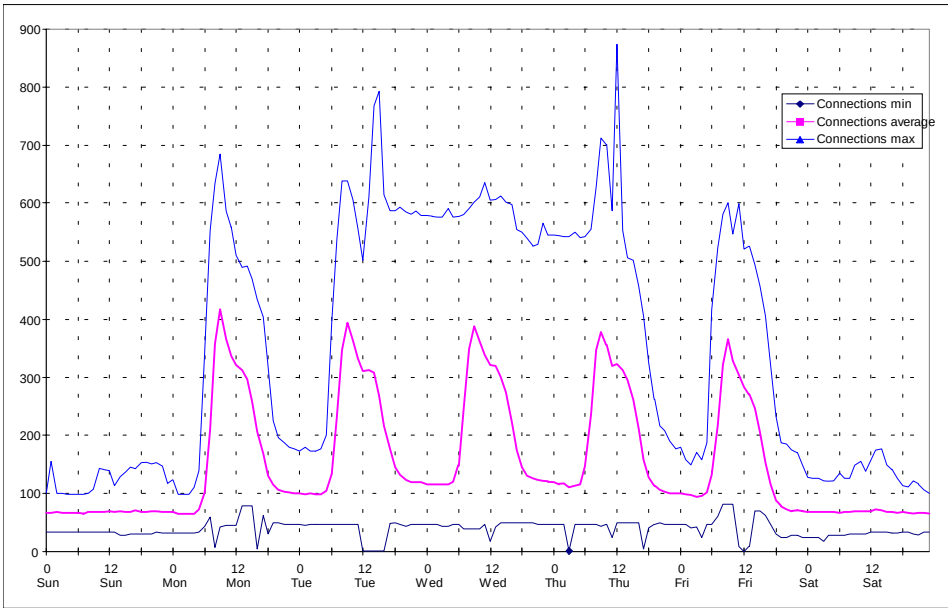
First we look at the number of active users. As mentioned above office hours drive the number of users and we see a cyclic up and down of users. A number of users is logged on all the time. We have to take into account that there is a *functional userid* active on each server. The difference between the ten userids on the servers and the number of users we see at night and on weekends is equal to the number of client workstations which are not turned off. We can derive that during the week about 20 of 80 user do not turn of their machines while over the weekend only 10 clients stay active.



Note the several "long weekends", that start at Thursday, and that are rather common in Austria (up to five times per year) influence the average week slightly.

8.1.3. Active Connections

The number of active connections to resources on the servers of the domain correlates to the number of users.

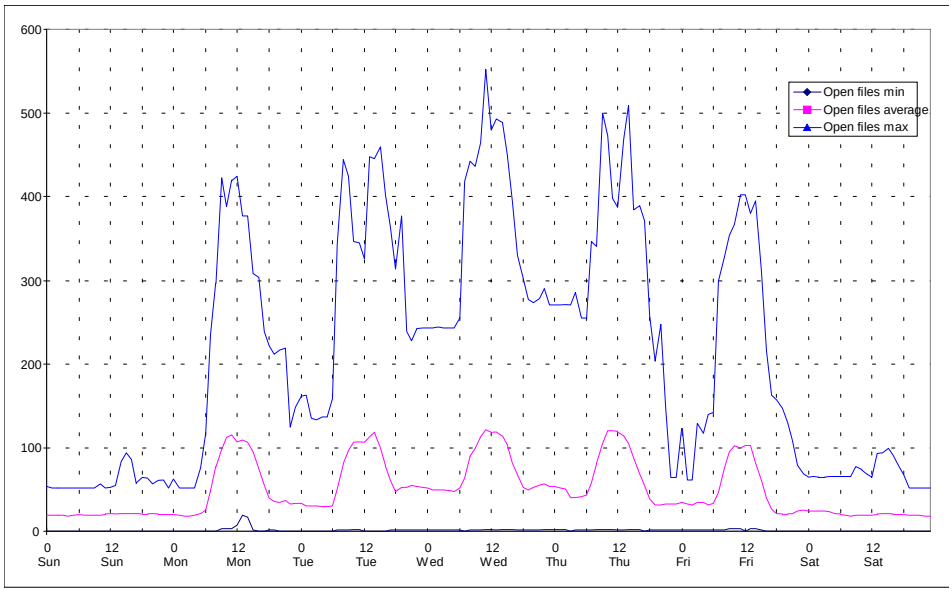


Several things can be noticed:

- the number of connections reach their peak at about ten o'clock in the morning (while the number of users top at about twelve o'clock)
- the next sharp decline is about two o'clock p.m.; that is because part time worker leave about that time
- there is a rather high value of maximum connections between Tuesday and Thursdays
- each user has about five connections

8.1.4. Open Files

The average number of files correlates with the number of active users and we can see that on average each user has one open file only. That means that many users do not use available application software from a server but install it on their local hard disk.

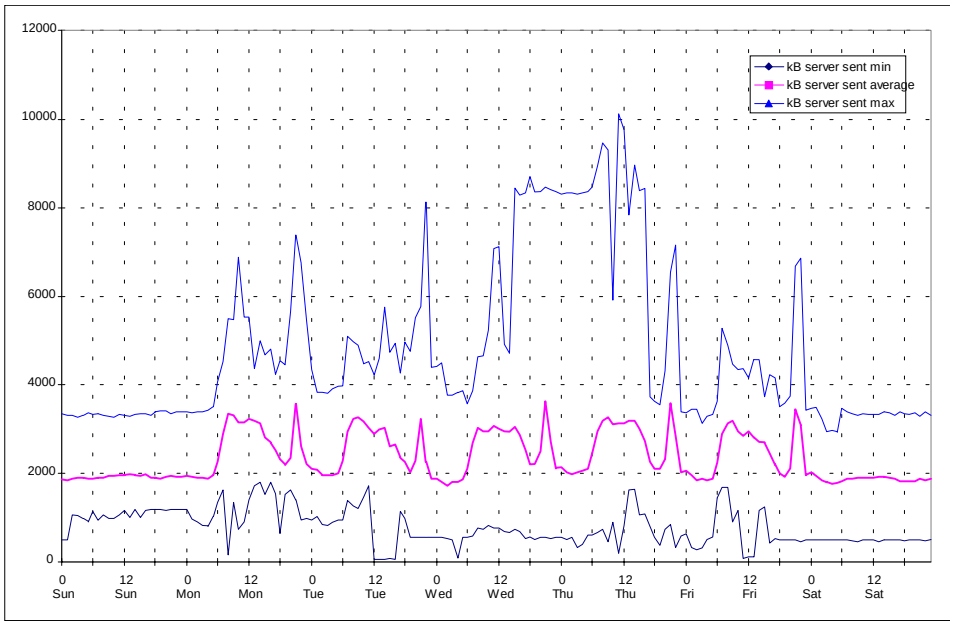


The maximum line correlates to the maximum line of active connections. It is an indication for the activity of those users which actively use server resources. Very often, there is no file open during an hour. The reason for that are holidays when nobody comes to works and an automatic disconnect - that is the server frees up resources which were not used for a long time. Unfortunately that has a strong influence on the overall average. Without the influence of holidays the number of open files per user is higher.

The following snapshot of a day's detail view is typical for the number of open files: many files are opened in the morning shortly after the connections were established. They remain open as long the connection persists. That behavior is typical for application software that is used from a server resource because many users do not really end an application but hide the window and keep it ready for its next usage.

8.1.5. Data Sent to Servers

The next figures shows how much kB of data are sent to all servers per hour. We can see that up to 10 MB per hour are sent with many peaks during the night. Note, that it can be seen that once a week during the night an automated backup and restore procedures generate a higher data volumes than during normal office hours.

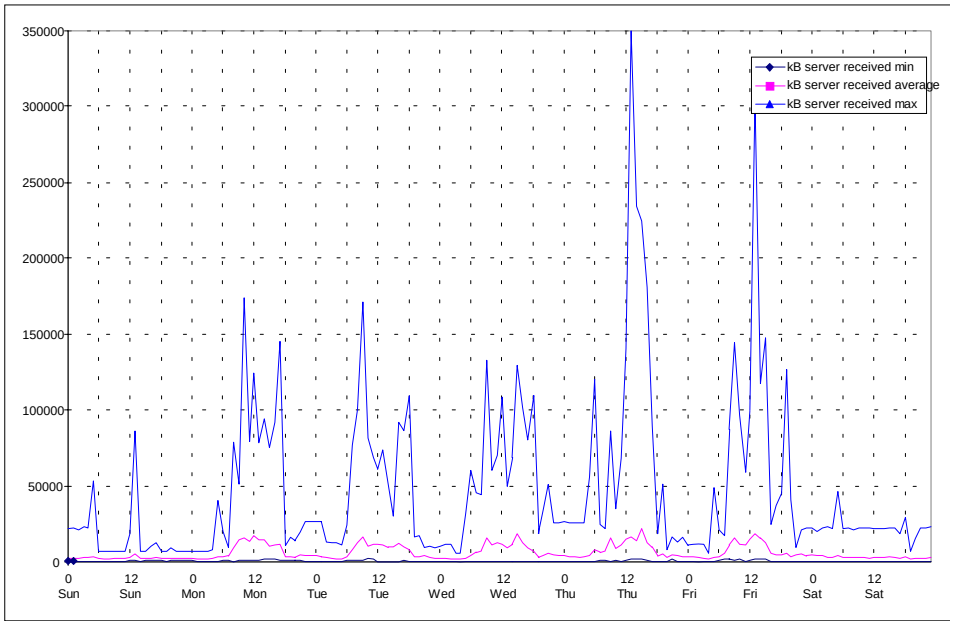


For the average value we still can notice a correlation with the number of users. Remarkable is that on average there is a traffic of about 2 MB per hour during off-time hours. That seems to be the amount of information the server has to exchange even without direct user activities.

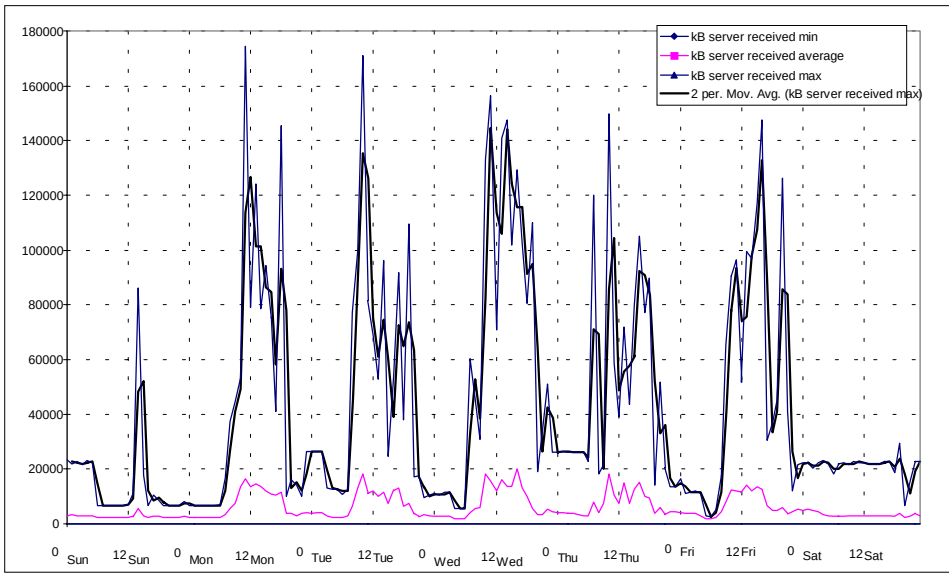
In addition we notice that automated procedures easily produce higher traffic volume than normal user operation does. We have observed the same system behavior in other circumstances (see “other observations”).

8.1.6. Data Received From Servers

For this information two figures taken from data samples at different times. The first one contains two peaks which did disappear over time. This example illustrates the high volatility of the data which are directly influenced by user activities.



The second picture represents the normal load on the servers. On average peak load occurs short before noon. This observation correlates with the fact that the number of logged on users rises until 11:30. Between 11:00 and 12:00 most users get on-line and do some work that generates load on the server



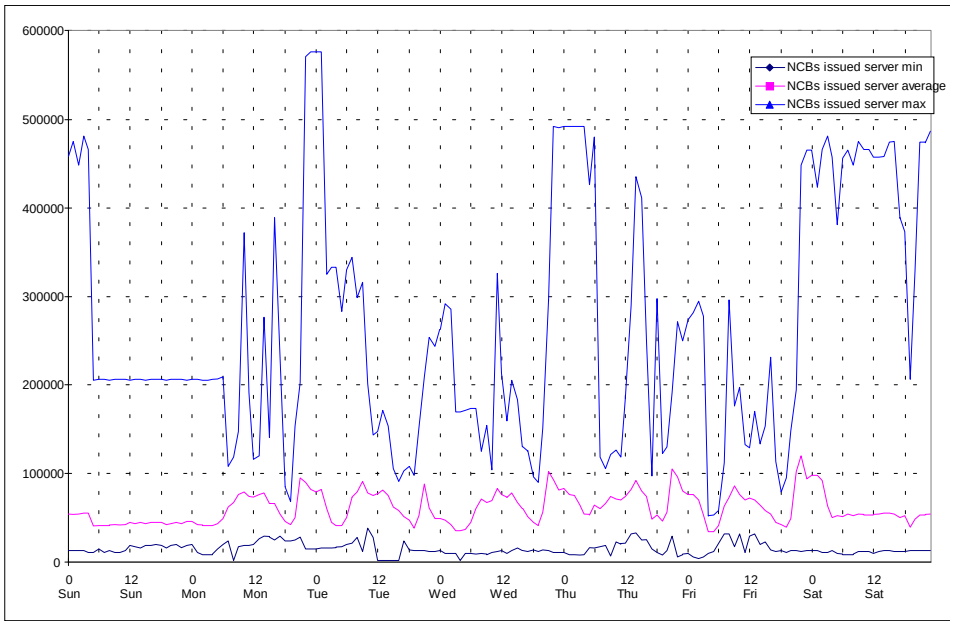
Part of that load consists simply of loading applications and data from the servers after the client computers are turned on. I measured peak loads near 18 MB per hour.

In a related “experiment” it was tried to start an application on hundredths of clients at the same time. The application was about 10 MB in size. Each client had to download the application (which was written in Smalltalk) in order to start execution. The result was that the network was down while the server load was low. The reason for this was that the capacity of the LAN (4 MB Token-Ring) was the bottleneck. The long-time measurements and experiments like this showed that the

power of today's servers are far beyond the capacity of the LAN infrastructure when it comes to volumes of file data.

8.1.7. Network Control Blocks Issued By Servers

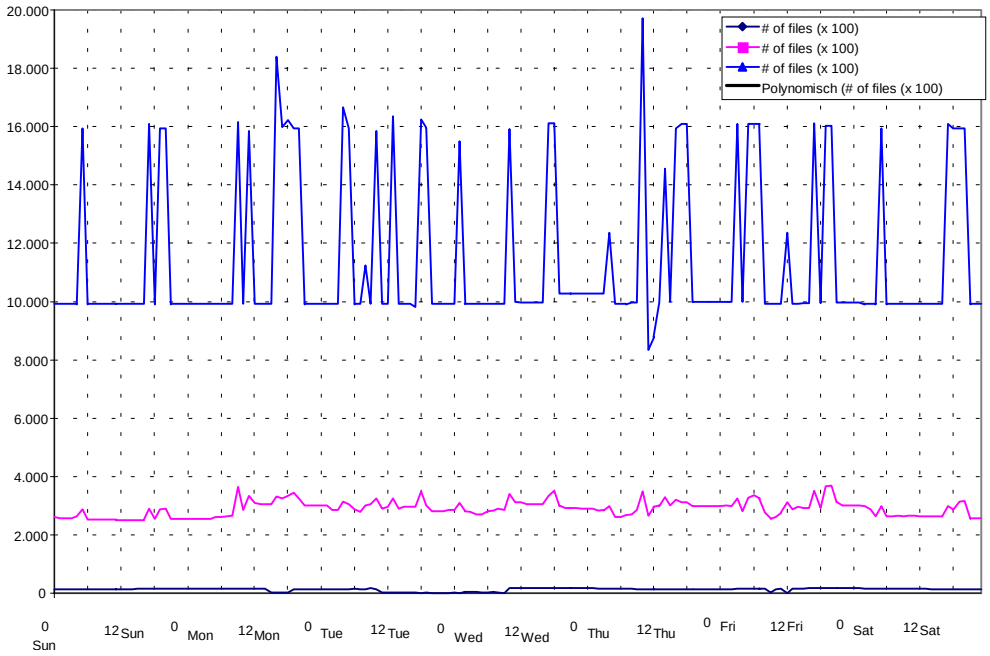
The LAN server software uses NETBIOS as transport protocol over the network. NCBs (network control blocks) are the means to work with NETBIOS. Applications reserve and issue NCBs to communicate over the network. Therefore the number of NCBs issued is a measure for all network activities (while it does not indicate the amount of data which is moved over the wire).



As we already saw at the graph for data transfer a lot of activity is going on at noon and during the night. Only early in the morning and in the evening network activities decrease significantly.

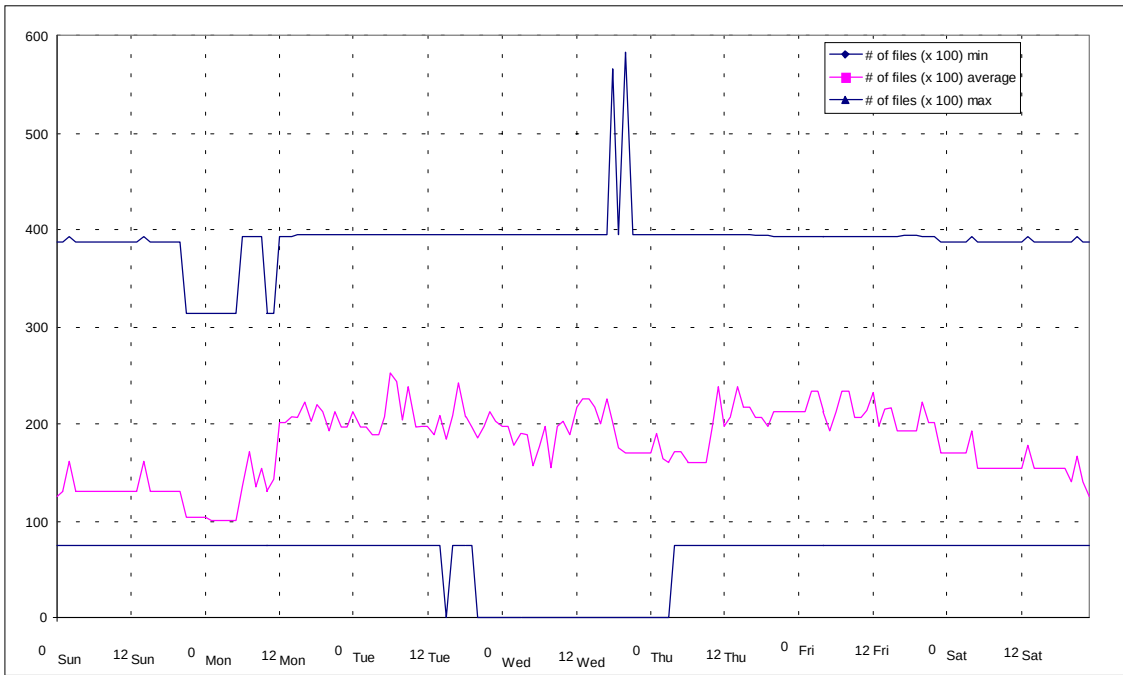
8.1.8. Number Of Files

Next we have a look at the file systems of the servers. This information gives some insights in the usage of the file sharing service. In the average over one year we see that the number of files vary in the range from 2.500 to 4.000 files during the week. At the middle of the week there are more files while this number goes down towards and during the week end. One possible reason for this is the heavy use of temporary disk space which is cleaned every night and is not reused during week ends. Knowledgeable users are able to reserve temporary diskspace for a longer time.



What we can see is that the bandwidth of possible values vary (from few thousand to 2.000.000 files). The lower margin comes from events where a server is setup or recovered after a disk failure. That happens from time to time and restore may take several days (due to the slow backup tape devices which were in use at that time). The minimum values do not tell us much about the real use of the machines. The maximum numbers do. These are the high watermarks for what the servers have to take.

As an example for details I add the same graph for the server with most of the home directories:



On average it hold between 10.000 and 25.000 files with a high watermark of 60.000.

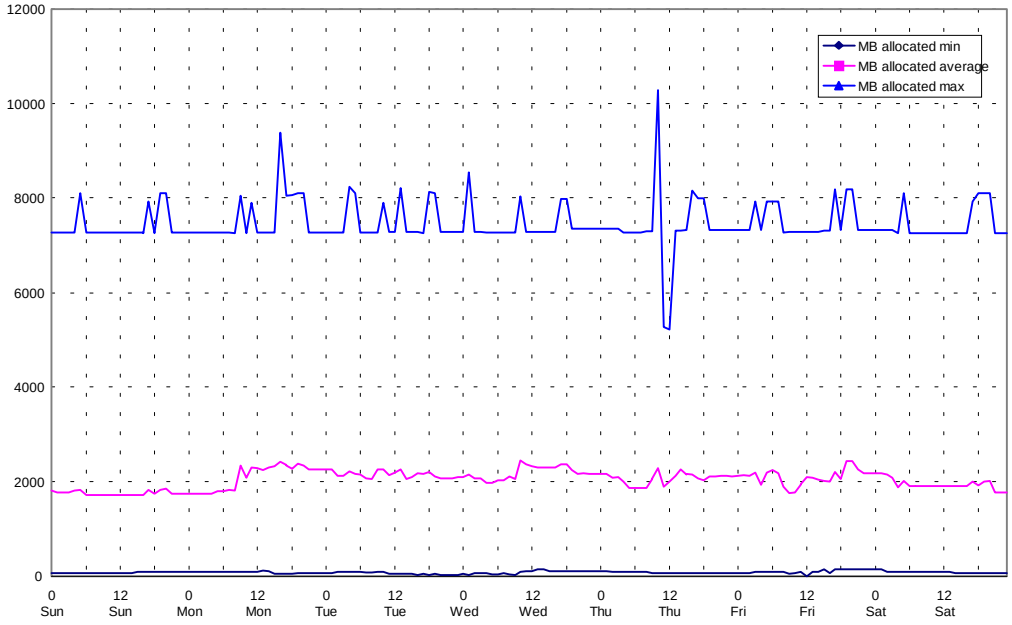
8.1.9. Number Of Directories

In the graph below we can see that even the number of directories vary between 2.000 and 3.000. That means up to 1.000 directory entries are created and removed during the week.

8.1.10. Allocated Disk Space

Two things are remarkable for this picture:

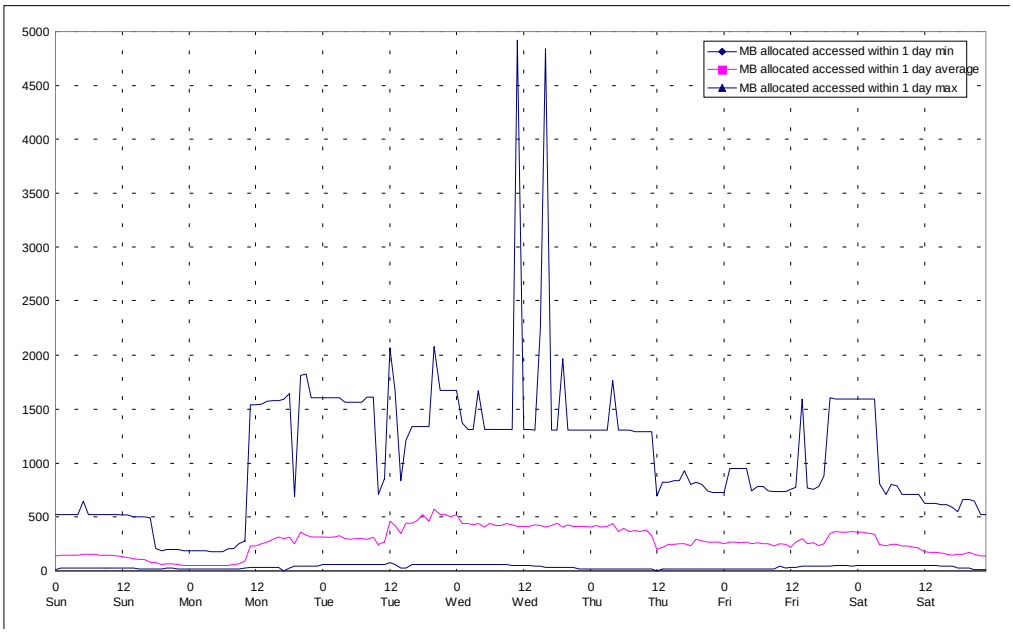
- The amount of allocated disk space does not show much variation over the week.
- The ups and downs in the graph below are nearly identical with the line for the number of directories. The main reason for this may be that the servers are not used to store much temporary information. The local PC disk is used for that purpose. Cleanup on the server disk is done directory-wise.



8.1.11. Access To Diskspace

This chapter shows the access pattern and the aging of information on the server disks.

Diskspace Accessed During The Last Day



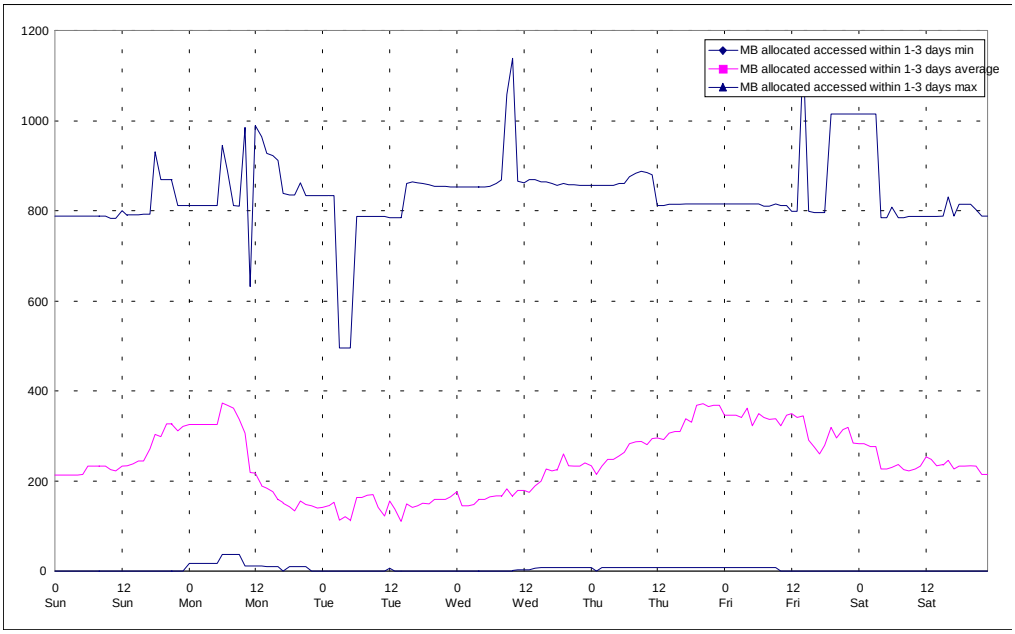
The OS/2 HPFS file system maintain information of "last access" versus "last update". Therefore it is possible to distinguish pure read and read/update transactions on the file system. During the weekend hardly any file is used (only the operating system itself). During the week up to 300 to 500 MB (on average) are actually used - compared to 2.000 MB that are allocated.

While I thought that it may be interesting to monitor and analyze that kind of data in reality the usefulness of the information can be rather low: Due to regular backup activities most of the files on the disks are "accessed" (as can be seen in the picture above for the maximum line). Obviously a backup tool was used which is not part of the operating system and which acts as a normal process that reads (and eventually writes) files. Because the backup task did not work well the actual access pattern of this domain could be extracted and is represented by the average line in the picture above.

Comparing the number of accessed space with the total space one can understand that everything except the system partitions is backuped.

Diskspace Accessed During The Last Three Days

Over the (extended) "weekend" (starting with Wednesday morning until Monday morning) the amount of information grows which was used recently but now starts to age out. Note that there are two reasons that disk space leaves that "category" - and thus disappears from the graph: first, the information is used again; second, it remains unused and is counted for the next category (below).

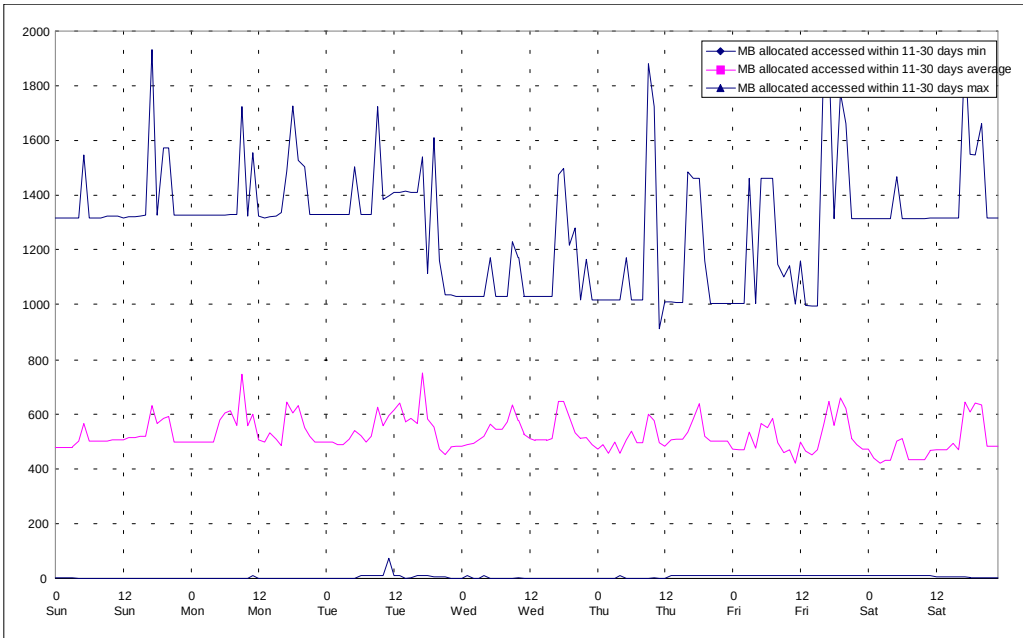


Diskspace Accessed During The Last 4 To 10 Days

A peak during Tuesday indicates those files which were used in the week before but not over the weekend and on Monday. The sharp decrease on Tuesday noon may be an indication for the fact that about one quarter of that information is used again during Tuesday.

Diskspace Allocated Within the Last 11 To 30 Days

The amount of information which is not used for a month is relatively constant. This information is not longer used and ages out. About 500 MB out of 2.000 MB remain in this category.

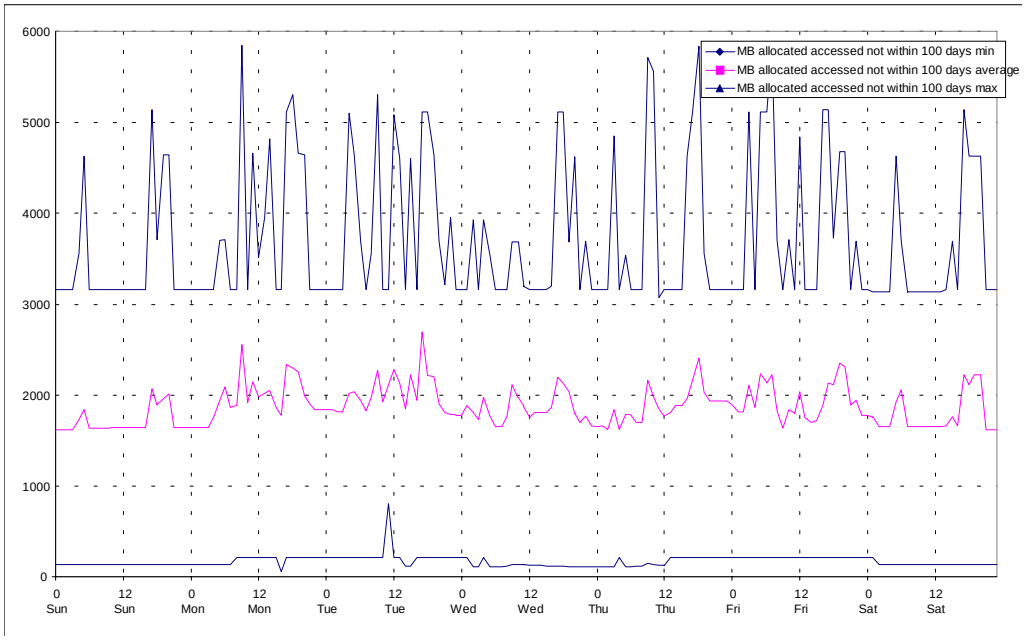


Diskspace Accessed Within The Last 31 To 100 Days

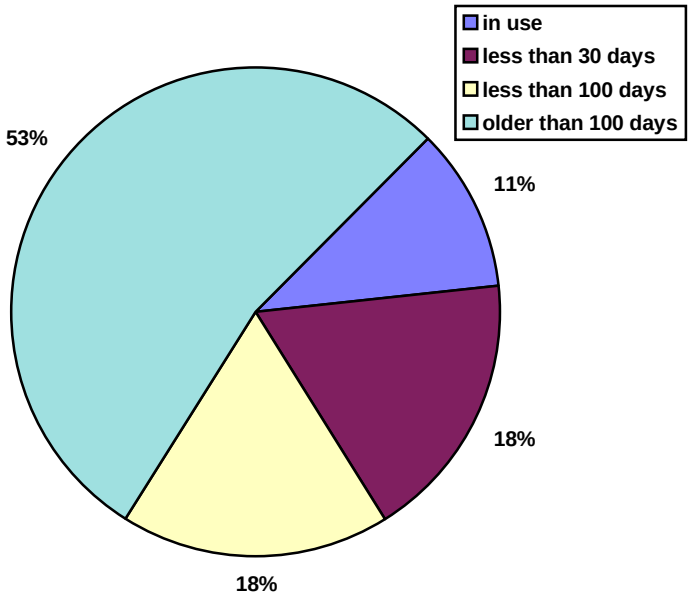
The same can be said for information which is not used for more than one month. About 600 MB out of 2.000 MB fall into this category.

Diskspace Not Accessed Within 100 Days

This graph shows the amount of "dead code" on the server systems. About 1.500 out of 2.000 MB seem to be of no further use.



Summary


in use
less than 30 days
less than 100 days
older than 100 days
53%
11%
18%
18%

8.1.12. Cache Efficiency

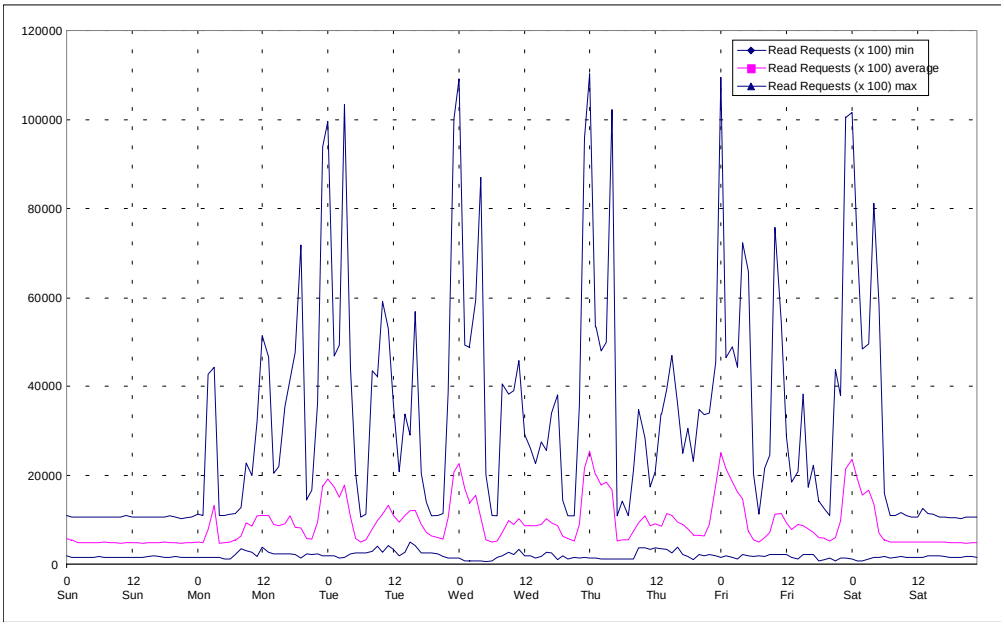
The HPFS386 file system makes available some information about the efficiency of the file system and the volume of access to the file system.

Summary

Most of the information on the server disks are read-only. Therefore there is much more read activity than write access.

Read Requests

Due to backup activities during the night we can see more read requests during the night than during working hours. As mentioned before automated procedures create much more load on the server (and the file system, in this case) than interactive work of many individual users.

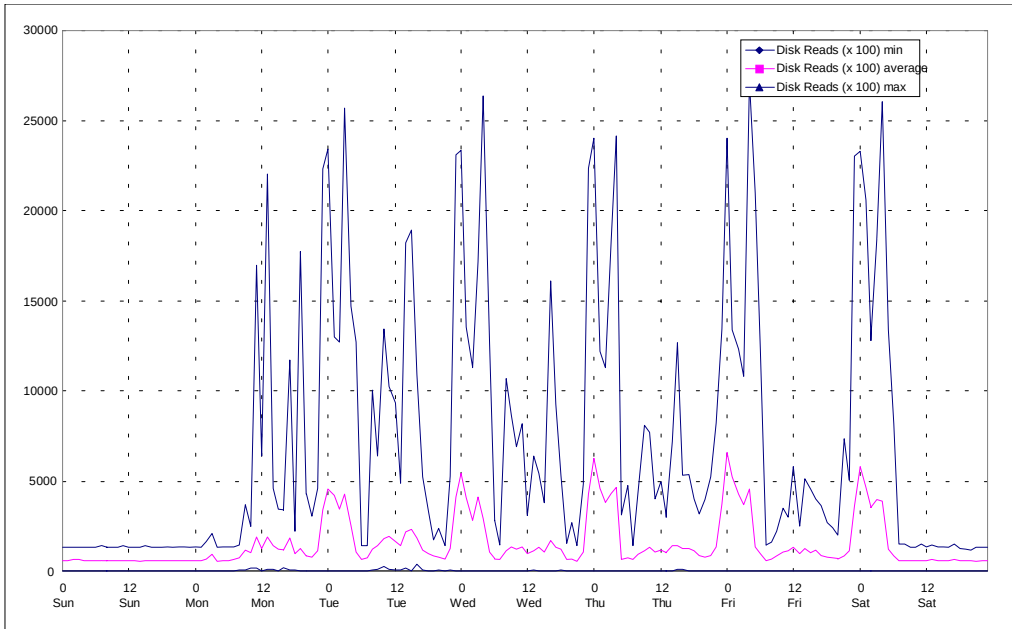


Note, that on average there are about 10.000 read requests per hour during office hours and about 20.000 read requests during backup activities in the night. About 5.000 requests are send during times of no user activity.

Disk Read Access

The number of read requests which actually trigger a (physical) read operation on a disk have a very similar pattern compared to the figure above. During the night 5.000 out of 20.000 read requests are performed on a disk. That is about a quarter of all read requests. During the day this ratio is better as there is nearly no increase in disk read operations compared to the "idle" state.

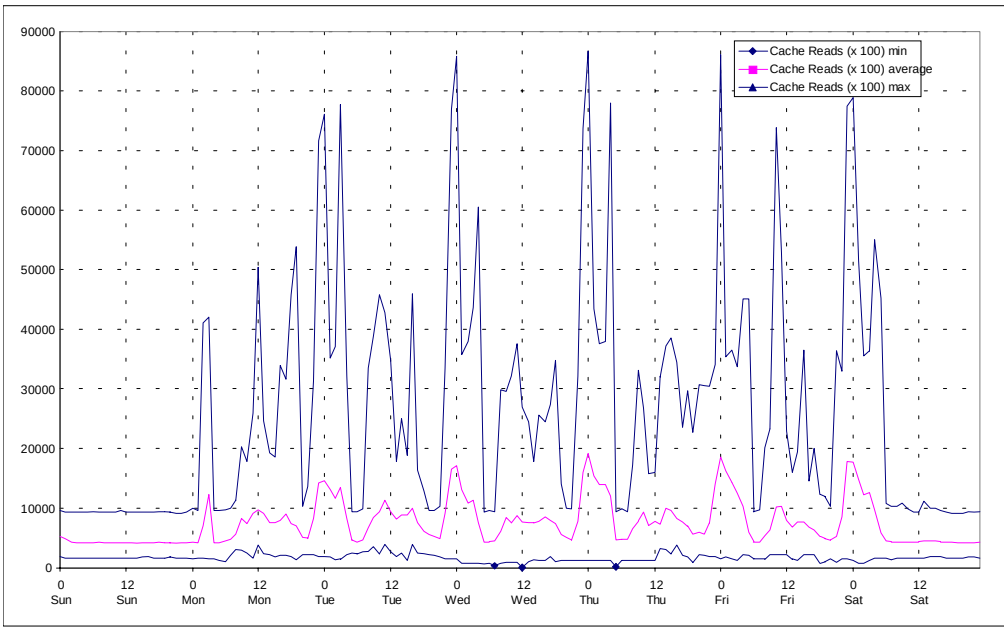
Many user access the same information (program files and data) and therefore the cache can be used more efficiently. The backup operation with its sequential access to a large amount of data seems to be a less optimal way to use this file system.



The cache reduces the actual disk reads by a factor of four.

Cache Reads

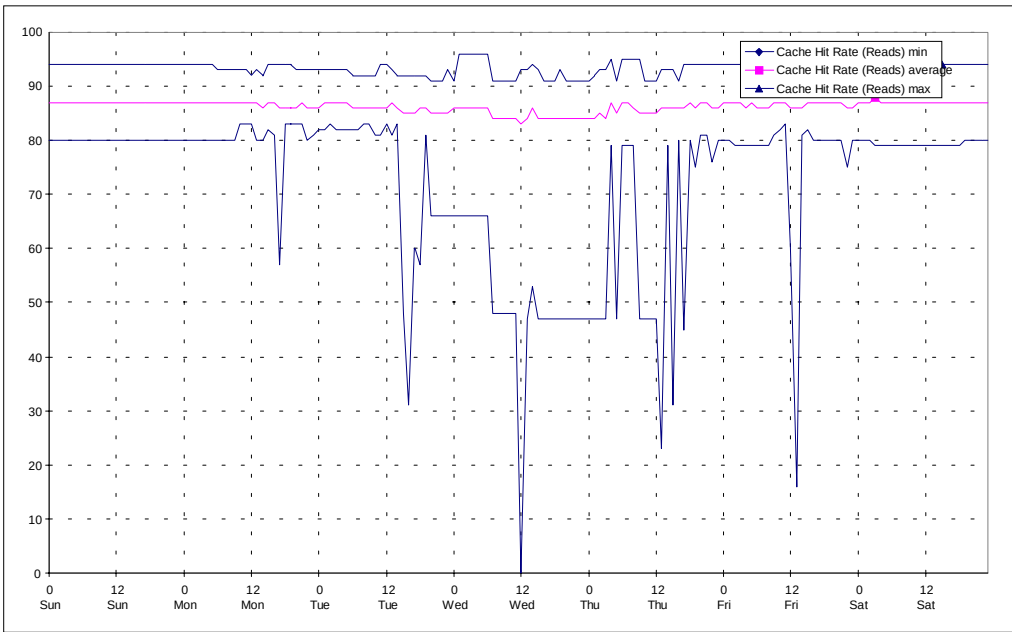
Of course, the rest of read requests are directly handled by the cache.



The required information may be in the cache due to another read request or because of the "read-ahead" feature which can use the time, the client/server system is busy with working on the data provided by the last read request, to read the next block of the file from disk.

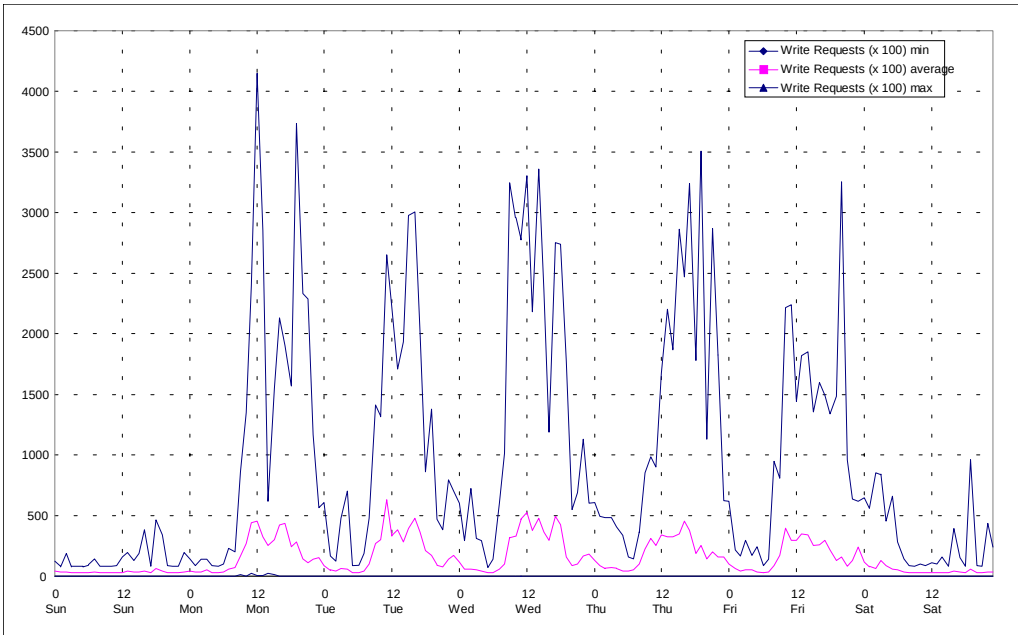
Cache Reads Hit Rate

On average the overall efficiency of the cache was about 85%. Most of the time it is not below 80%.



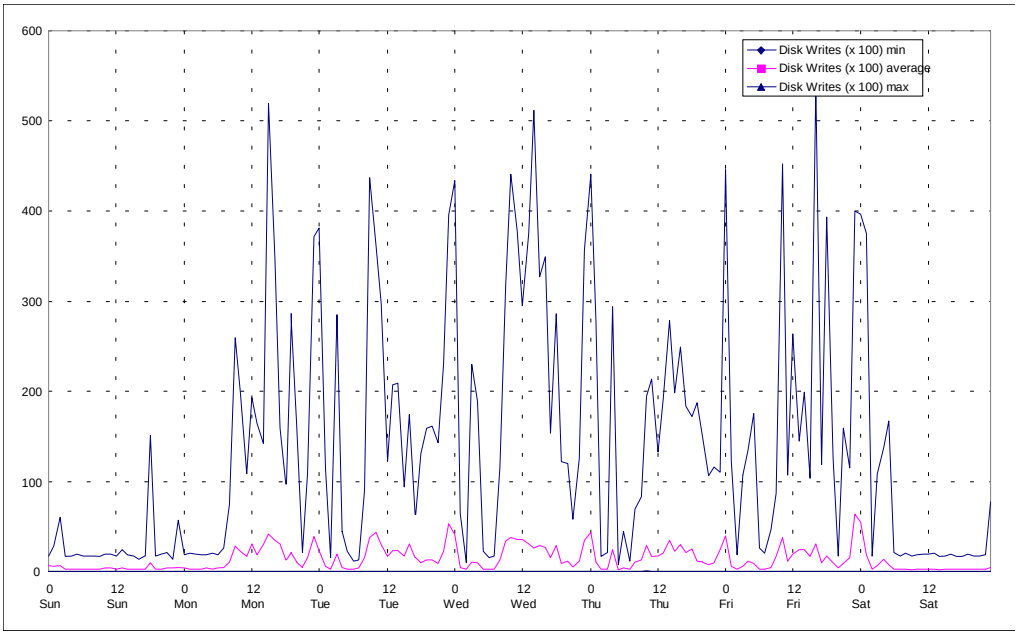
Write Requests

In contrast to the read operations there are not many write requests and there is no activity during the night (backup does not generate write requests). Idle activities do not generate write requests either. On average there are up to 500 write requests per hour.



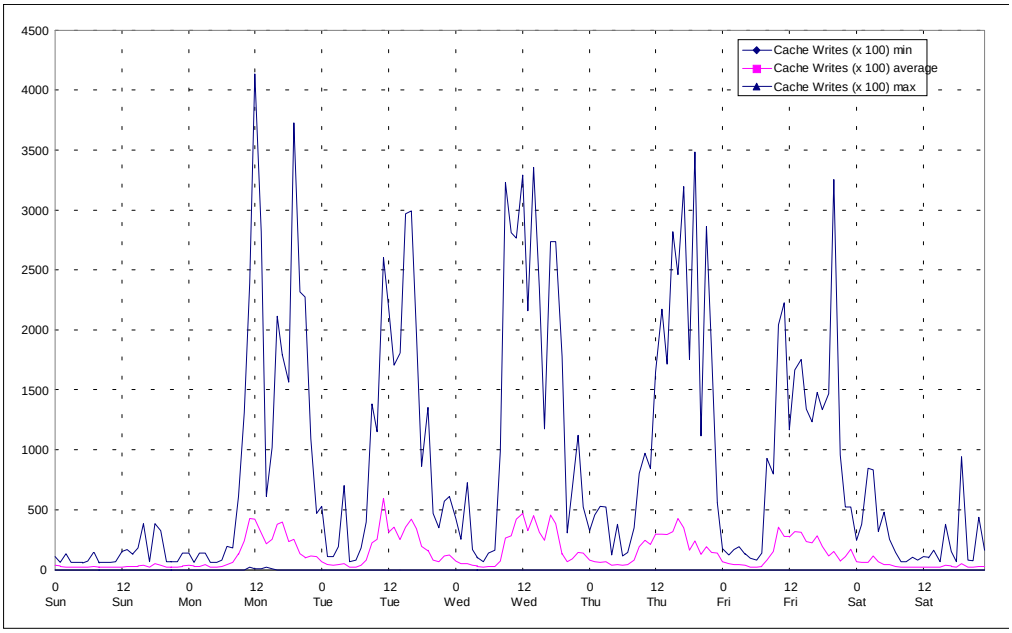
Disk Write Access

Out of 500 requests not even 50 trigger actual disk access. The ration between requests and disk operations with 1:10 is much better than for read requests (partially due to the low number of requests).



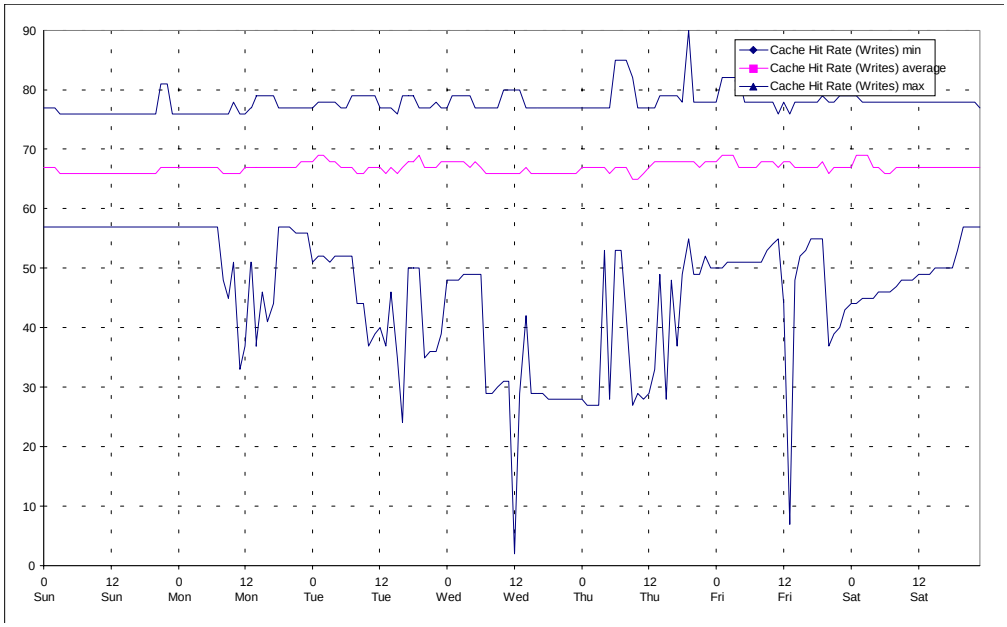
Cache Writes

Most of the requests can be handled by the cache.



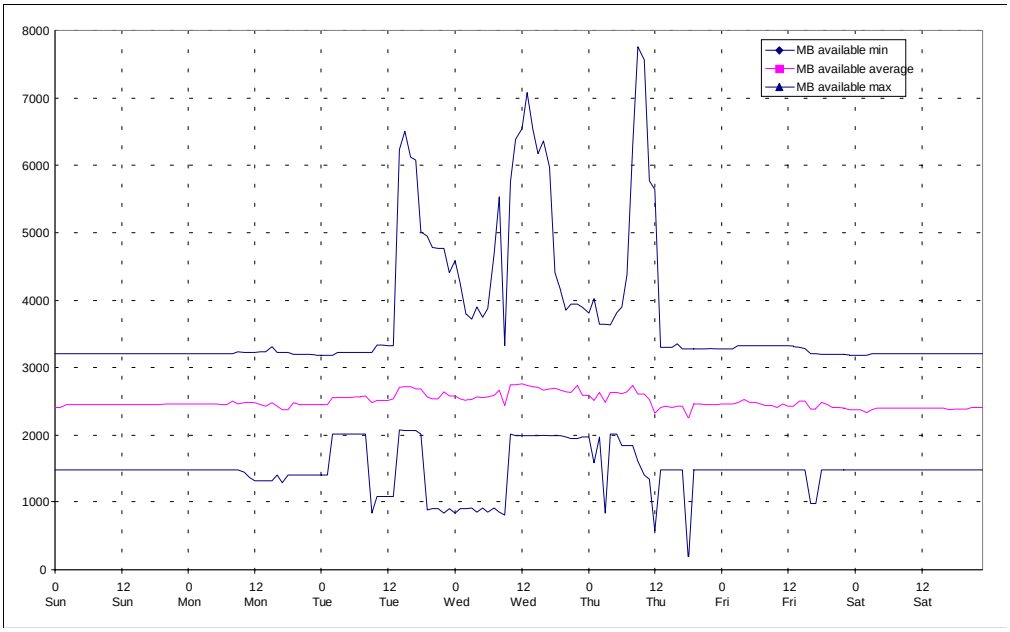
Cache Writes Hit Rate

Although most of the requests can be handled by the cache the file system reports its efficiency with an average value of about 78% (which would be less efficient than for read requests).



8.1.13. Diskspace Available On Servers

As there is not much variation in the amount of allocated disk space there is not much activity in the amount of available disk space.



8.1.14. NETBIOS

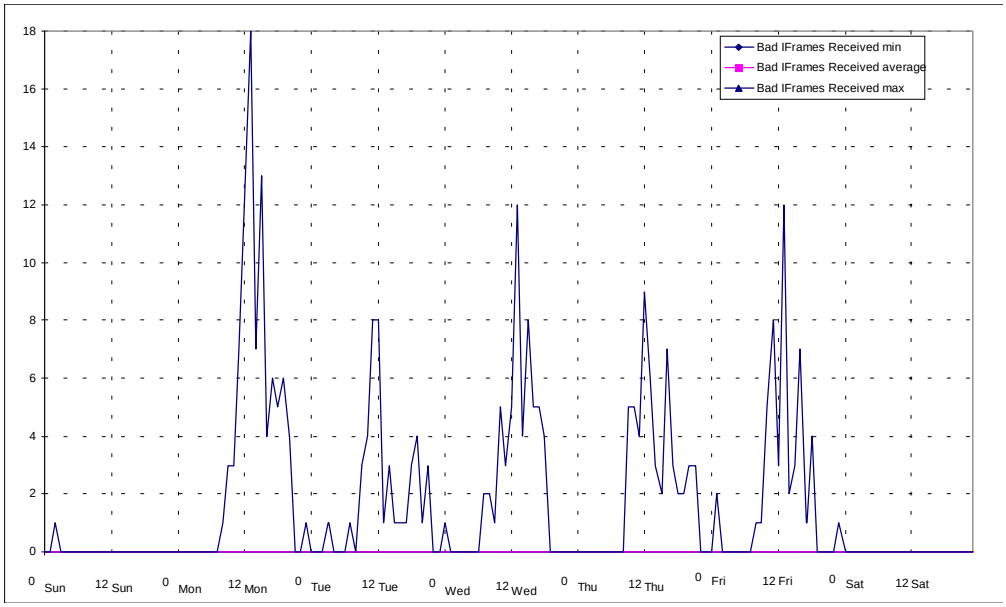
See [2] for more information on NETBIOS specific attributes.

NETBIOS - Frames Received

No Information about frames was recorded.

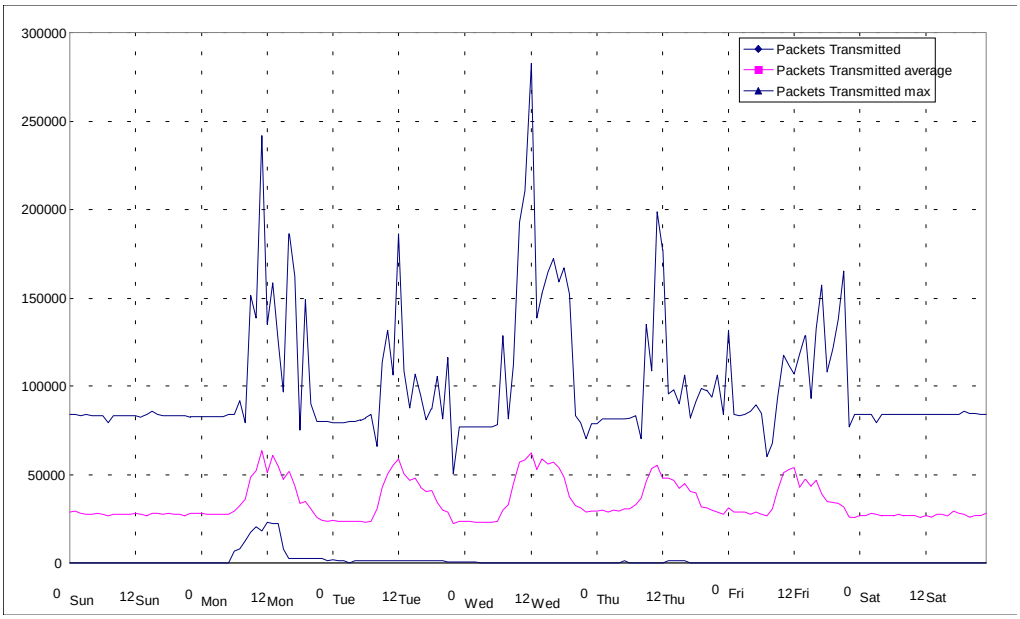
NETBIOS - Bad Iframes Received

This measurement is an indication of network problems. Bad frames may decrease the network performance. Here the average number of bad frames is zero. Thus, most of the time there are no problems.



NETBIOS - Packets Transmitted

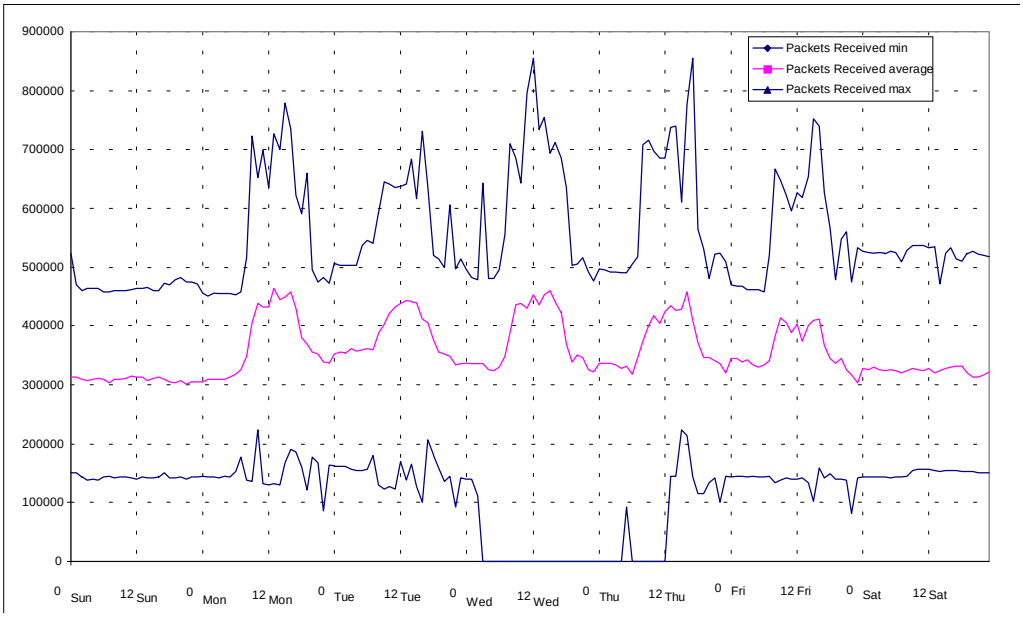
The number of packets which are transmitted or received are an indicator for the activities on the network. We can see that there is a lot of background activity during non-working hours of about 30.000 packets per hour. Depending on the number of users this number increases to about 60.000 packets.



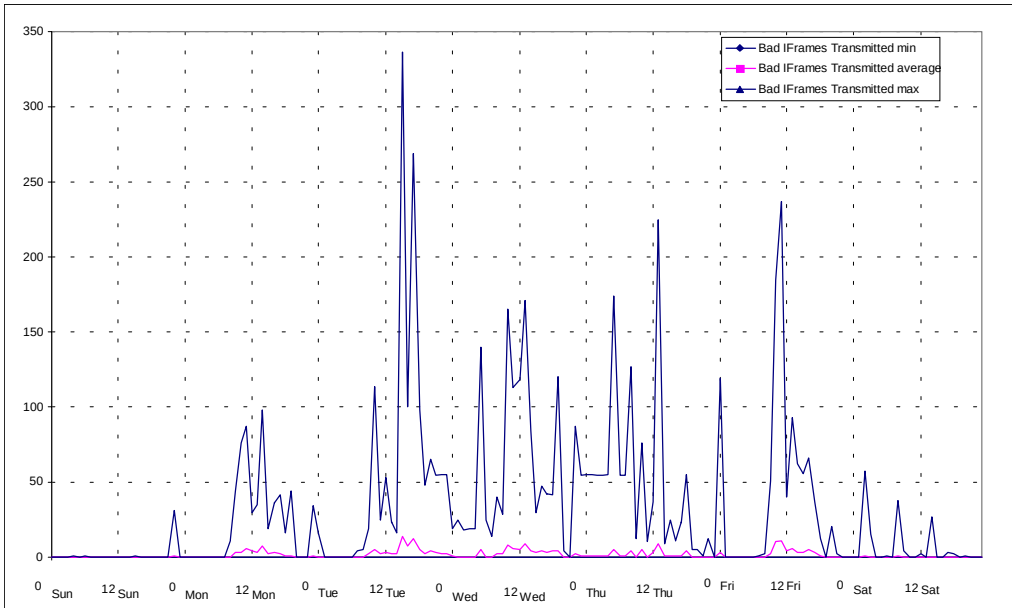
Again we notice the user behavior we have seen before: a sharp rise of activities till noon and a slowly decreasing line in the afternoon. Note that the number of packets, that are transmitted by servers, is much lower than the number of packets which are received by them.

NETBIOS - Packets Received

We see a similar graph for the number of received packets.

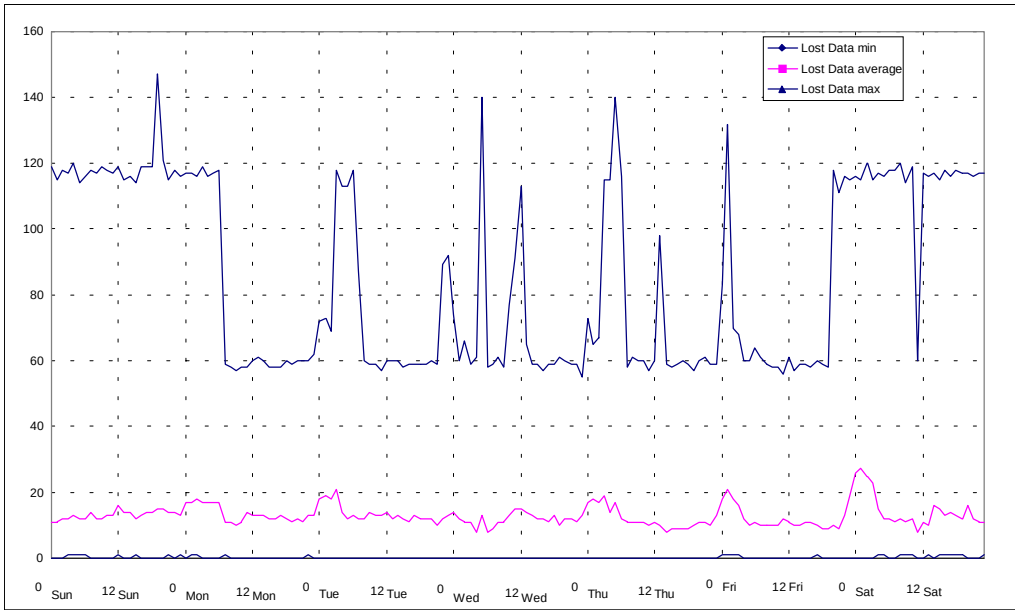


NETBIOS - Bad IFrames Transmitted



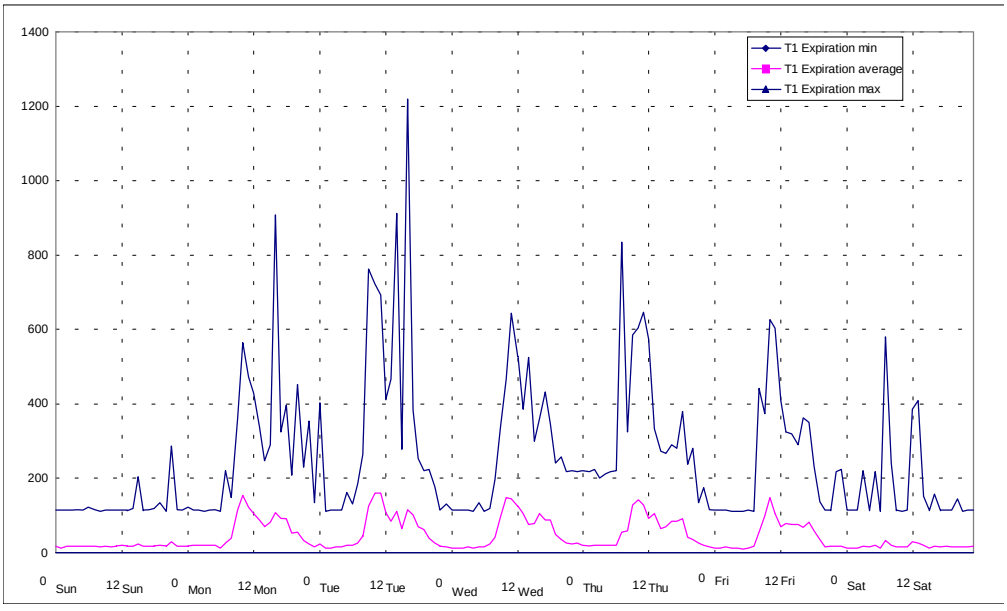
NETBIOS - Lost Data

There are not many errors due to network or application problems.



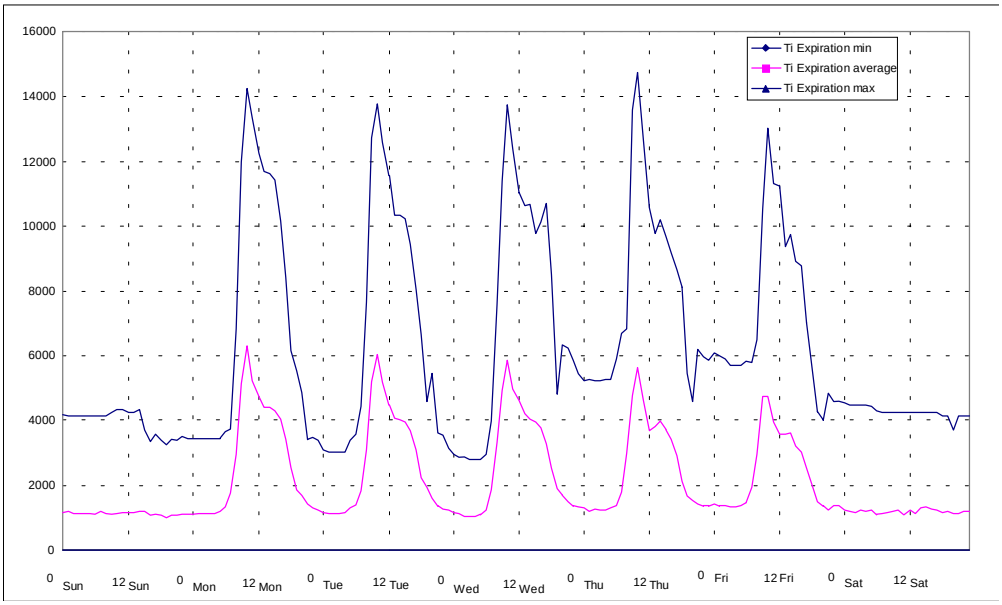
NETBIOS - T1 Expirations

The T1 timer checks whether a request for setup or resume of a connection is serviced within time. The number of expirations correlates to the activities on the network.



NETBIOS - Ti Expirations

The Ti timer checks the activity on an active connection. If it expires due to lack of activity the network software will check the connection. The number of expirations correlate with the number of sessions (and connections) to the servers. This contributes to the network traffic and generates some of the traffic during idle times (non working hours).



NETBIOS - Free NCBs

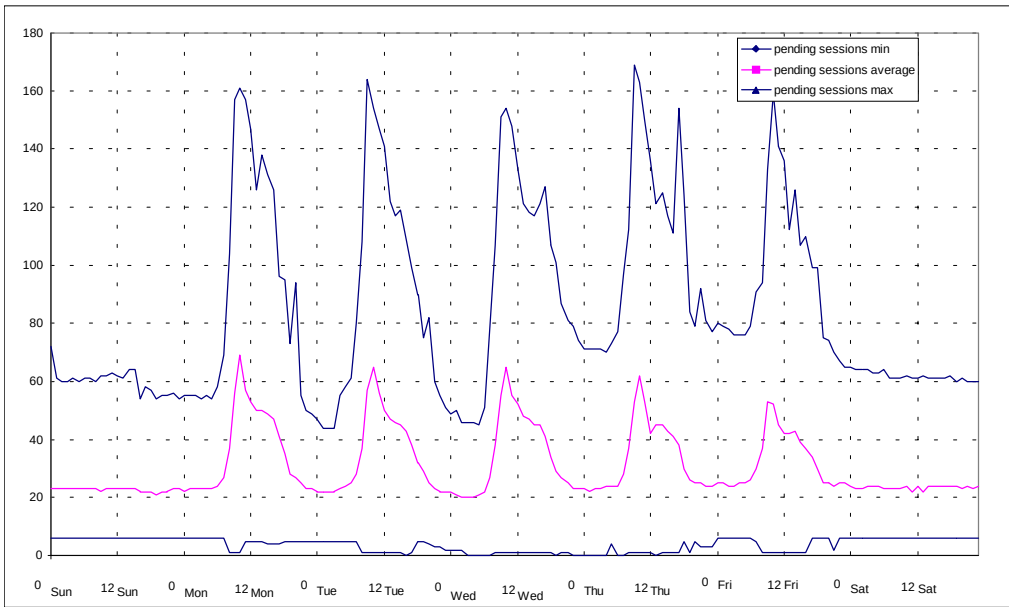
A network control block (NCB) is one of the key resources provided by a token-ring adapter. All network related activities relay on the access to one or more NCBs. As users connect to the servers network resources are allocated.

NETBIOS - busy conditions

There were no busy conditions detected.

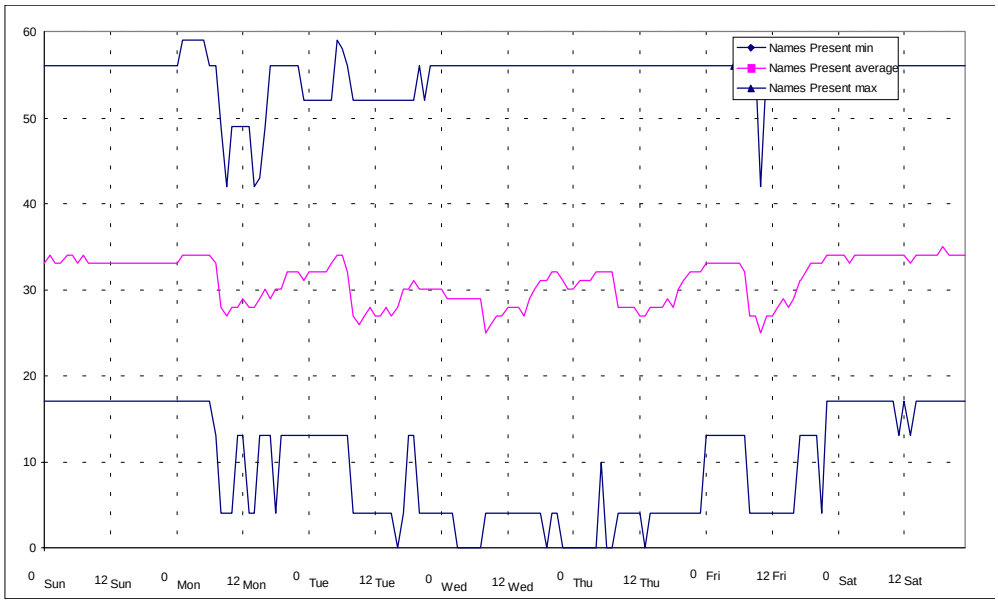
NETBIOS - Pending Sessions

The number of pending sessions clearly depends on the number of open connections.



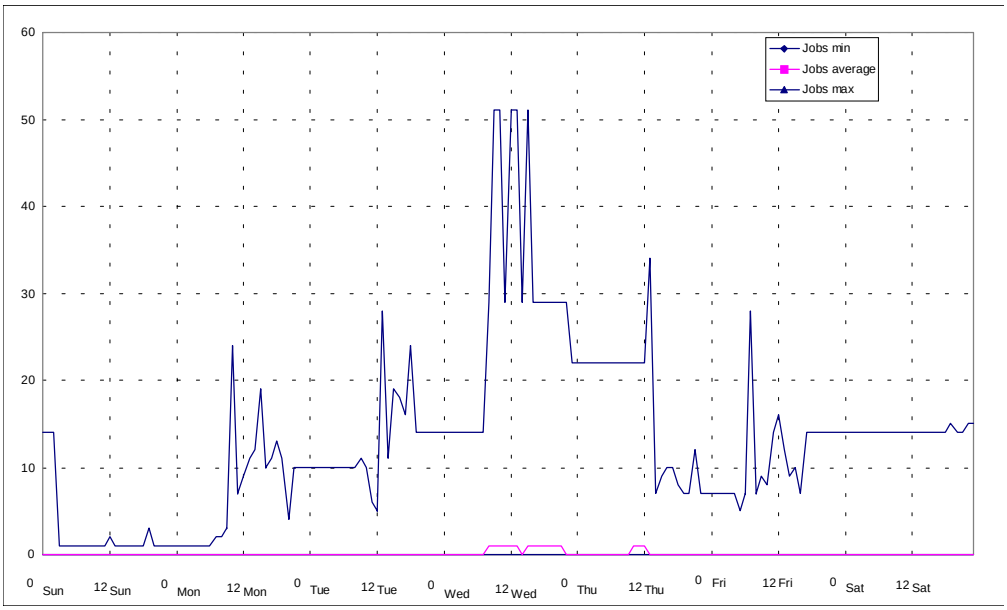
NETBIOS - Names Present

A name is another important resource. It is the name of a NETBIOS application on the local host. An application that is going to open a session on the adapter has to register one (or more) names.



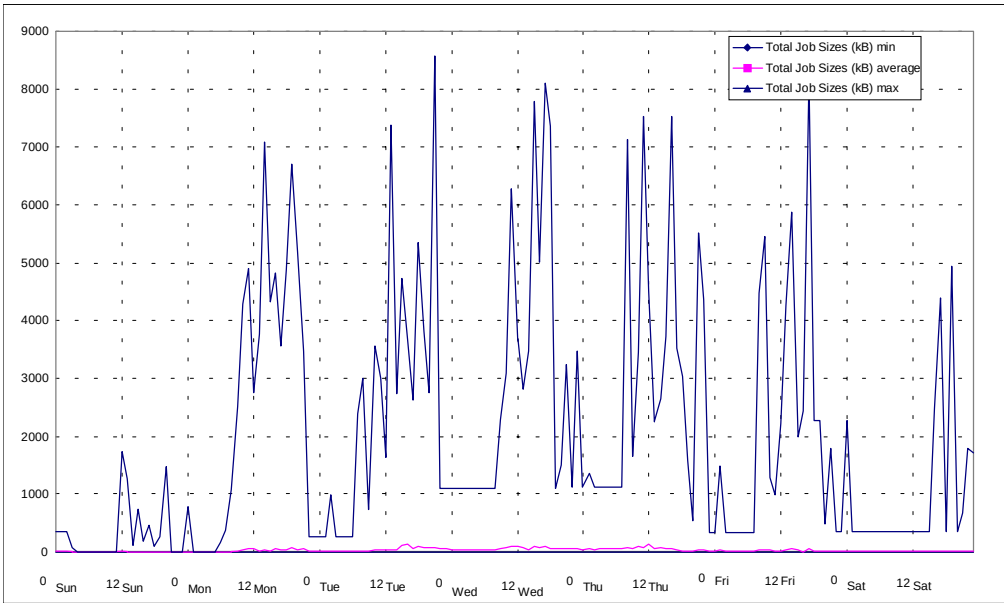
8.1.15. Number of Waiting Print Jobs in Spooler Queues

Under normal conditions there is not more than one job waiting in a print queue. Due to printer problems the number may rise. Note that this is not the number of processed print jobs (see "Printjobs Queued To Servers" on page 172 for this figure).



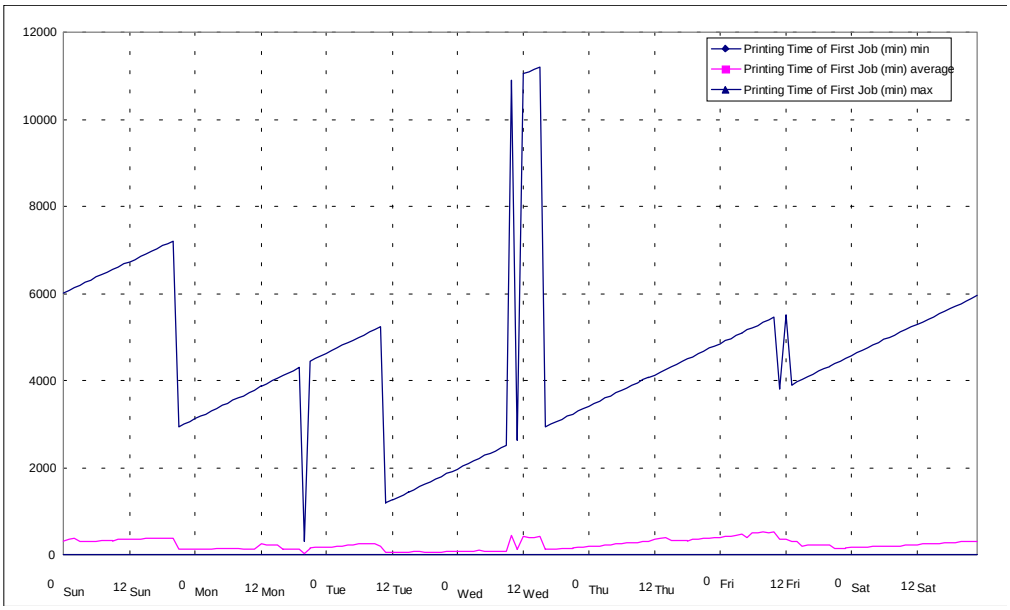
8.1.16. Total Job Size

The graph below shows how much data on average a spool server must be able to store at any time during one hour. We see that the actual maximum is at about 9 MB that were waiting to be send to printers. On average most print jobs are very small in size and there is nearly no data in the spool area of the server.



8.1.17. Printing Time Of First (Active) Printjob

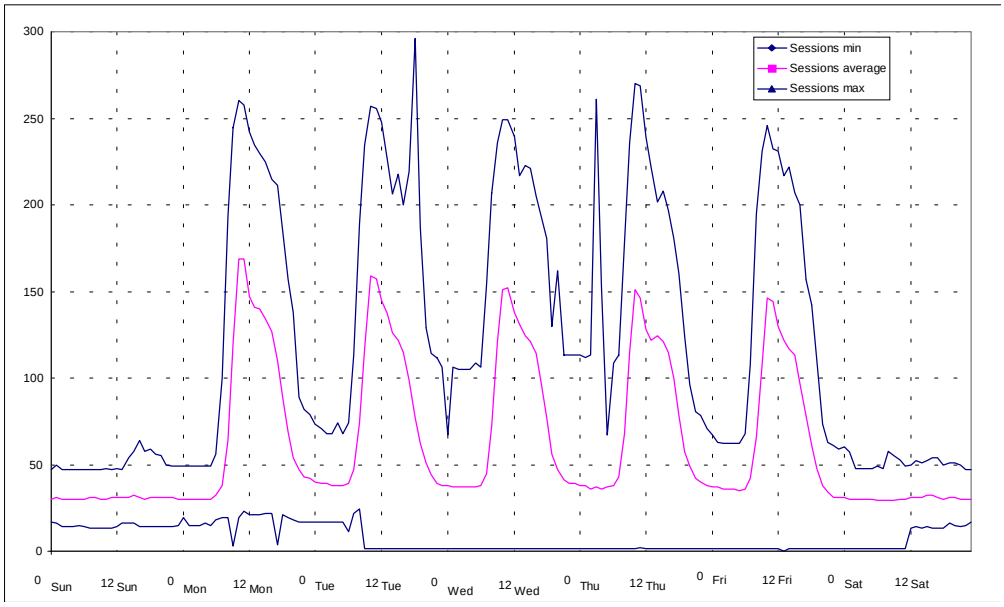
The following figure shows the time that the first print job (the one which is about to be printed) is waiting in this position or - in other words - how long it takes to process the job. Normally this time is very short. Longer times indicate printer problems and the time spend in detecting and correcting the problem.



As we can see it may take up to 200 hours until the job is spooled to the printer: that cannot be a very important job.

8.1.18. Number Of Server Sessions

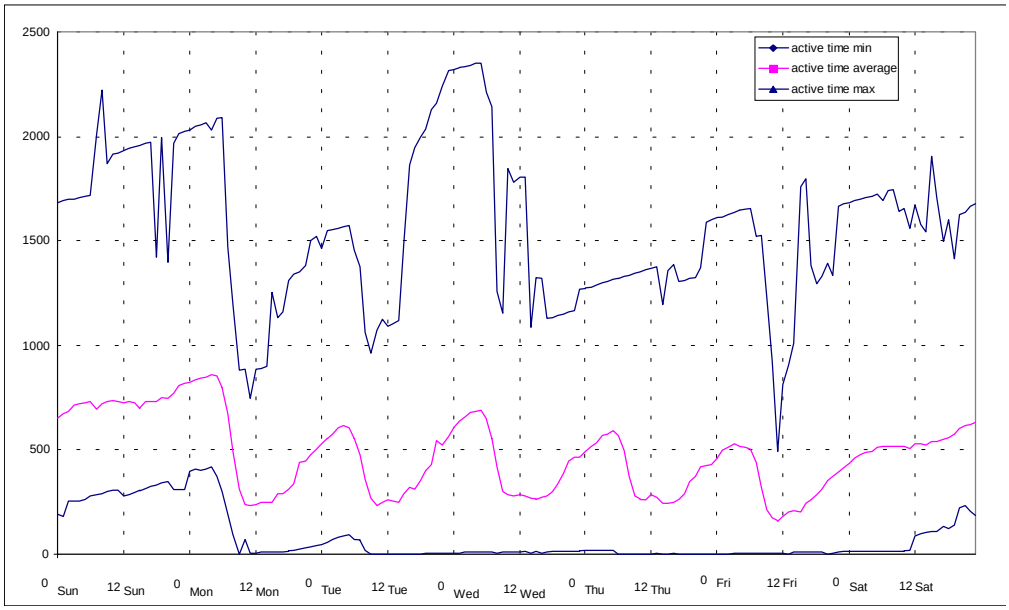
Note that this value is practically identical to the sessions which are reported by NETBIOS. If servers with additional server applications (like DB2 or Lotus Notes) would be part of the domain, these values could differ.



The difference in absolute values comes from the fact that this values are reported by the LAN server agent and therefore all servers in the domain were monitored (remotely) and account for the total numbers. In case of NETBIOS the local agent must be active and this component was not installed on all servers of the domain.

8.1.19. Active Time Of Server Sessions

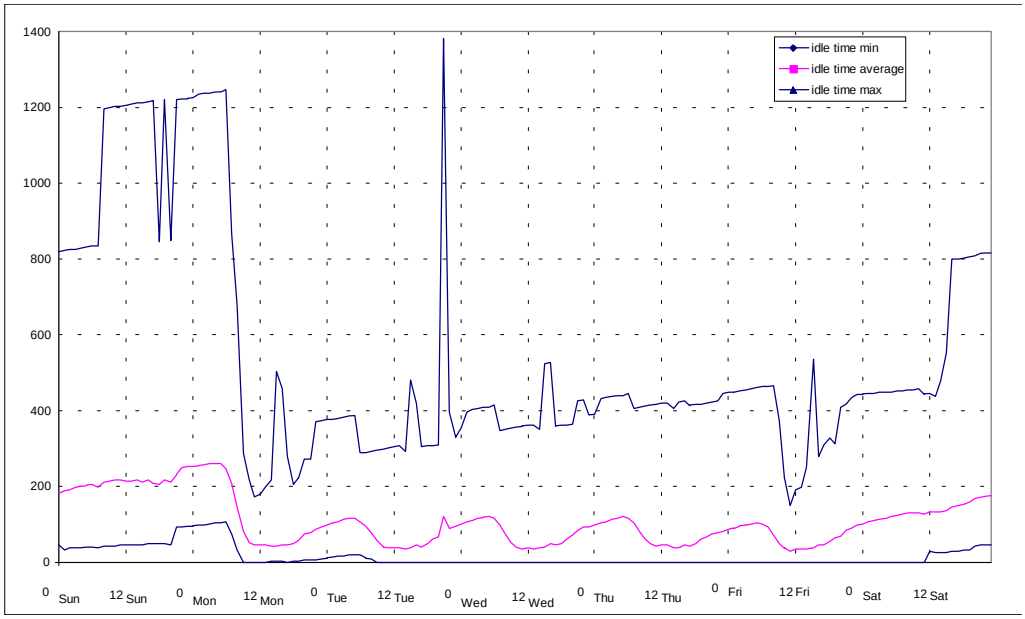
This graph shows the average of accumulated time that the sessions to a server have been active. This is an indication for the age of sessions and the duration sessions are used (or kept alive).



Only existing sessions contribute to the number. This explains why the numbers rise until about 8 am. At this time more and more new sessions appear. They "reset" the average age of sessions again.

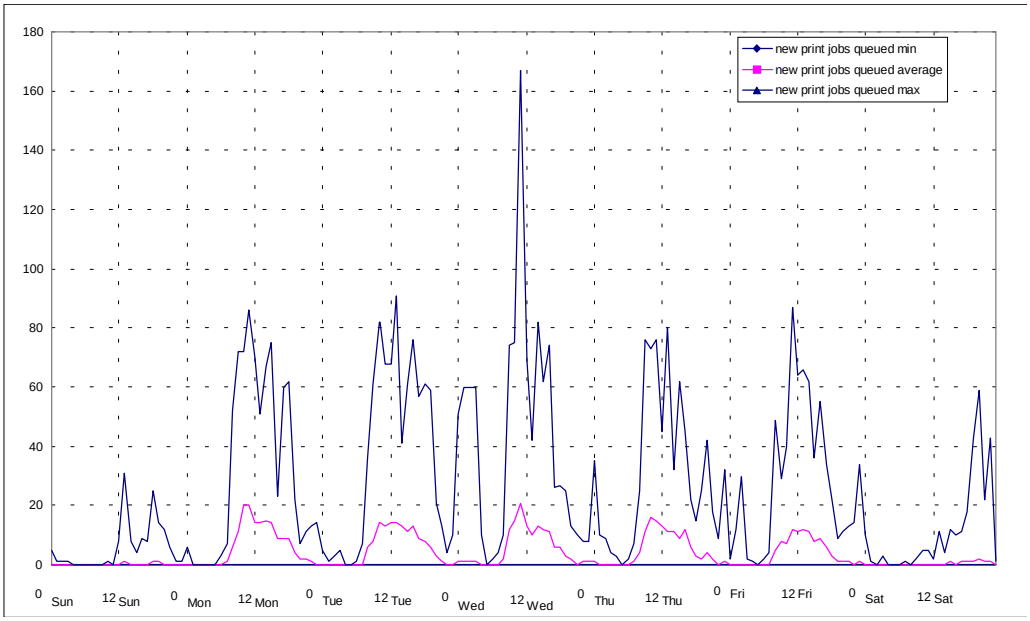
8.1.20. Idle Time Of Server Sessions

This measurements shows the average of accumulated time that active sessions have not been used. This is an indication for the number of "forgotten" sessions to stay alive. From the picture we see that many of them begin on Mondays and remain for a week. A week later the machine are reset. I likely source for that measurements are all machines that are rebooted once a week like any kind of server.



8.1.21. Printjobs Queued To Servers

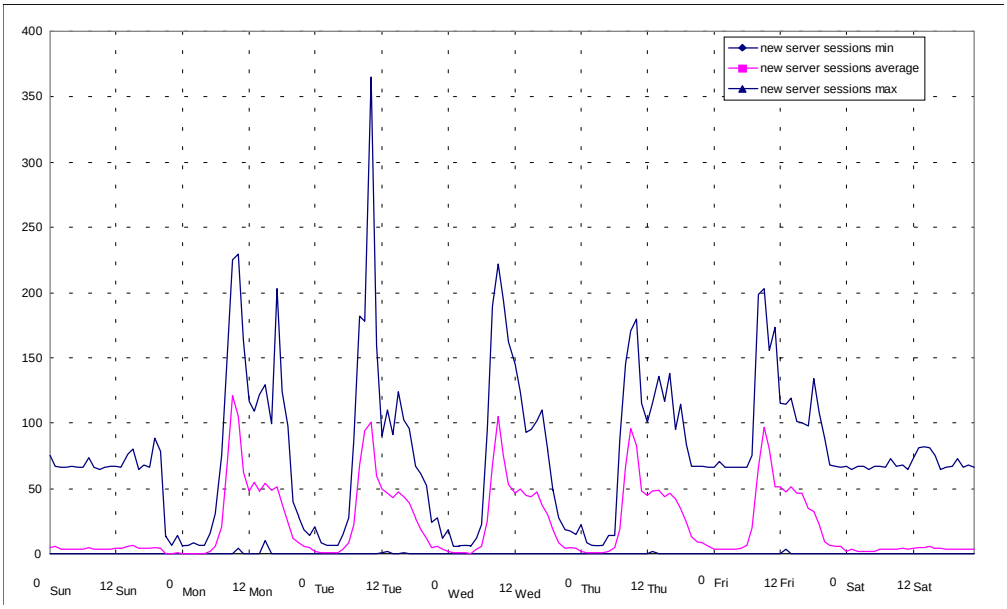
The next graph shows the number of print jobs that are submitted to a spooler queue during one hour. On average the daily peak load is about 20 jobs per hour. From the server's point of view that is not much but it may be too much for the connected printers.



Note, that it is not possible to monitor the number of pages which have to be processed by the printers. Maybe, that would be a better measurement for the load on the printers than the number of jobs.

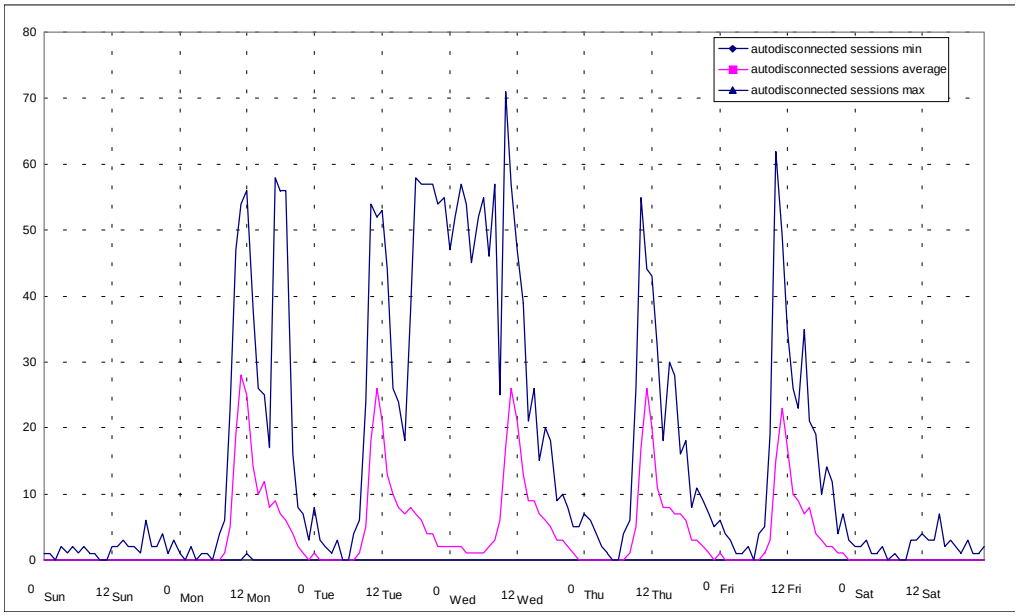
8.1.22. New Server Sessions

This resource attributes shows how many new sessions have been established during one hour. We can see that most sessions are created between 6 am and 10 am and that the peak load is highest on Mondays and decreases over the week. Besides a window of no activity after midnight all the day new sessions are created. We assume that some background application creates and discards sessions.



8.1.23. Autodisconnected Sessions

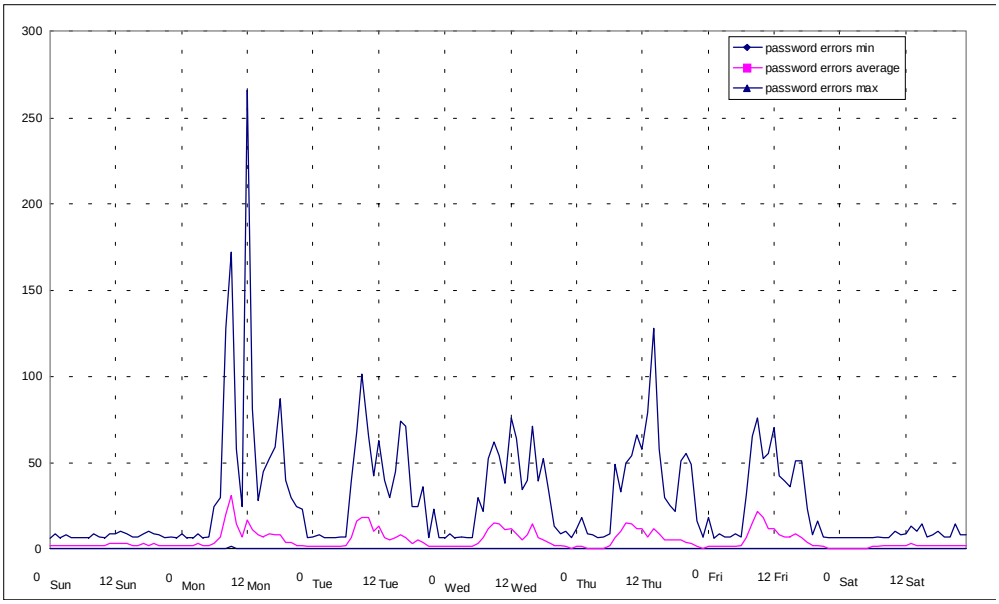
This measurement shows the number of sessions that have been closed down by the server due to lack of activity. This behavior must be enabled on a server otherwise a session will remain open until the requesting application closes the session.



The peak shortly before noon indicates that about 20% of the newly established sessions are not needed.

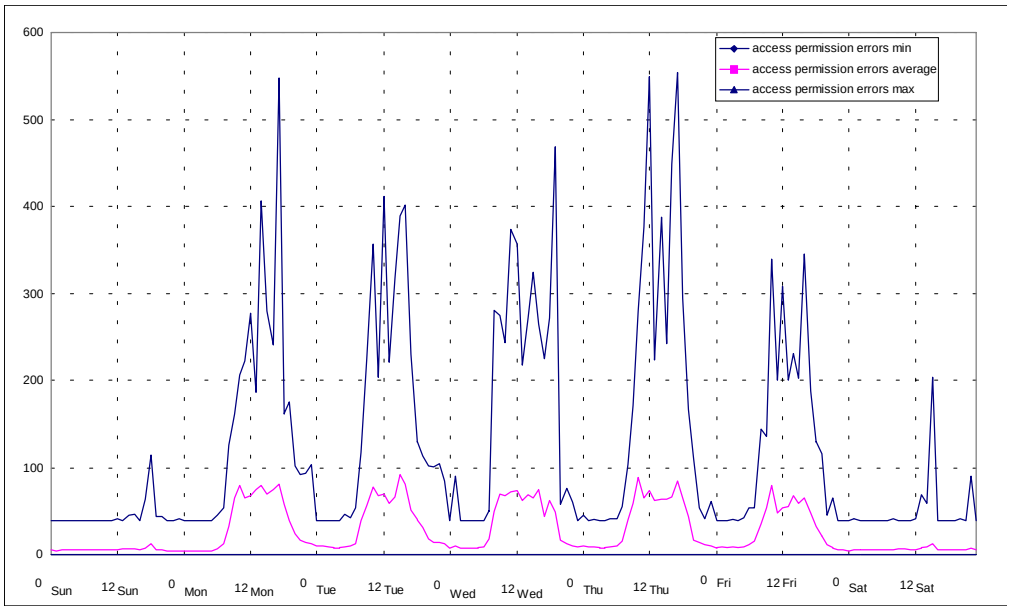
8.1.24. Password Errors

This numbers indicate the number of times an application tries to access a shared resource but supplies a wrong password. This happens when a user enters a wrong password or if he is logged on to another domain and uses a different password there. Every time he tries to access a resource in the observed domain a password error occurs. The user may not even notice this. For security reasons the number of illegal accesses should be monitored all the time. Unusually high numbers may indicate an intended break into the domain or an increasing number of cross-domain access with out-dated passwords.



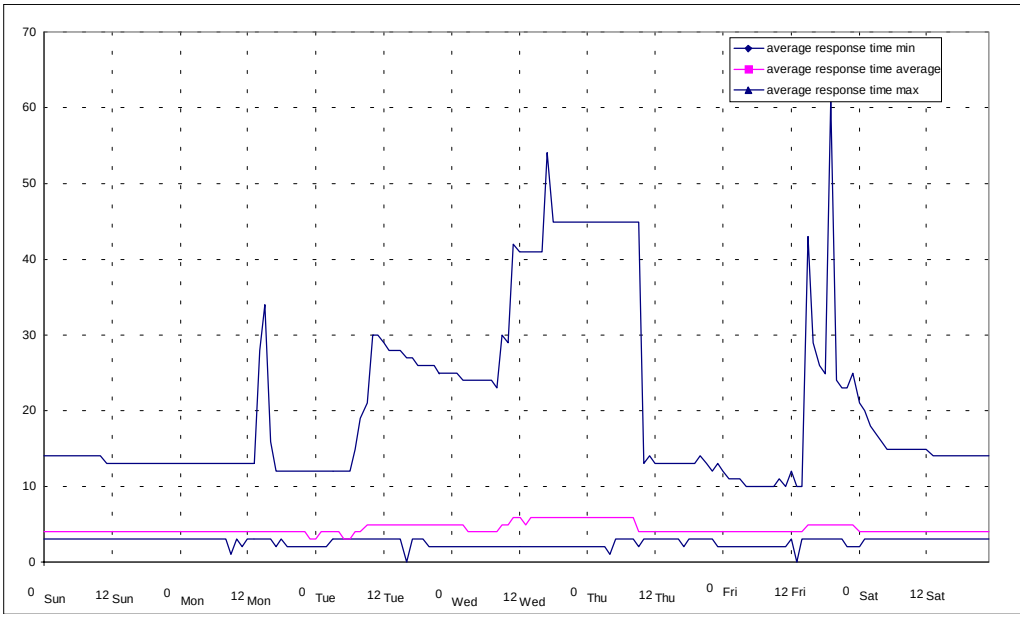
8.1.25. Access Permission Errors

In contrast to the number before this numbers indicate the number of time an positively identified user tries to access a shared resource he is not allowed to access. Again this may happen without the user noticing this (for example, when an application tries to scan a directory tree).



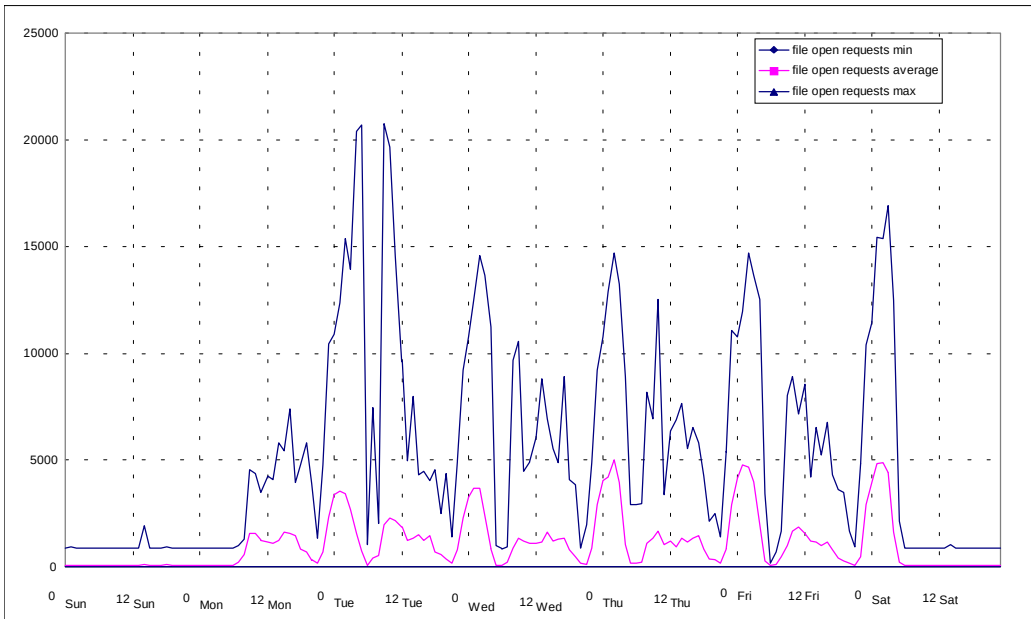
8.1.26. Average Responsetime

This is the average time in milliseconds the server needs to answer a request. We have no information about how the server measures this value. During some experiments we could not find any relation between this instrument reading and actually measured response times.



8.1.27. File Open Requests

This resource attribute gives the number of times that clients ask for opening a file on the server disk. This is an indication for usage of files on the server.



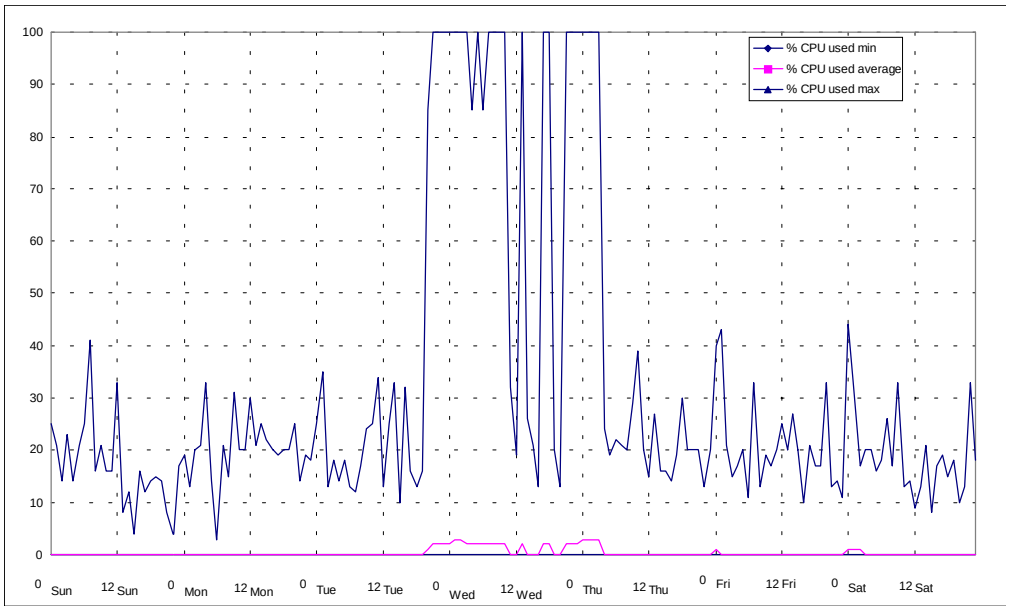
The peaks short after midnight after an work day indicate that most requests originate from automated procedures but not from user interaction.

8.1.28. (Virtual) Memory Available

This graph represents the sum of virtual memory available to applications at the server. Mainly it is in indication on how this number varies over the day. We can see that there is not much variation because in the observed environment the number of processes is rather stable and most processes allocate most of required memory during startup.

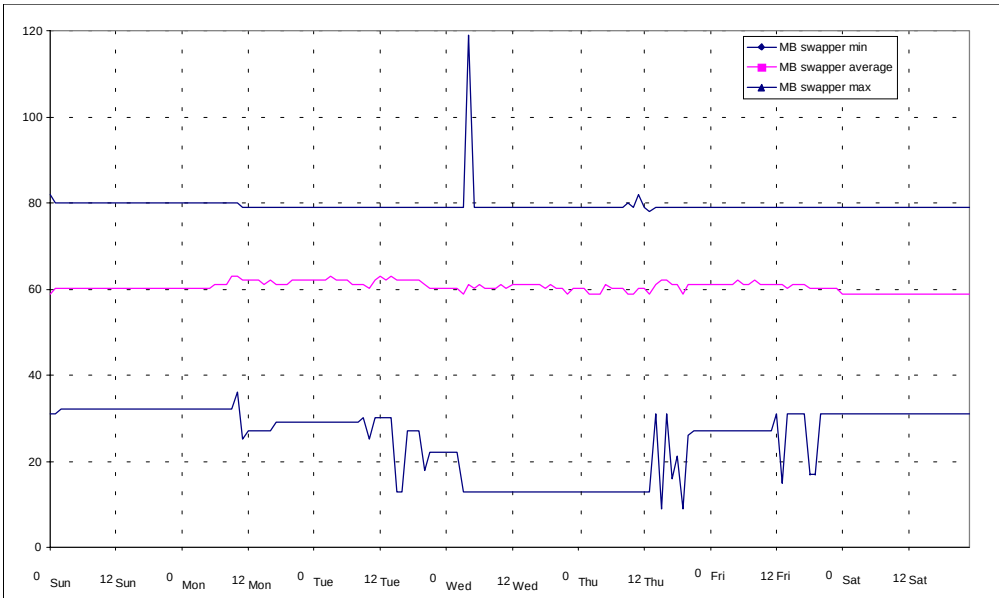
8.1.29. CPU Usage

This graph shows the average CPU utilization on the server machines. We can see that most of the time there is virtually no utilization of the CPU. The few non-zero values are around midnight. As other graphs already indicate there seem to happen automated some routine procedures (like backup) which slightly use the CPU.



8.1.30. Size Of Swappfile

This graph shows the amount of process memory that is swapped to disk. OS/2 writes to disk only the information that does not fit into memory and does not prepare a swapfile relative to the size of RAM. The summary graph mainly indicates the variations over the day and again we can see that there is not much variation. Memory usage is very stable.



8.2. Case II

The second scenario is a domain of 21 server machines which support up to 400 user in an back office type of work group. This domain represents a large domain which spans several organizational units. Although from a technical point of view a domain could support up to 1.000 user, in reality, with the hardware of that time (setup 1994) 400 users were already the maximum.

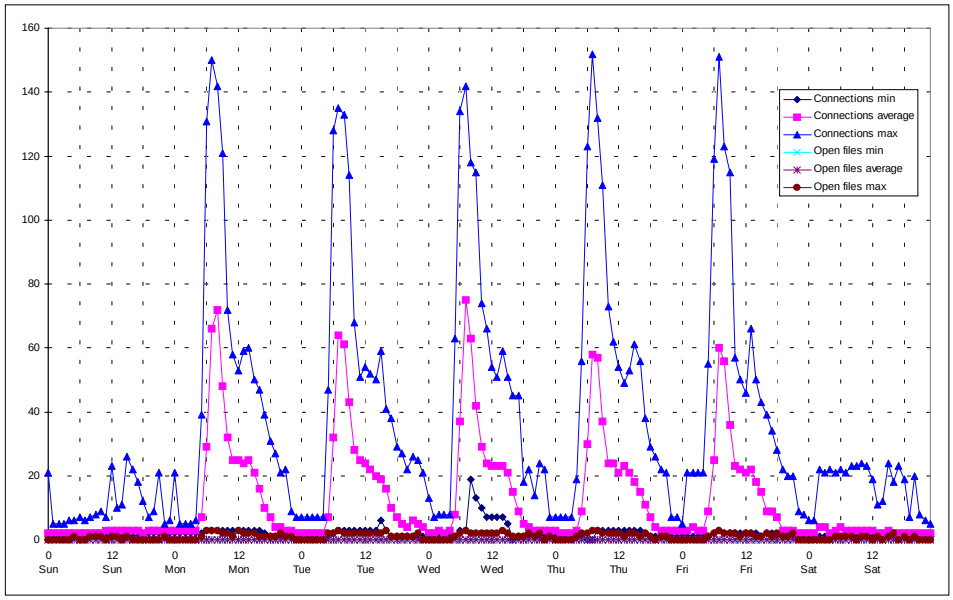
With a few exceptions the roles and profiles of servers and users were not known to us. Therefore this represents the more usual scenario of an administrator who usually does not care to much about servers and domains (especially if he/she has to maintain many of them).

8.2.1. Server Characteristics

To better understand the usage of the different servers the number of connections and open files is used. These two figures give a good idea about the number of users who actually make use of the services of each machine. Unlike the first case study, I did not know details about role and tasks of the servers. Therefore, like in the previous case study, the number of connections and open files is presented for each server to give an indication for the utilization of the server. The read can see that many of the servers are hardly used. We do not know what the roles of these machines are.

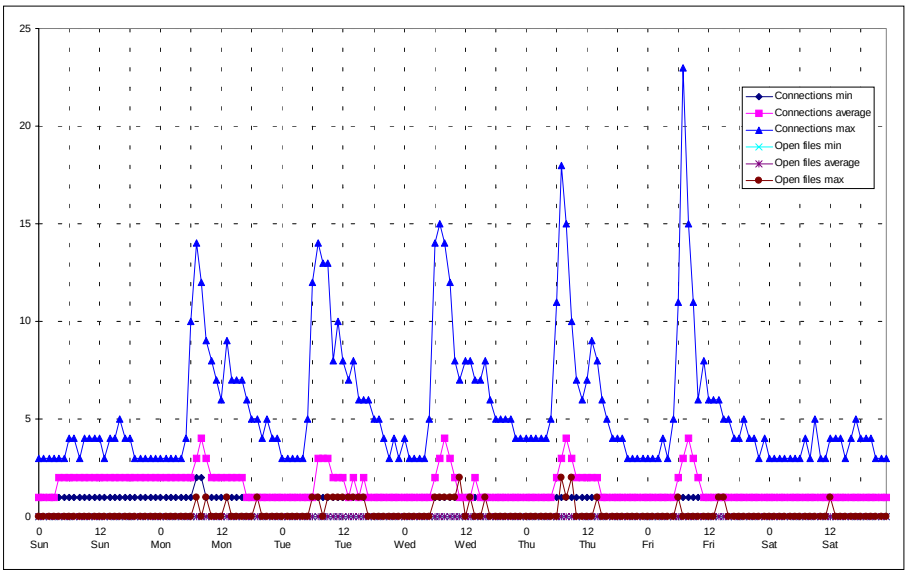
FAW9500S - Domaincontroller

Note that, like in the first case study, a good deal of the existing load is covered by the domain backup controller. Logon-load is balanced between the main controller and the backup controller. In this scenario the main task of the backup controller is to keep the domain running in the case of a failure of the main controller. In other scenarios load balancing may be a more serious task.

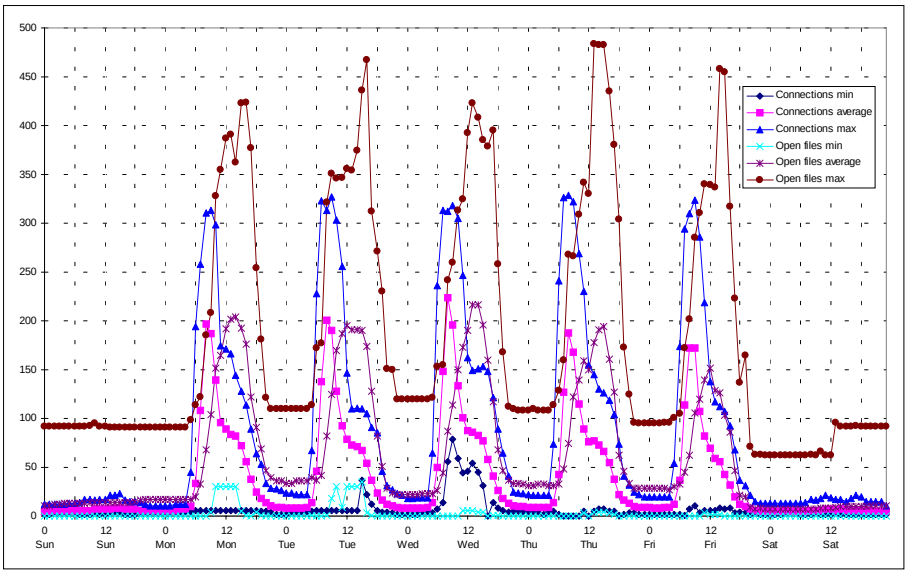


Each active user has a connection to the domain controller. As the controller does not provide much more the number of connections is identical to the number of users which are handled by that machine. Normally there are no open files because the machine is not used as a file server.

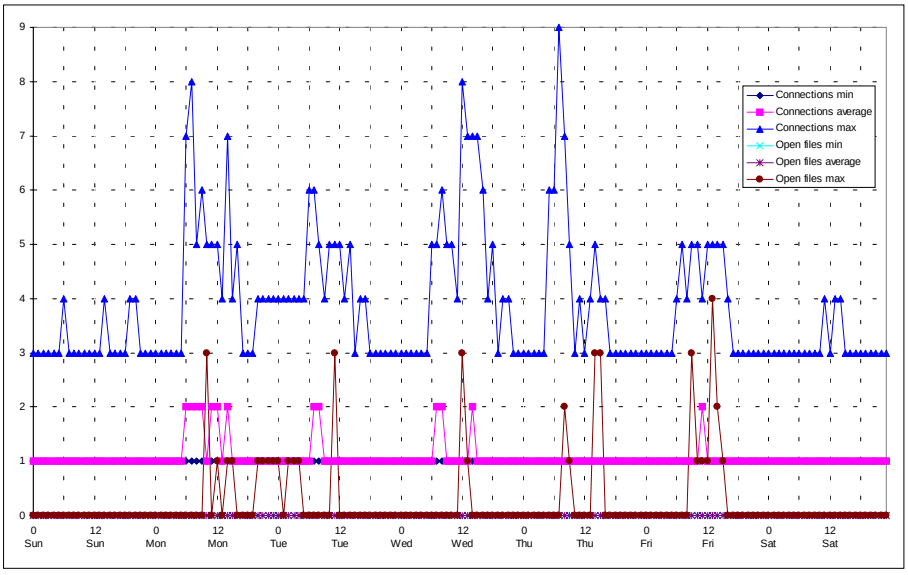
FAW9510S



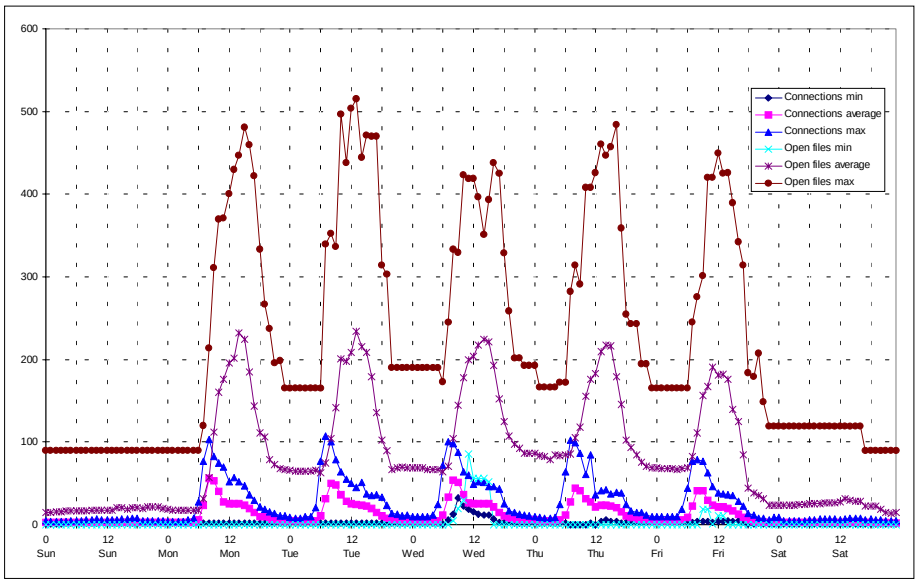
FAW9520S



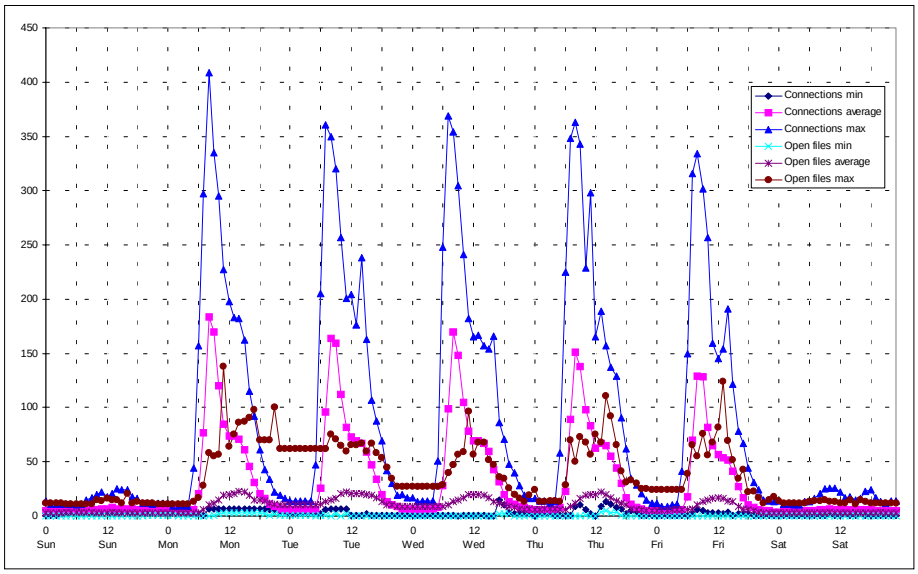
FAW9521S



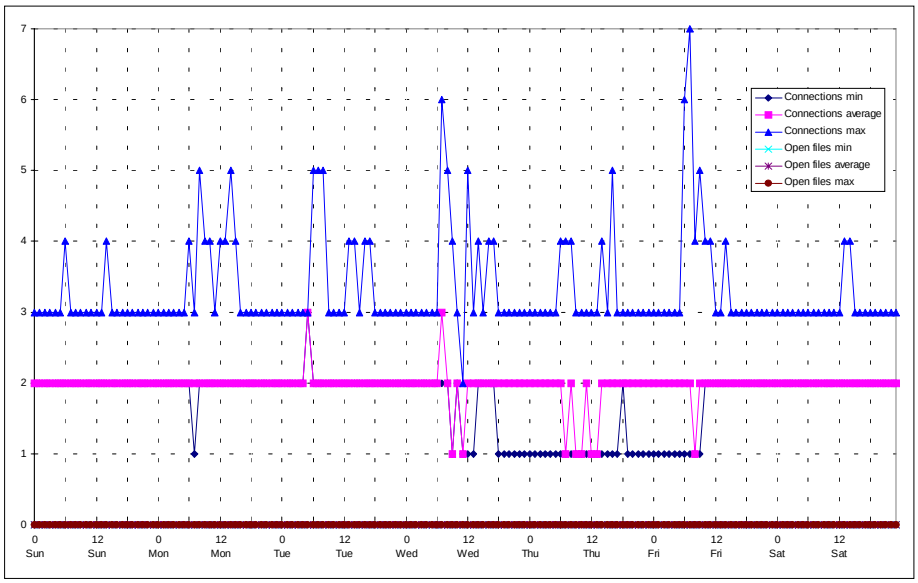
FAW9530S



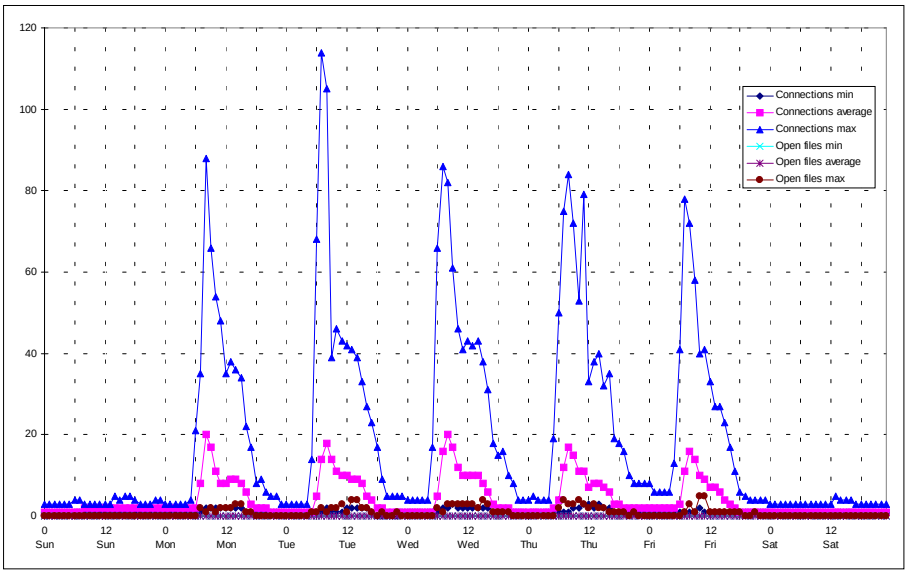
FAW9535S



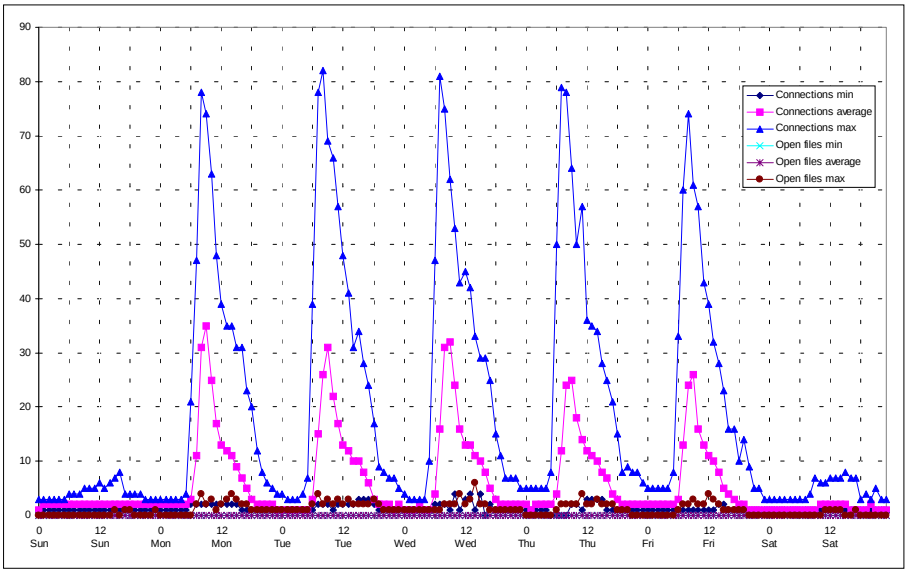
FAW9536S



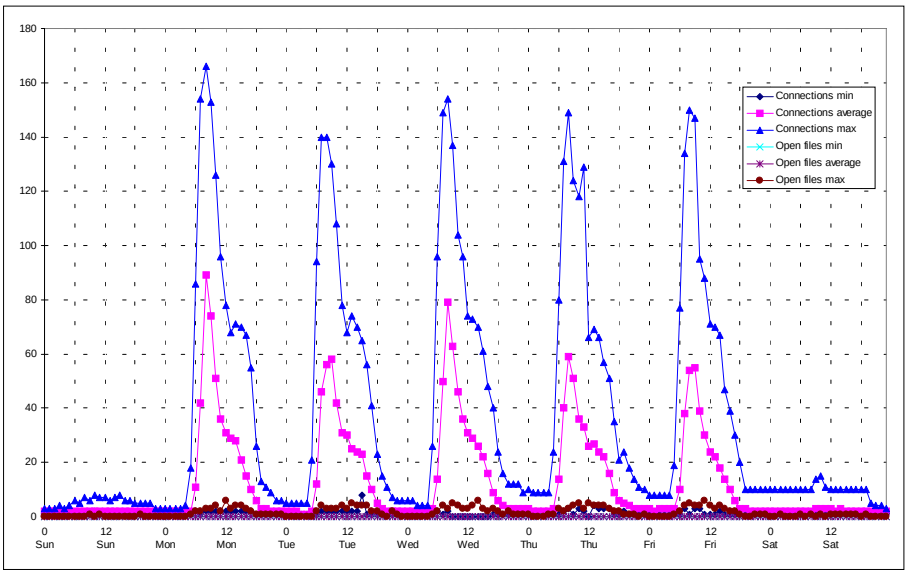
FAW9540S



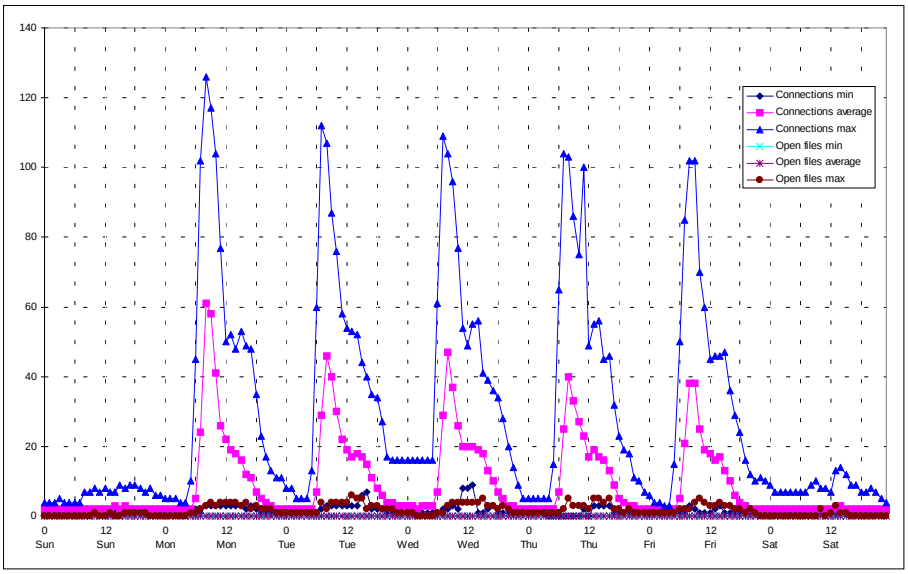
FAW9550S



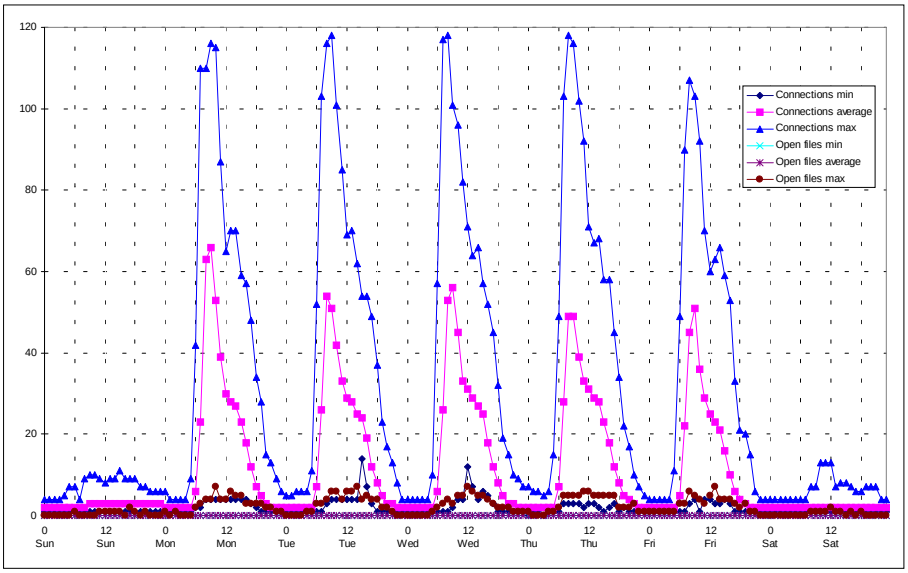
FAW9551S



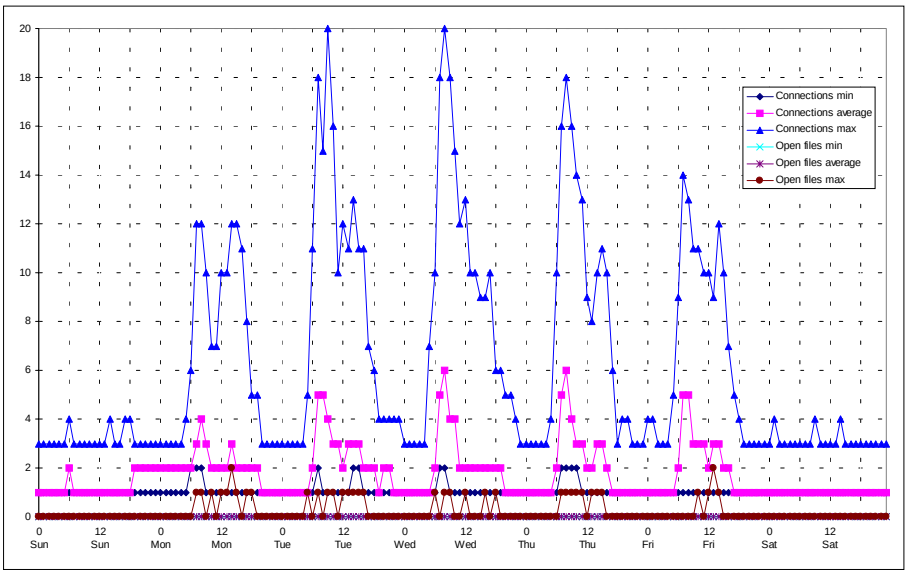
FAW9552S



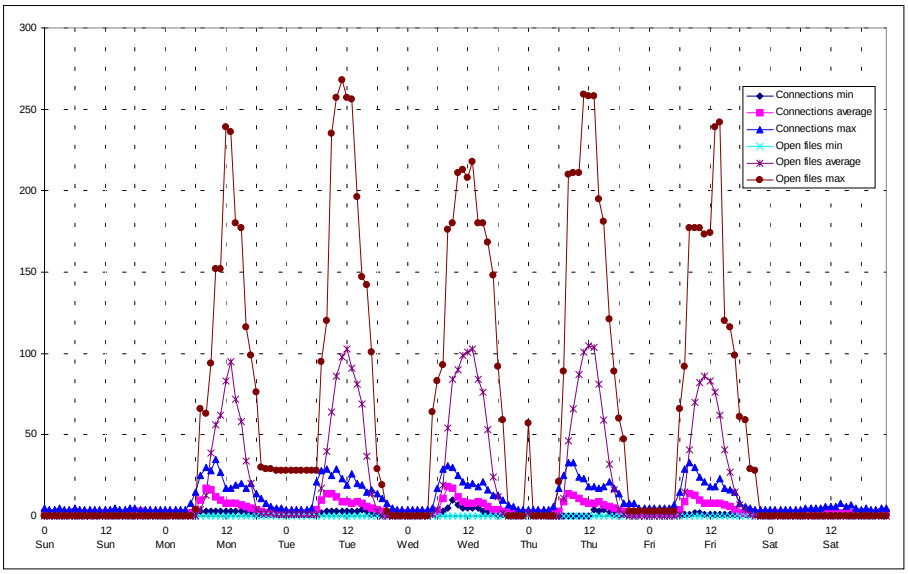
FAW9553S



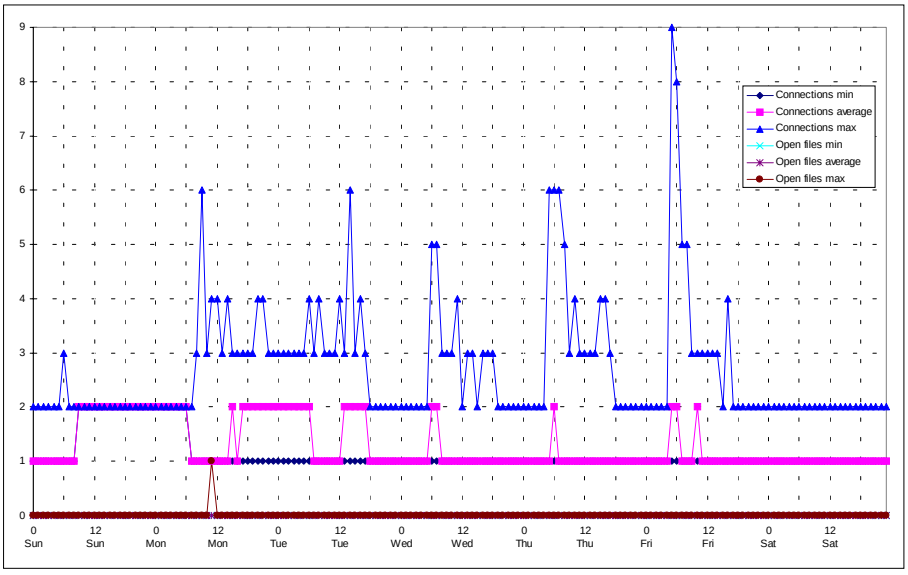
FAW9554S



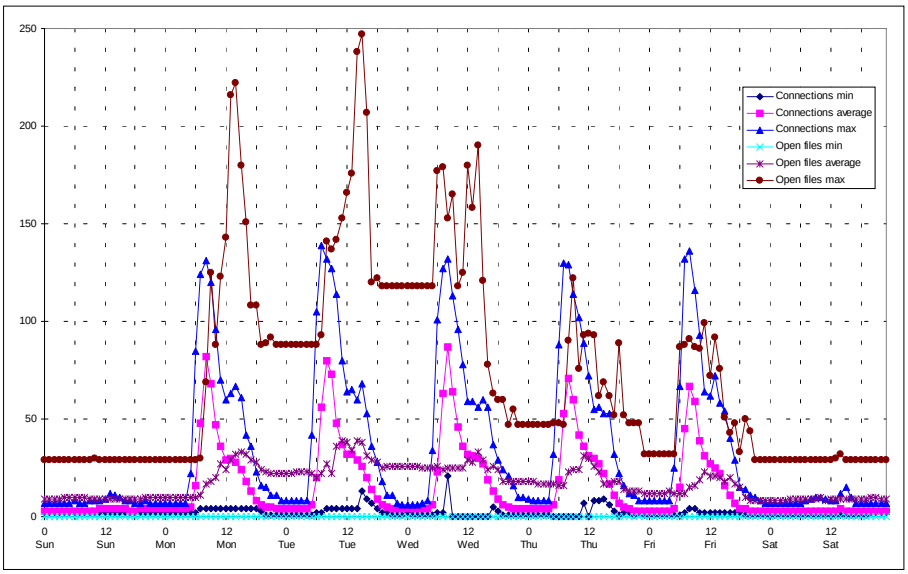
FAW9570S



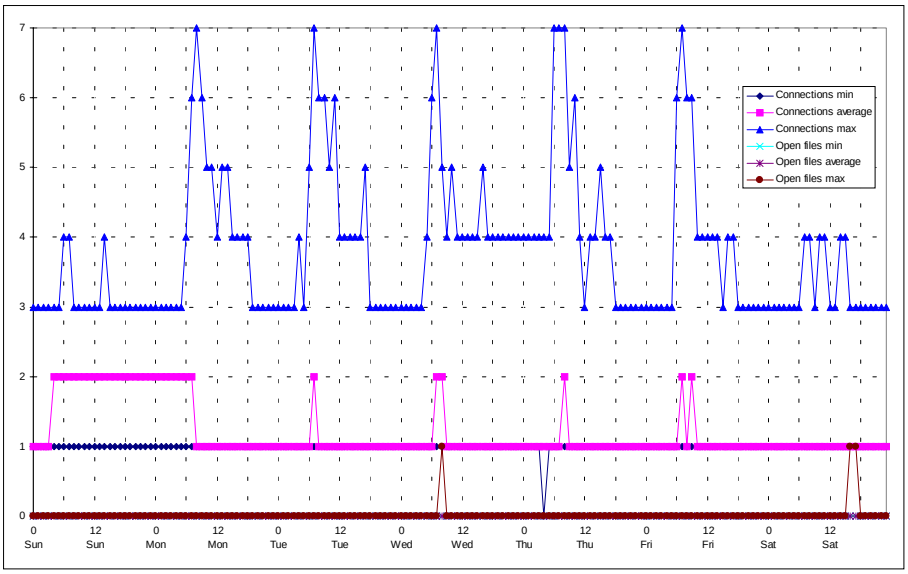
FAW9576S



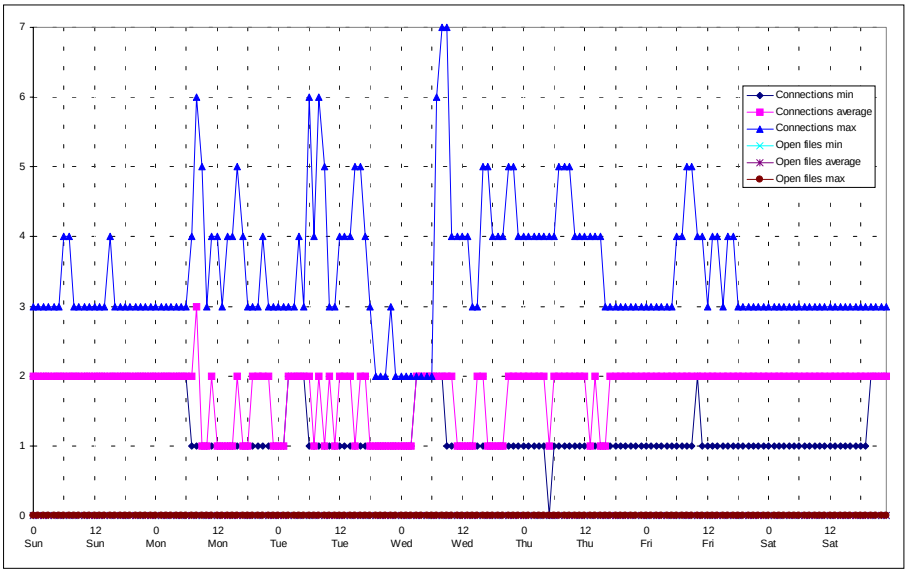
FAW9580S



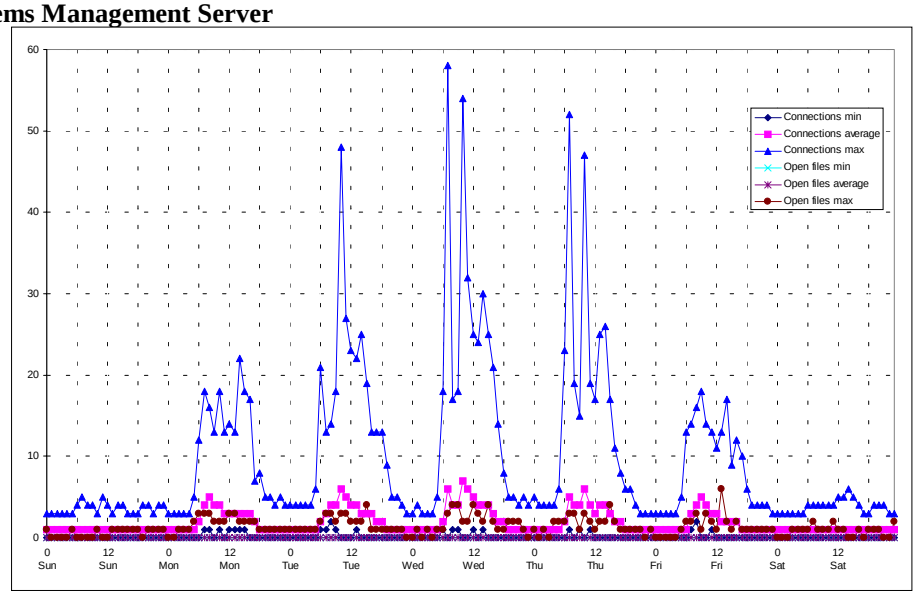
FAW9585S



FAW9590S

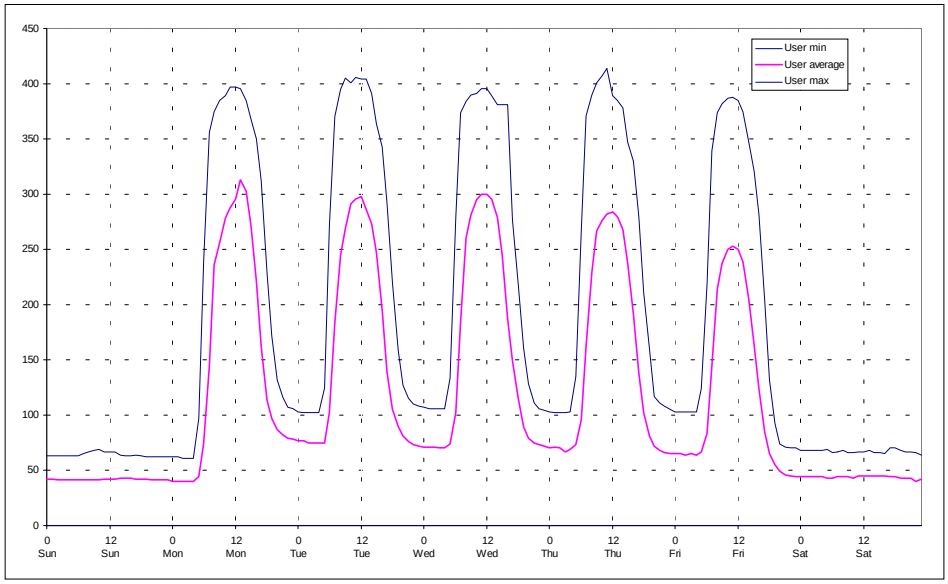


FAW9595S - Systems Management Server



8.2.2. Active User

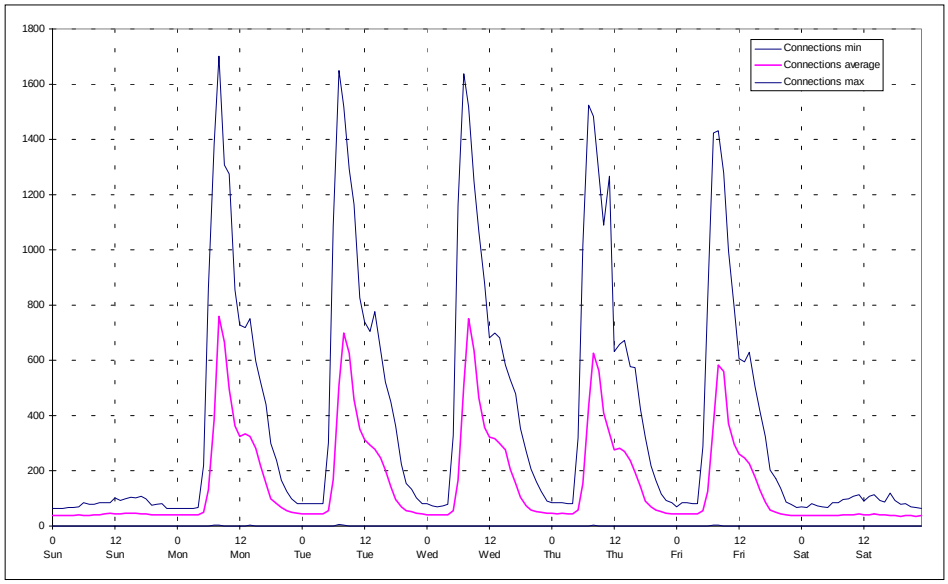
First we look at the number of active users. As mentioned above office hours drive the number of users and we see a cyclic up and down of users. A number of users is logged on all the time. The number of users we see at night and on weekends is equal to the number of client workstations which are not turned off during non-working hours. We can derive that during the week about 75 of 400 users do not turn of their machines while over the weekend only 45 clients stay active.



Note the several "long weekends", that start at Thursday, and that are rather common in Austria (up to five times per year) influence the average week slightly (see the lower average peeks on Thursday and Friday).

8.2.3. Active Connections

The number of active connections to resources on the servers of the domain correlates to the number of users, but the peak load occurs much early than the maximal number of users (at about 10:00).

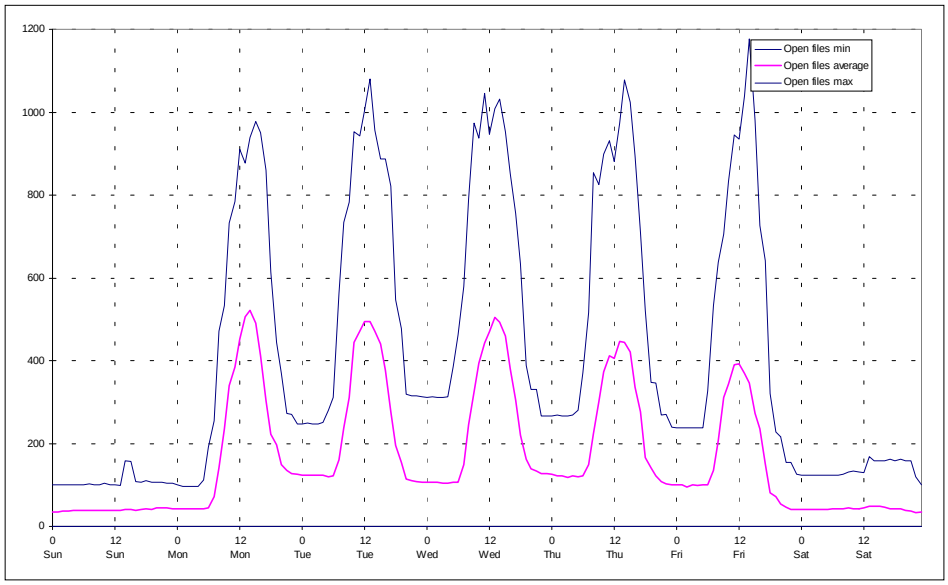


Several things can be noticed:

- the number of connections reach their peak at about ten o'clock in the morning (while the number of users top at about twelve o'clock)
- the decline in the number of connections follows immediately (in contrast to case I); the reason for that is a different connection policy: unused connections are disconnected after a shorter time
- that means the most of the connections are not really needed or used
- initially, each user has two or three connections, then the number drops to one per user (which has to exist due to the active logon)

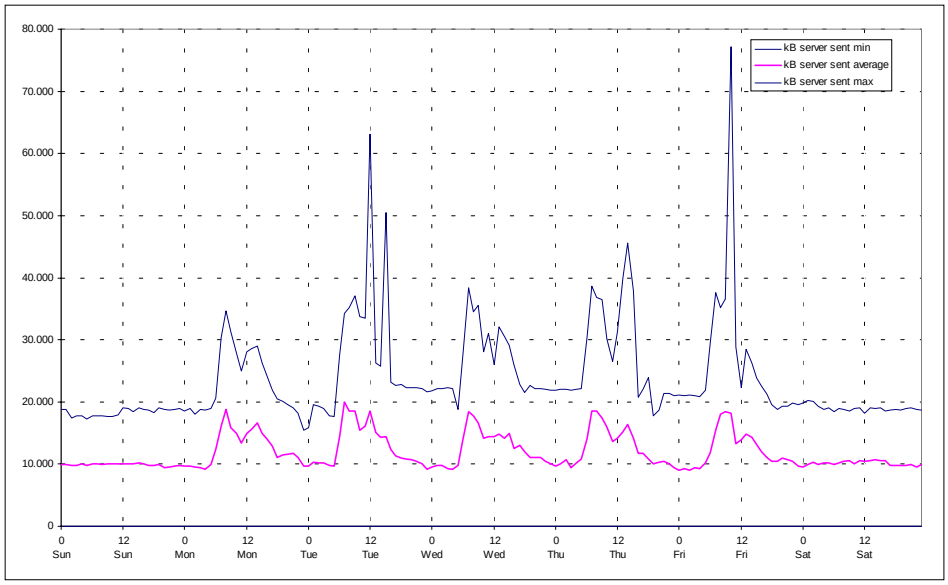
8.2.4. Open Files

The average number of files correlates with the number of active users and we can see that on average each user has one or two open files only. That means that many users do not use available application software from a server but install it on their local hard disk.



8.2.5. Data Sent to Servers

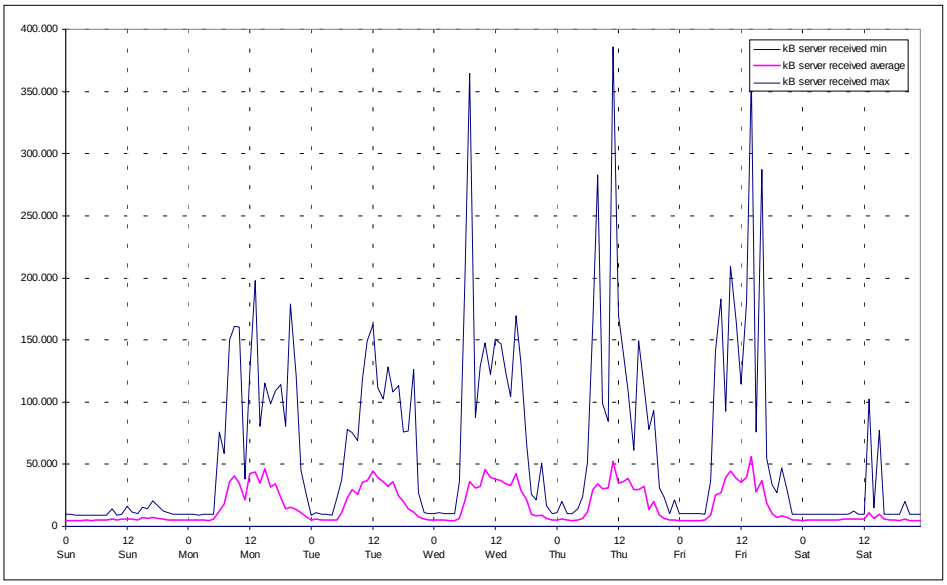
The next figures shows how much kB of data are sent to all servers per hour. We can see that up to 80 MB per hour are sent. In contrast to case I there are no peaks during the night.



For the average value we still can notice a correlation with the number of users. Remarkable is that on average there is a traffic of about 10 MB per hour during off-time hours. That seems to be the amount of information the server has to exchange even without direct user activities.

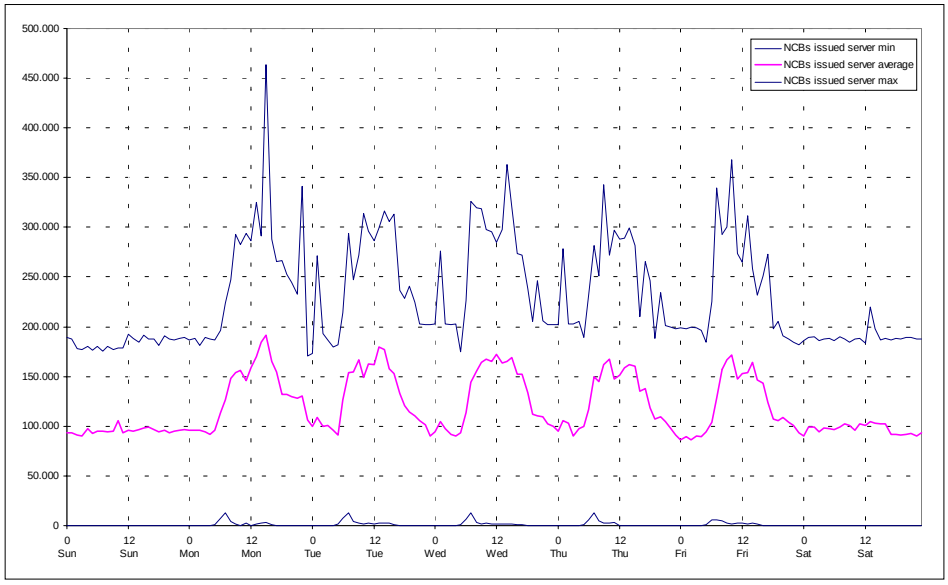
8.2.6. Data Received From Servers

Despite the small time frame of peaks for connections and users, data transfer activities start early in the morning and end not much before 0:00. As nearly nobody is logged on during that time I assume that a good deal of the traffic is generated by automated procedures that are triggered by applications.



8.2.7. Network Control Blocks Issued By Servers

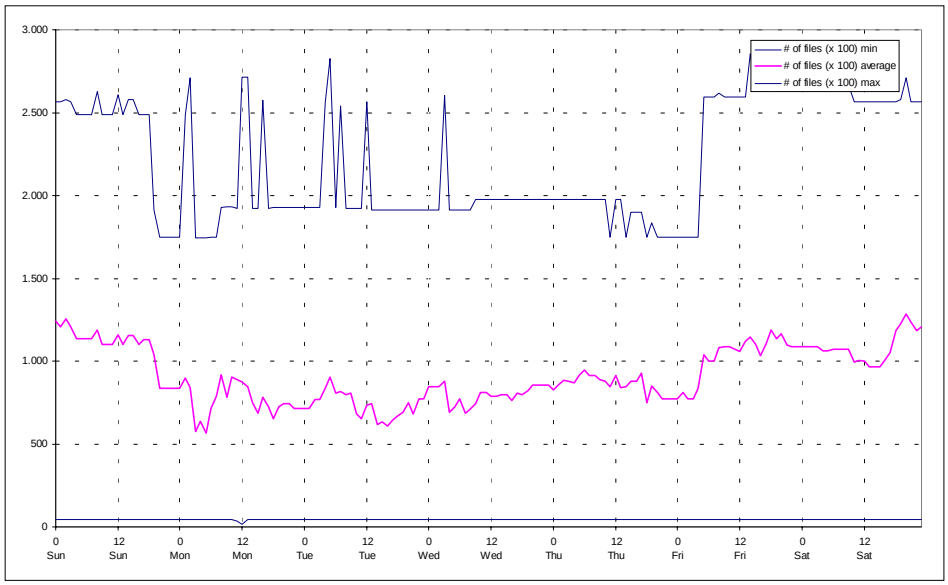
The LAN server software uses NETBIOS as transport protocol over the network. NCBs (network control blocks) are the means to work with NETBIOS. Applications reserve and issue NCBs to communicate over the network. Therefore the number of NCBs issued is a measure for all network activities (while it does not indicate the amount of data which is moved over the wire).



8.2.8. Number Of Files

Next we have a look at the file systems of the servers. Note that not all servers of the domain were part of the detailed measurements. This information gives some insights in the usage of the file sharing service. In the average over one year we see

that the number of files vary in the range from 50.000 to 125.000 files during the week.

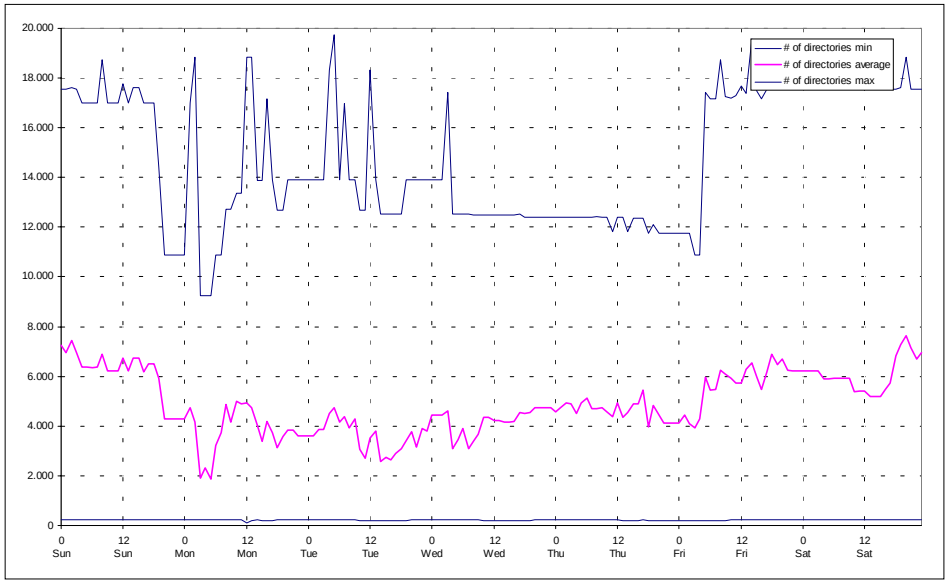


(Ignore the minimum value in the graph above - it does not have any meaning.)

Over the week the average number of files rises and then drops on Monday morning.

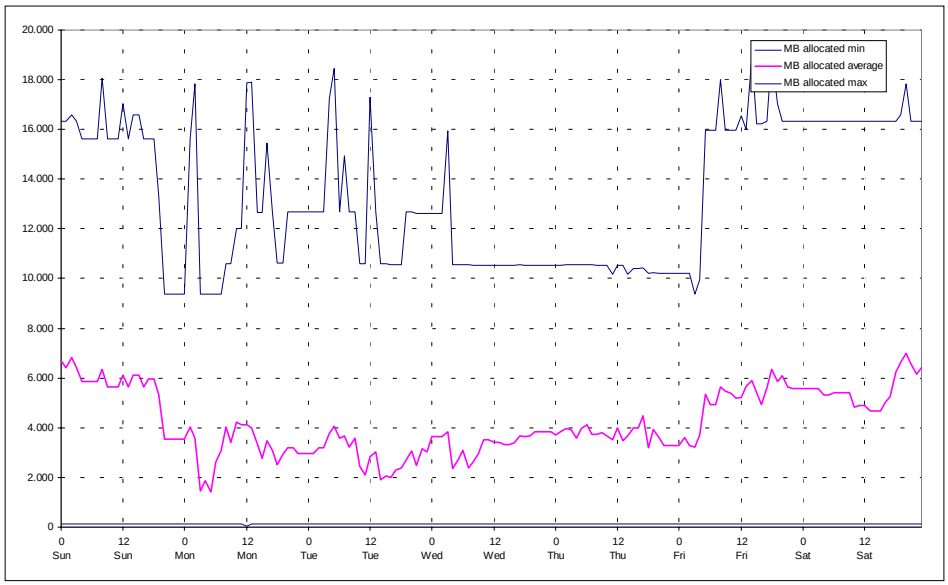
8.2.9. Number Of Directories

In the graph below we can see that even the number of directories vary between 2.000 and 8.000. That means up to 6.000 directory entries are created and removed during the week.



8.2.10. Allocated Disk Space

Like in case I the previous and the next pictures are nearly identical. One has to look twice to see the differences. Therefore the same conclusion seems to be valid: cleanup is done directory-wise and a directory contains on average one megabyte of data.

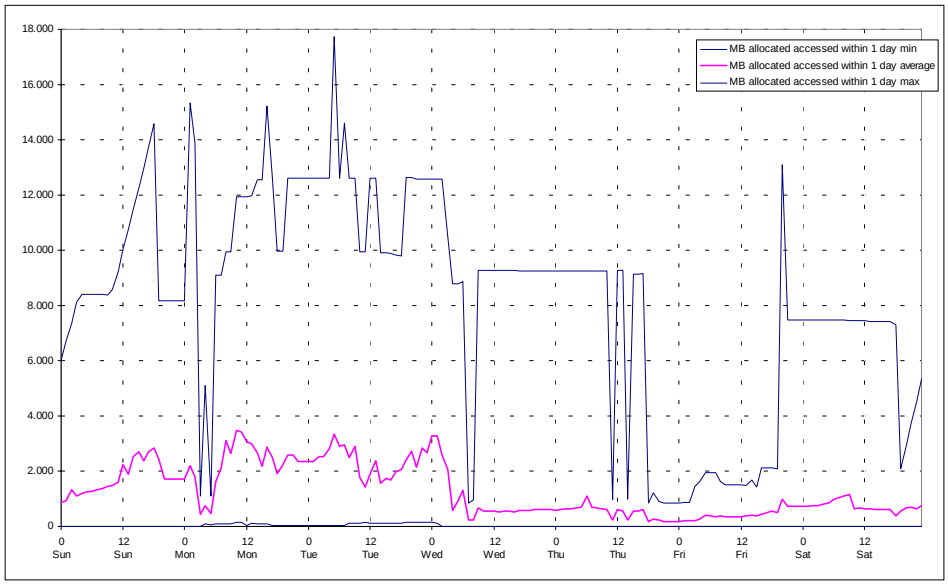


8.2.11. Access To Diskspace

This and the following chapters show the access pattern and the aging of information on the server disks.

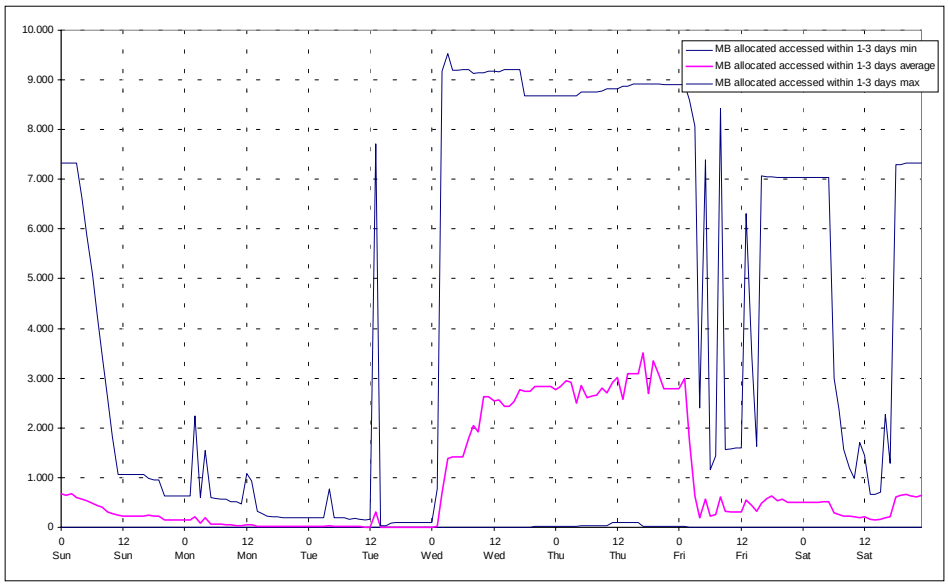
Diskspace Accessed During The Last Day

From the following picture one has to have the impression that most work-files are used only between Sunday afternoon and Wednesday morning. On these days most of the allocated information is really accessed (used).



Diskspace Accessed During The Last Three Days

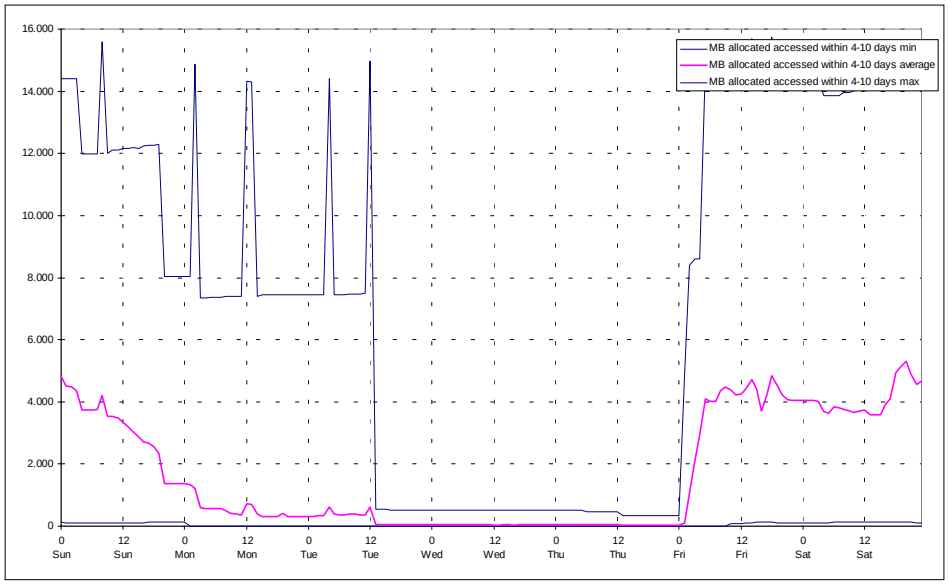
This case shows a similar behavior than case I: starting with Wednesday morning a lot of the files start to ages out (thus they were not used later than Tuesday noon). Then the files fall into the next category (see below).



On Monday and Tuesday nearly no files are in this category: either there were already used or they aged out to another category.

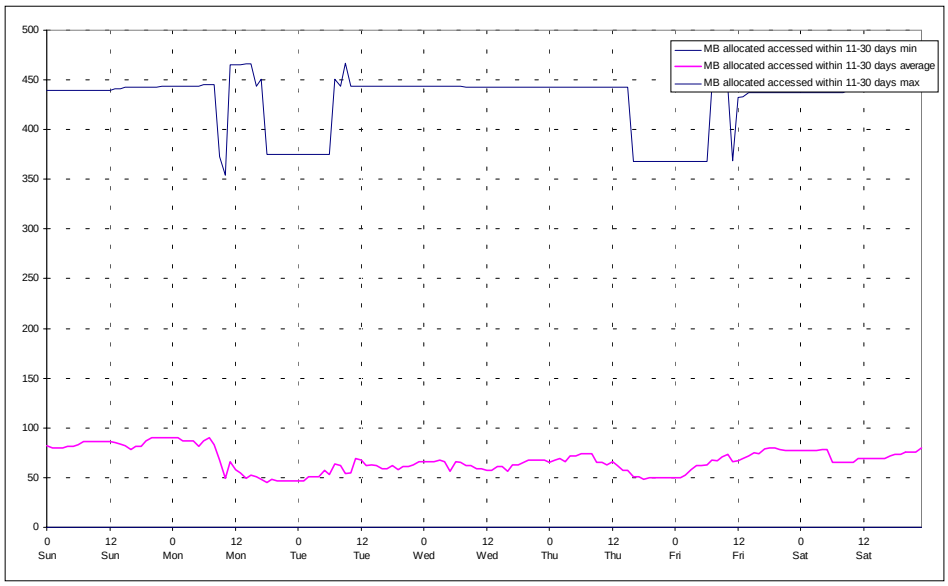
Diskspace Accessed During The Last 4 To 10 Days

Many of the files that were used only on Monday "move" into this category on Friday. This is another indication for the assumption that most of the server related work is done on Monday. During Sunday and Monday most of the files are "touched" again (I assume that this due to "end-of-week" backup procedure.



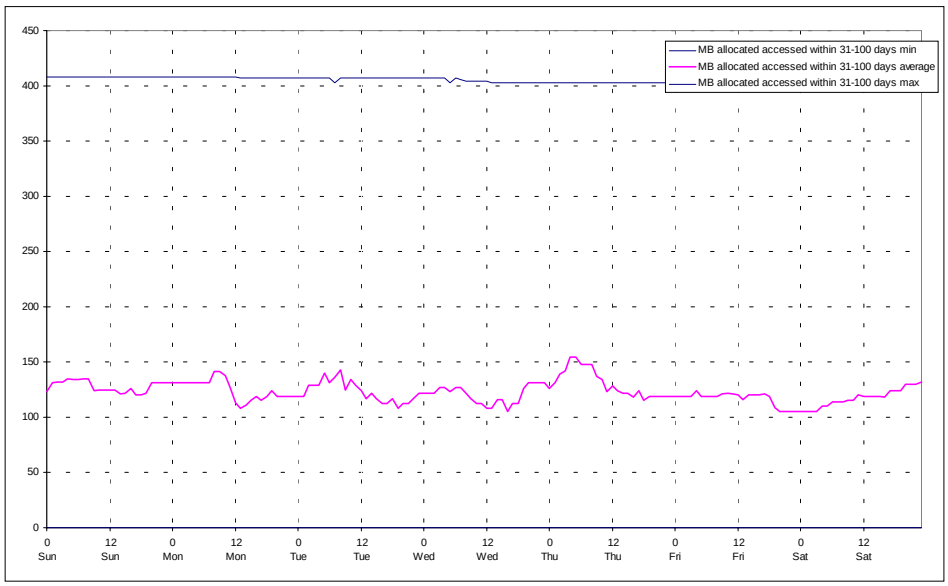
Diskspace Allocated Within the Last 11 To 30 Days

The amount of information which is not used for a month is relatively constant. This information is not longer used and ages out. About 80 MB out of 18.000 MB remain in this category (that is not much).



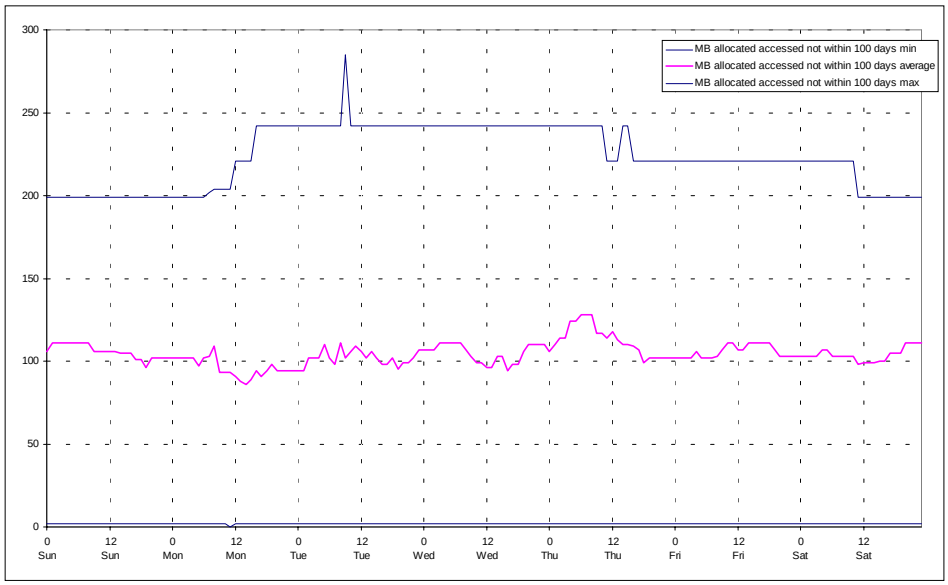
Diskspace Accessed Within The Last 31 To 100 Days

The same can be said for information which is not used for more than one month. About 150 MB out of 18.000 MB fall into this category.

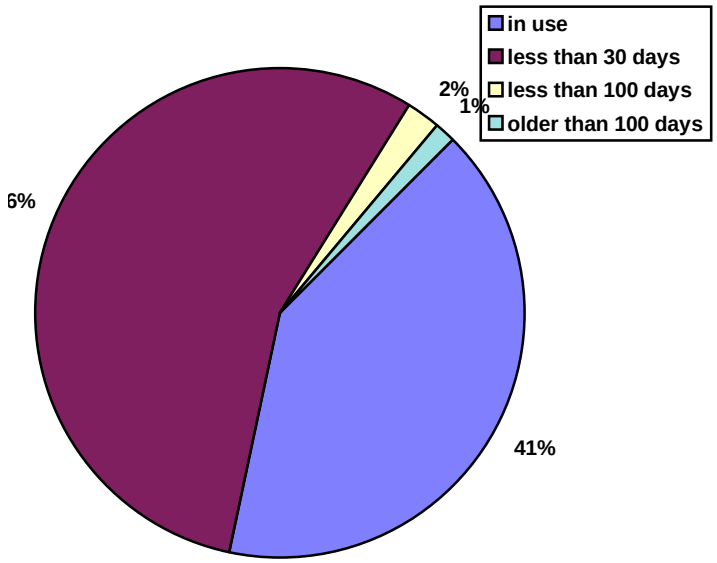


Diskspace Not Accessed Within 100 Days

This graph shows the amount of "dead code" on the server systems. About 110 out of 18.000 MB seem to be of no further use.



Summary



In this domain there is nearly no dead data.

8.2.12. Cache Efficiency

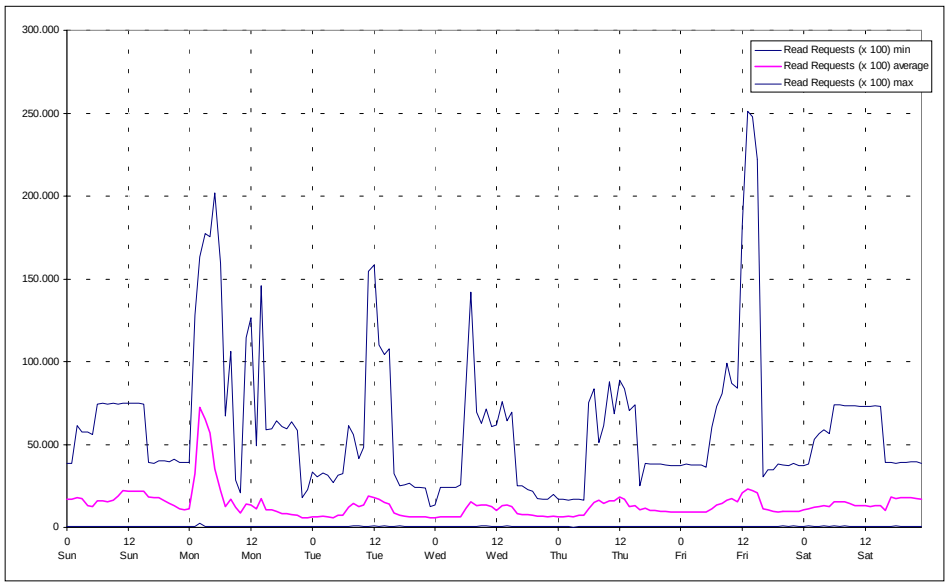
The HPFS386 file system makes available some information about the efficiency of the file system and the volume of access to the file system.

Summary

Most of the information on the server disks are read-only. Therefore there is much more read activity than write access.

Read Requests

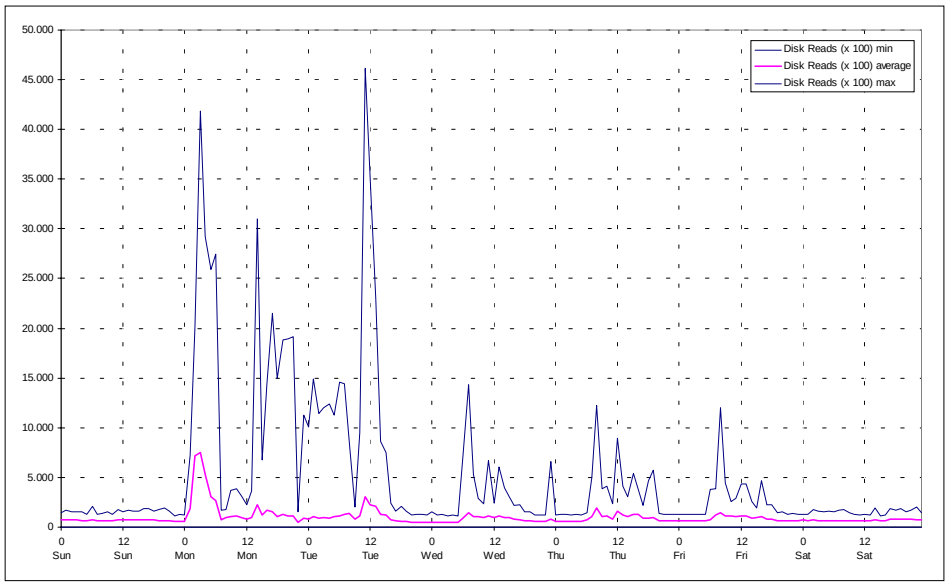
What one might expect regarding other measured data Monday morning is the time of highest read activity. Beside that, regarding the cache noon is the time of most activity (despite the fact that many connections are already dropped due to lack of activities). Sundays maintain the same level of disk activities than a normal work day.



The number of active users does not have as much impact on the disk activities than other parameters.

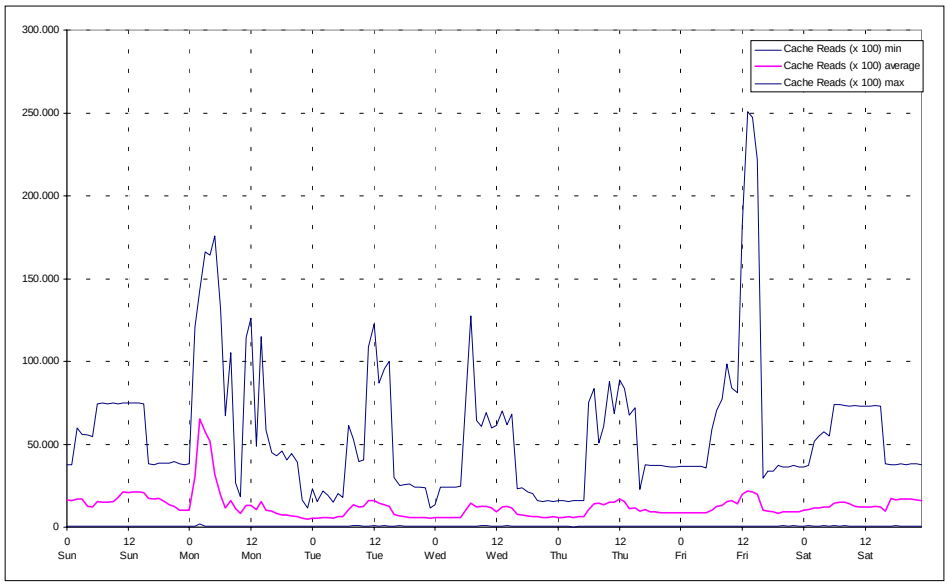
Disk Read Access

The cache reduces the actual disk reads by a factor of 10.



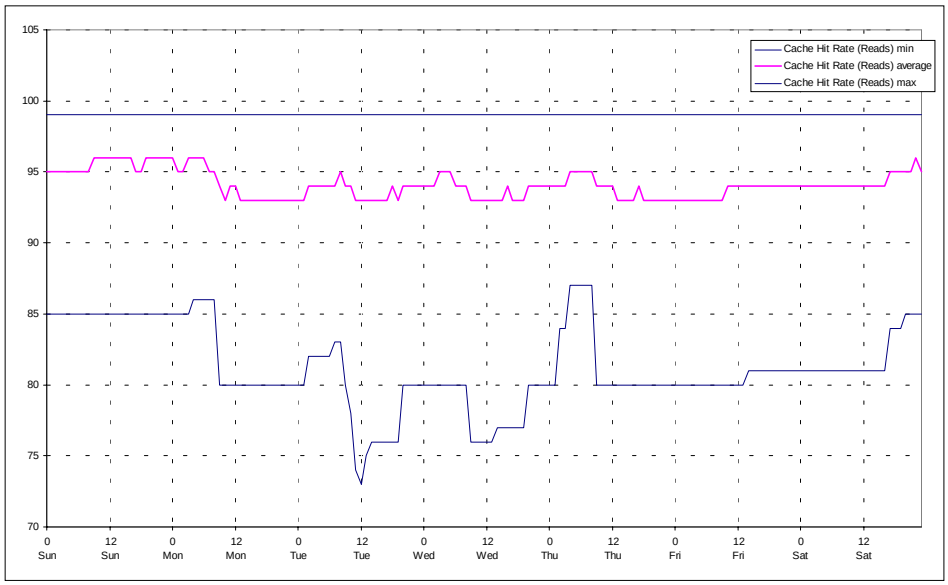
Cache Reads

Of course, the rest of read requests are directly handled by the cache.



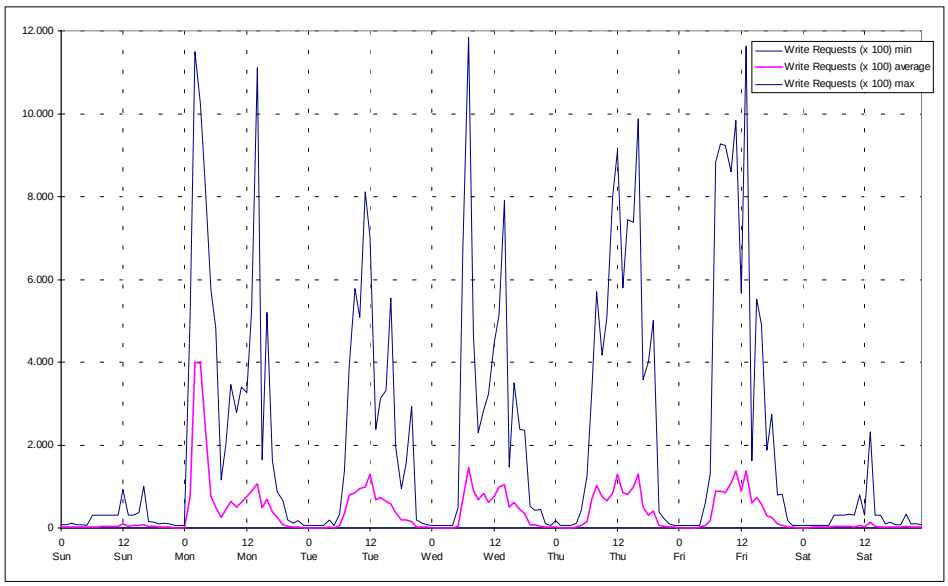
Cache Reads Hit Rate

On average the overall efficiency of the cache was about 95%. Most of the time it is not below 80%.



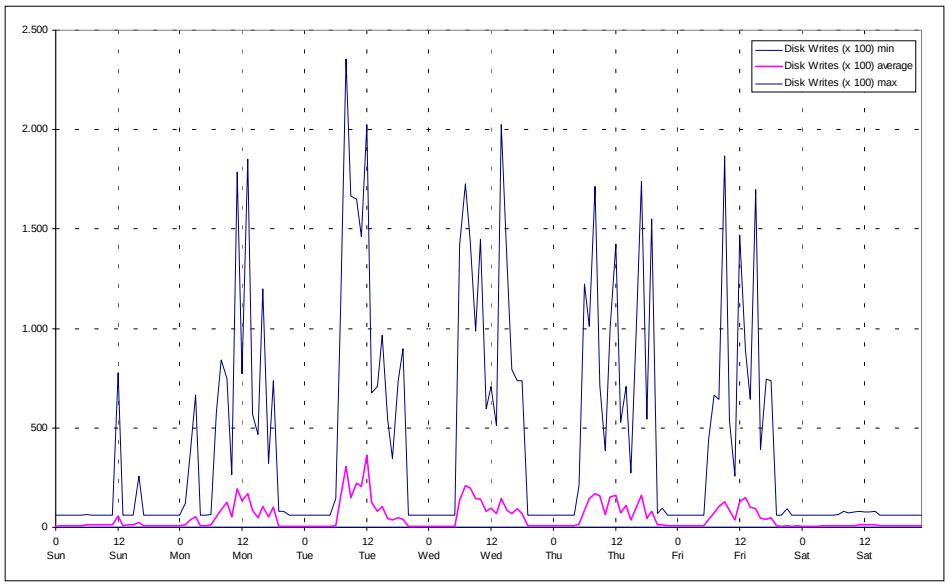
Write Requests

Again Monday morning is the time of most activities.



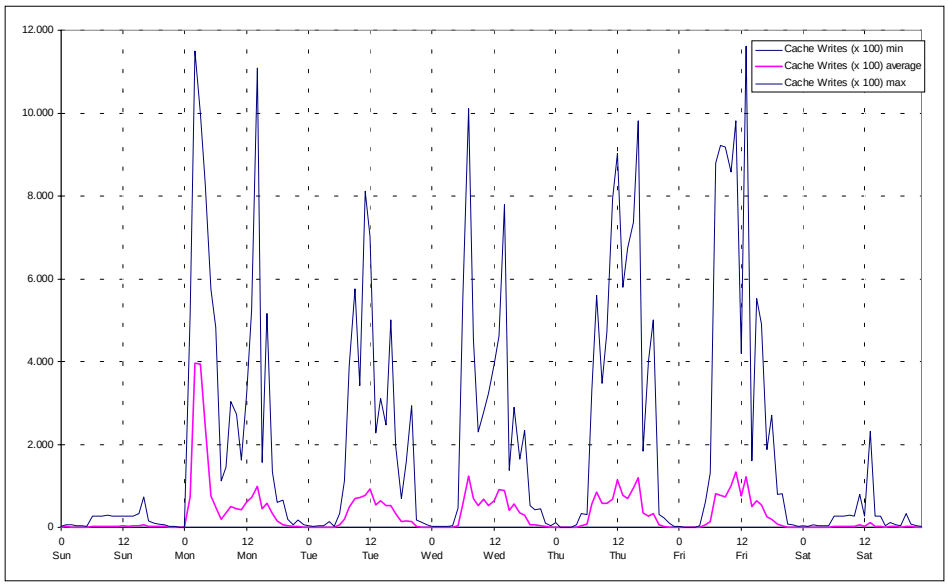
Disk Write Access

Out of 1000 requests about 100 trigger actual disk access. The ration between requests and disk operations is 1:10 (like in case I). Note that the peak load on Monday morning is not reflected in the graph below.



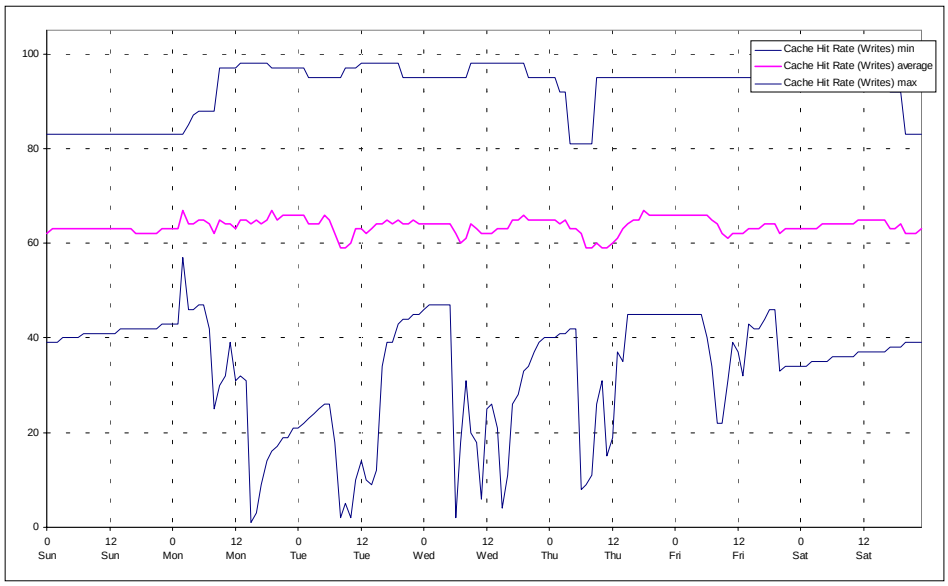
Cache Writes

Most of the requests can be handled by the cache.



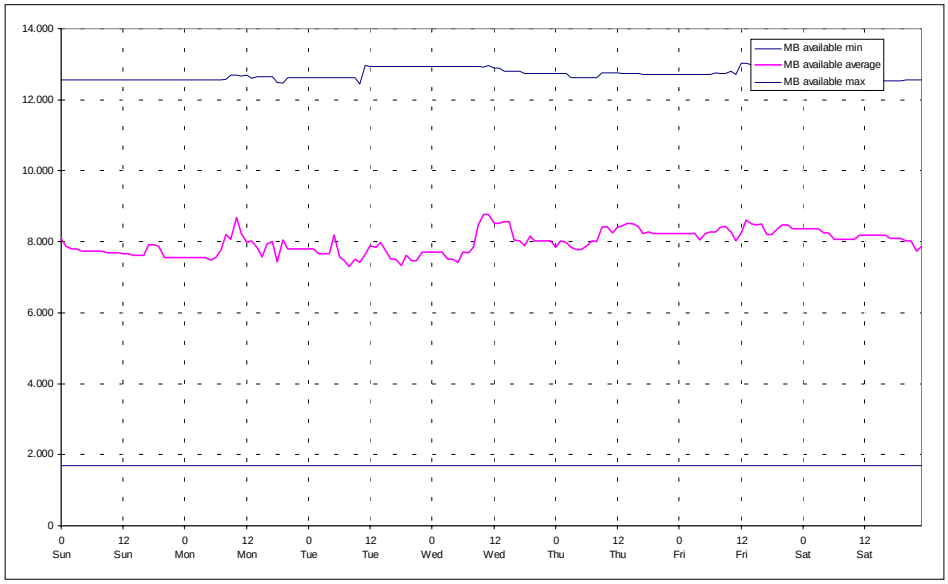
Cache Writes Hit Rate

Although most of the requests can be handled by the cache the file system reports its efficiency with an average value of about 60% (which would be less efficient than for read requests).



8.2.13. Diskspace Available On Servers

As there is not much variation in the amount of allocated disk space there is not much activity in the amount of available disk space. That is another sign for the fact that the server are not used to store much user data during daily work.



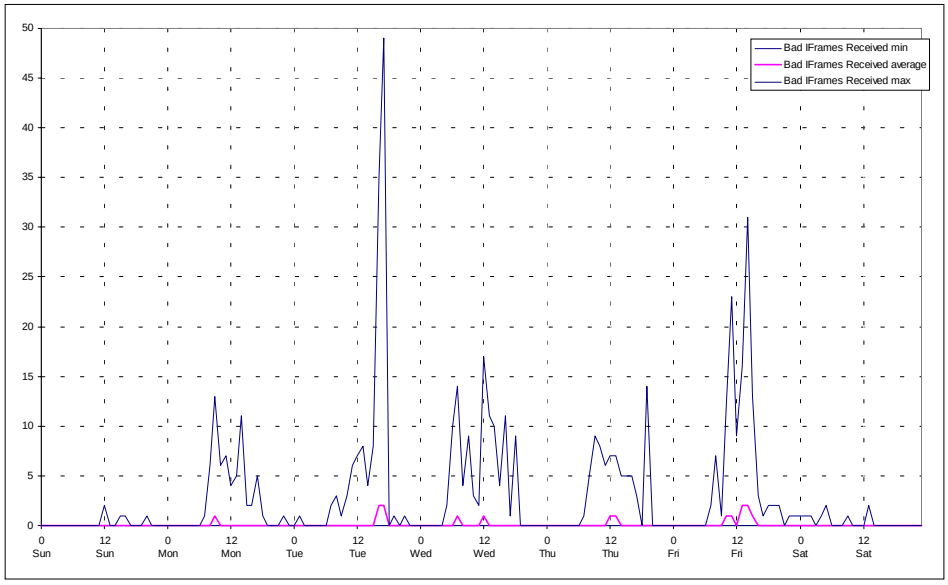
8.2.14. NETBIOS

Note, that only five important servers where considered for the following detailed measurements. See chapter 8.1 for an explanation of the resource attributes.

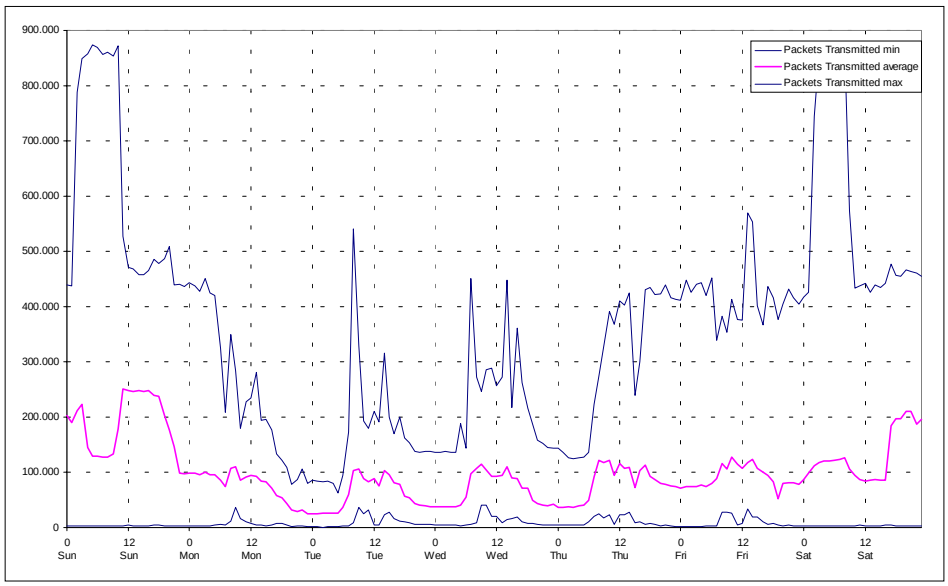
NETBIOS - Frames Received

No Information about frames was recorded.

NETBIOS - Bad Iframes Received

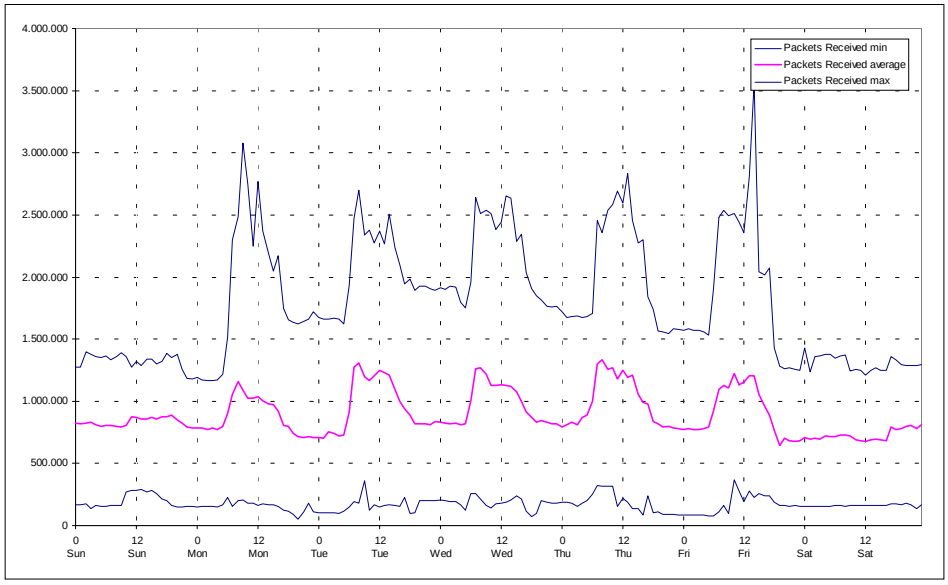


NETBIOS - Packets Transmitted

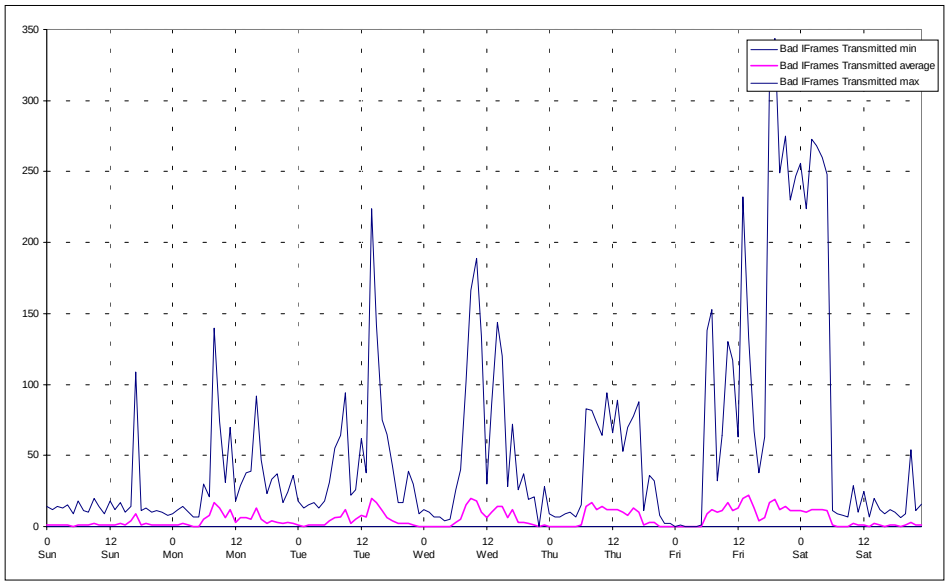


NETBIOS - Packets Received

We see a similar graph for the number of received packets.

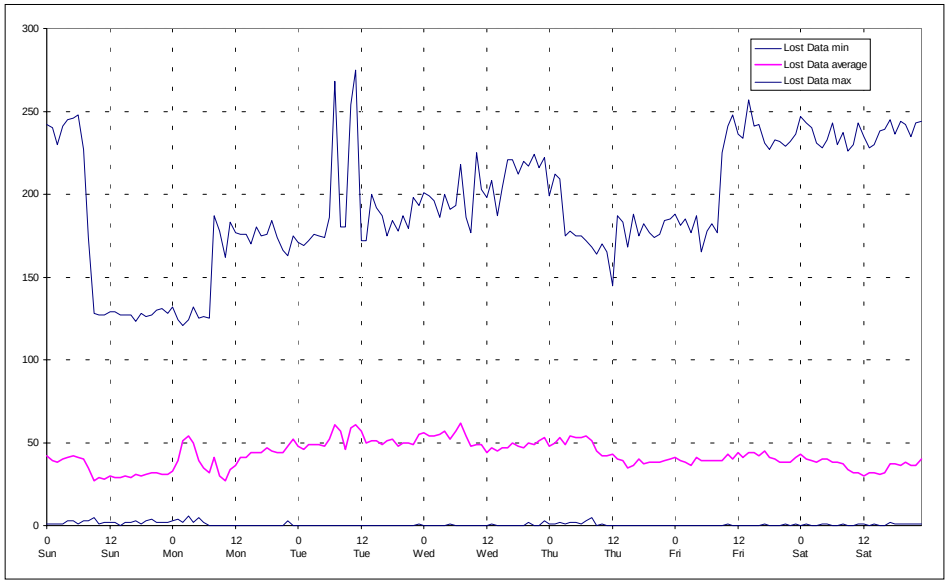


NETBIOS - Bad IFrames Transmitted



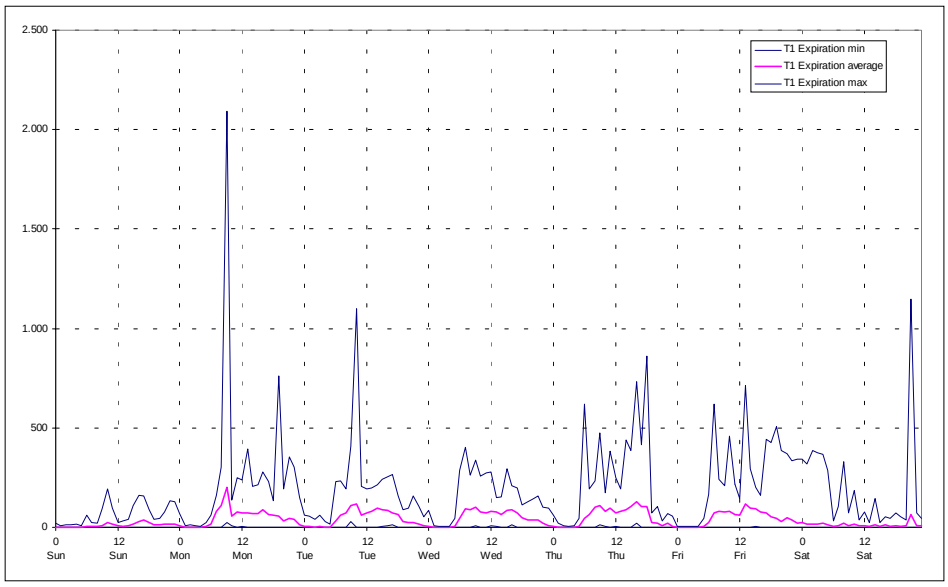
NETBIOS - Lost Data

There are not many errors due to network or application problems.



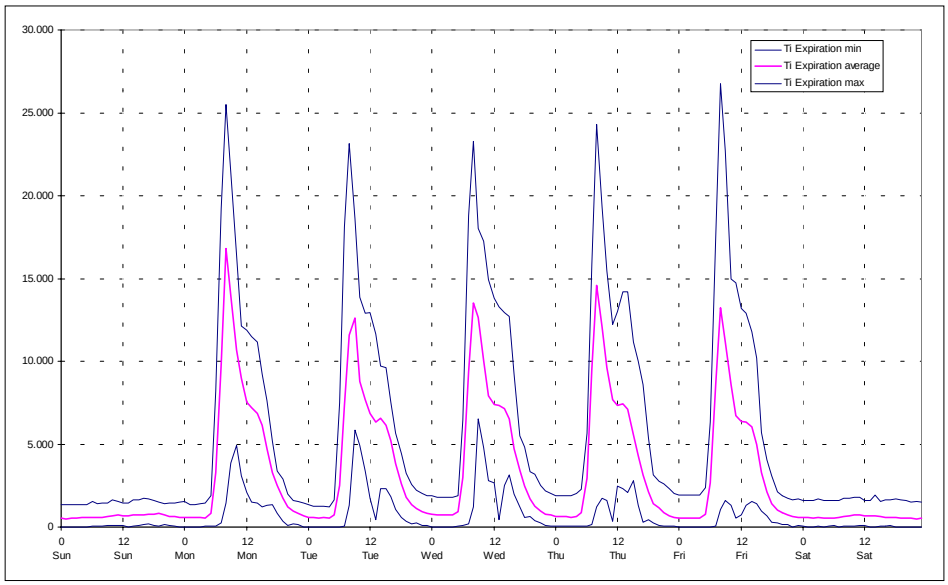
NETBIOS - T1 Expirations

The T1 timer checks whether a request for setup or resume of a connection is serviced within time. The number of expirations correlates to the activities on the network.



NETBIOS - Ti Expirations

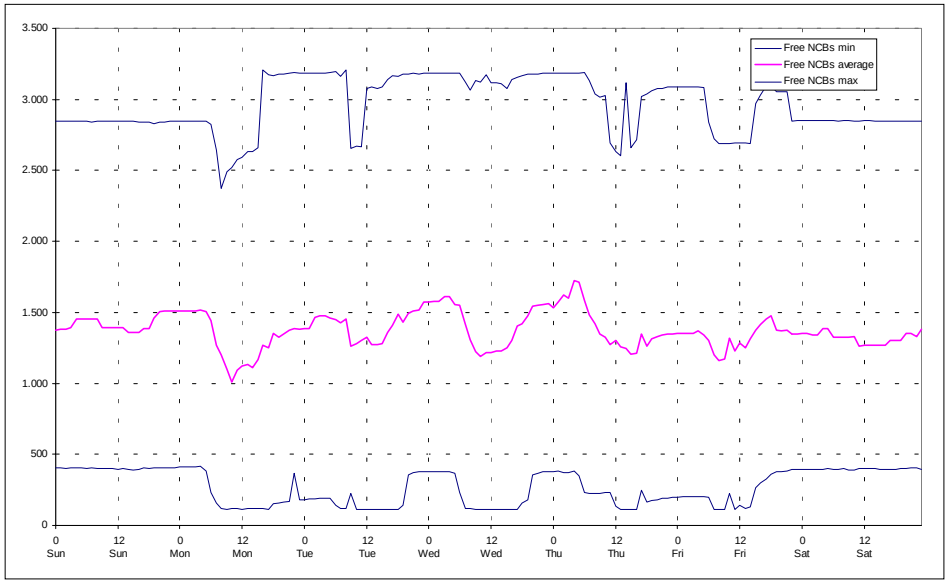
The Ti timer checks the activity on an active connection. If it expires due to lack of activity the network software will check the connection. The number of expirations correlate with the number of sessions (and connections) to the servers. This contributes to the network traffic and generates some of the traffic during idle times (non working hours).



These numbers clearly correlate with the number of connections.

NETBIOS - Free NCBs

As users connect to the servers network resources are allocated.

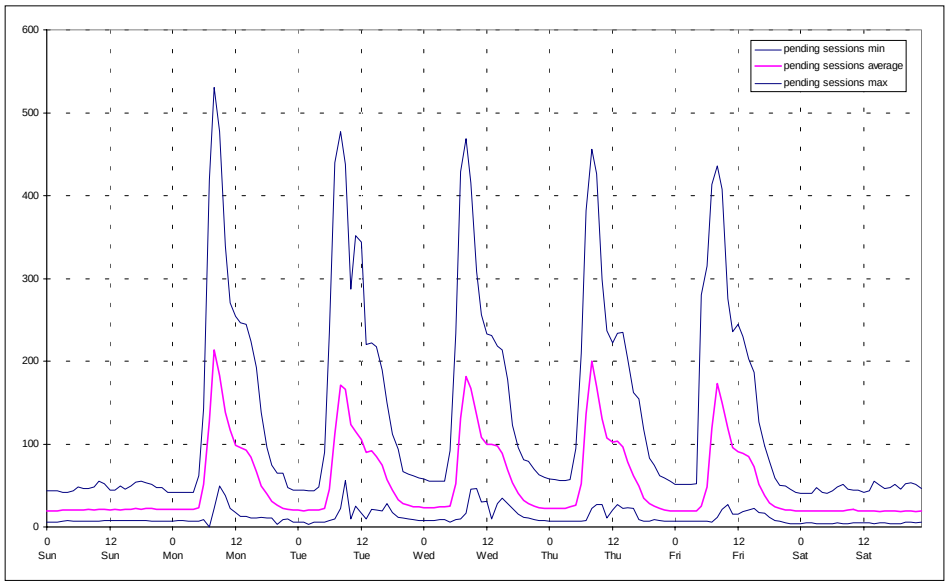


NETBIOS - busy conditions

There were no busy conditions detected.

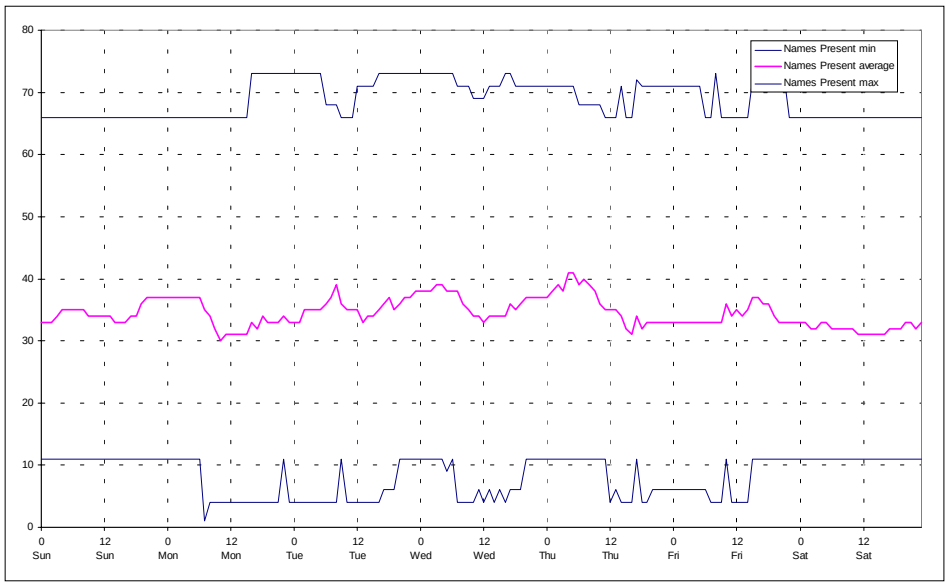
NETBIOS - Pending Sessions

The number of pending sessions clearly depends on the number of open connections.



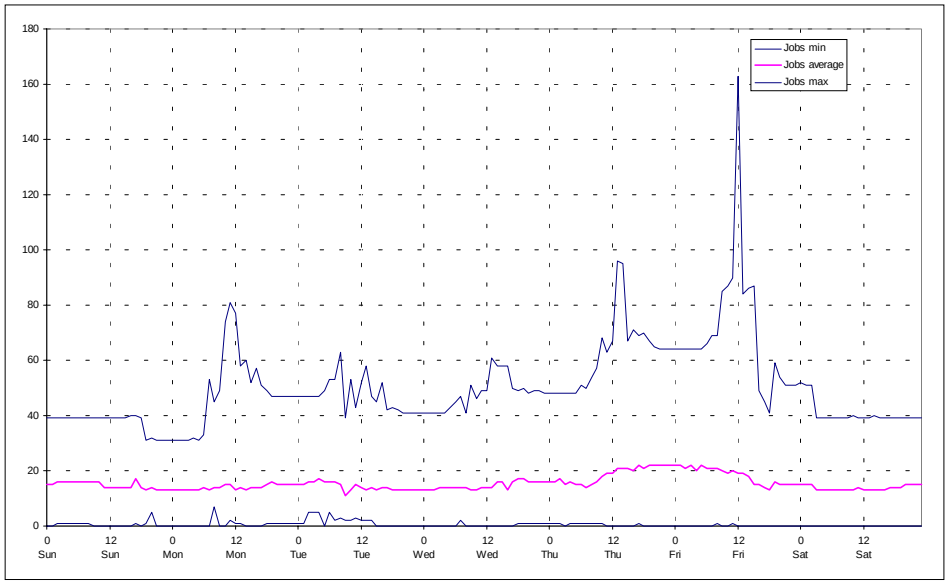
NETBIOS - Names Present

NETBIOS names are registered during startup of application and most of the time there is no dynamic allocation and deallocation of names (or other net resources).



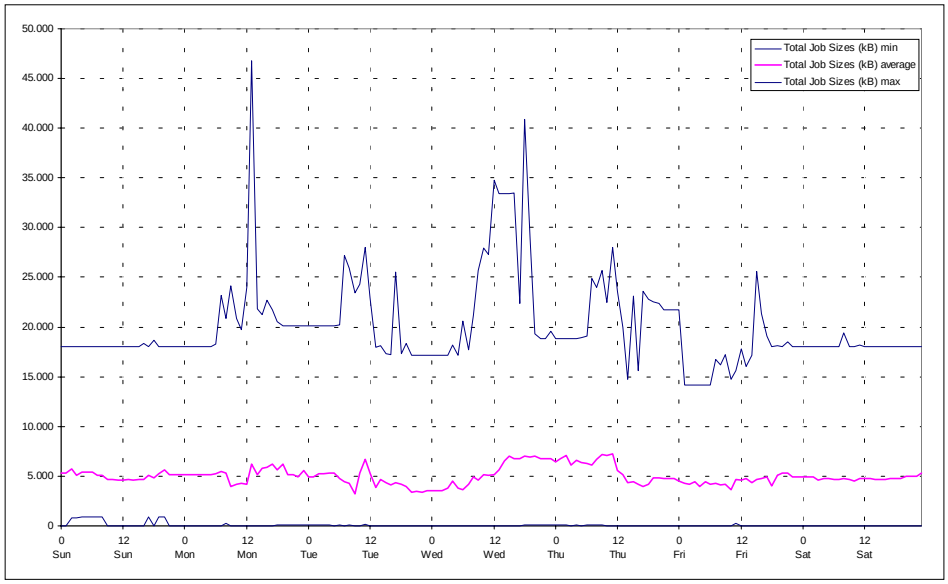
8.2.15. Number of Waiting Print Jobs in Spooler Queues

Other than in case I there are always a number of print jobs waiting; but this domain contains more than 100 print queues. The number of waiting print jobs does not depend on the number of active users.



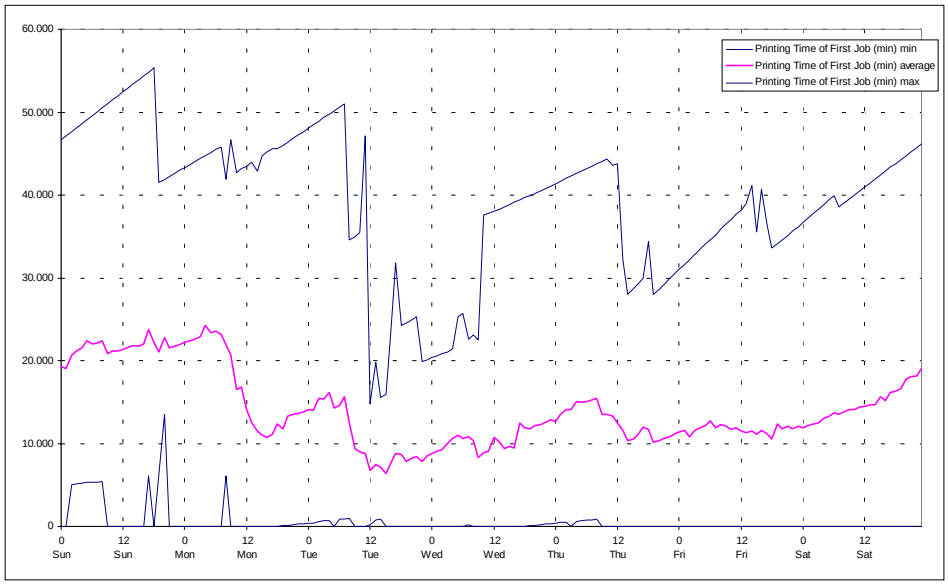
8.2.16. Total Job Size

The graph below shows how much data on average a spool server must be able to store at any time during one hour. We see that the actual maximum is at about 45 MB that were waiting to be send to printers. On average up to 8 MB must be stored in the spool area.



8.2.17. Printing Time Of First (Active) Printjob

The following figure shows the time that the first print job (the one which is about to be printed) is waiting in this position or - in other words - how long it takes to process the job. Normally this time is very short. Longer times indicate printer problems and the time spend in detecting and correcting the problem.

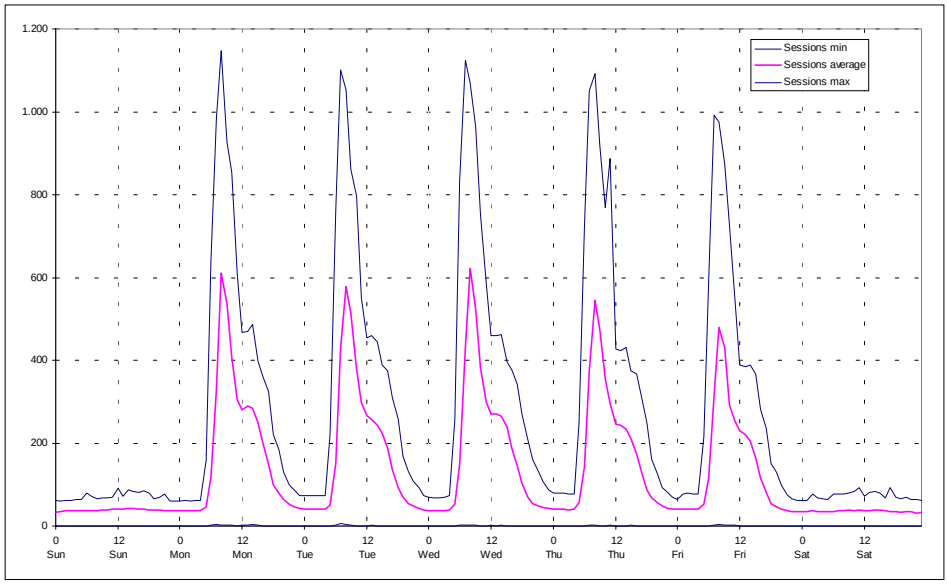


As we can see it may take up to 38 (!) days until the job is spooled to the printer (or is removed from the queue). Even the average during the week is 250 hours. In that case the average is misleading because blocked jobs waiting in seldom used spool queues lead to the numbers.

Our experience with end users is that, when they come over a problem with the spooler queue, they try to use another printer and do not report the problem to the responsible person. Another observation is, that - for an unknown reason - many user do not care for their print output and many print-outs on PC-printers are never picked up by their owner.

8.2.18. Number Of Server Sessions

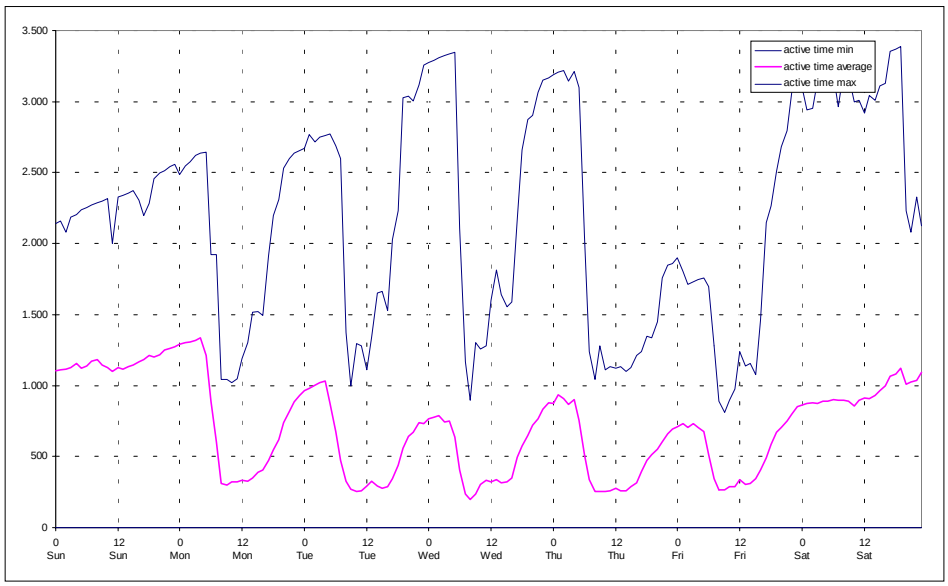
Note that this value is practically identical to the sessions which are reported by NETBIOS. If servers with additional server applications (like DB2 or Lotus Notes) would be part of the domain, these values could differ.



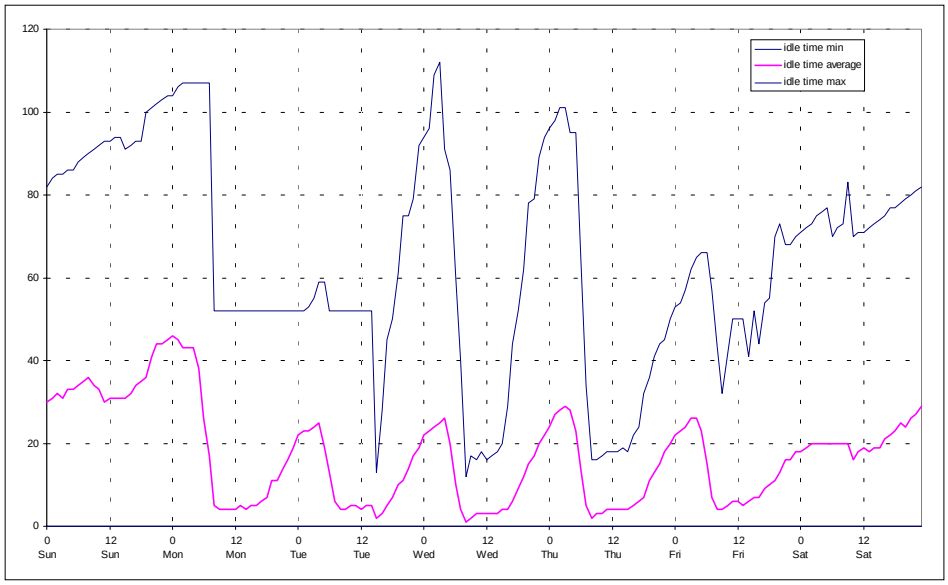
The difference in absolute values comes from the fact that this values are reported by the LAN server agent and therefore all servers in the domain were monitored (remotely) and account for the total numbers. In case of NETBIOS the local agent must be active and this component was not installed on all servers of the domain.

8.2.19. Active Time Of Server Sessions

The graph reflects the age of server sessions. Machines that are not switched off at night or over the week end lead to the high numbers.

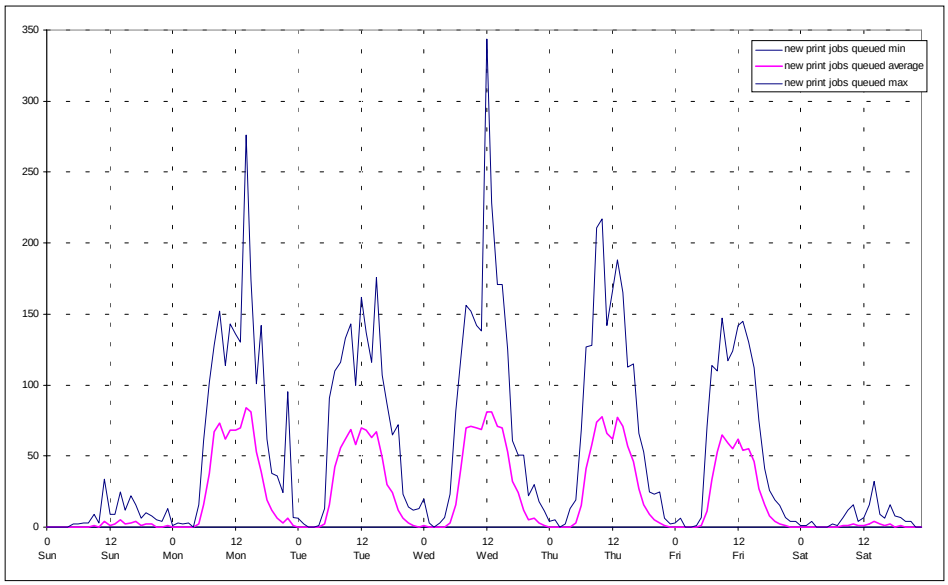


8.2.20. Idle Time Of Server Sessions



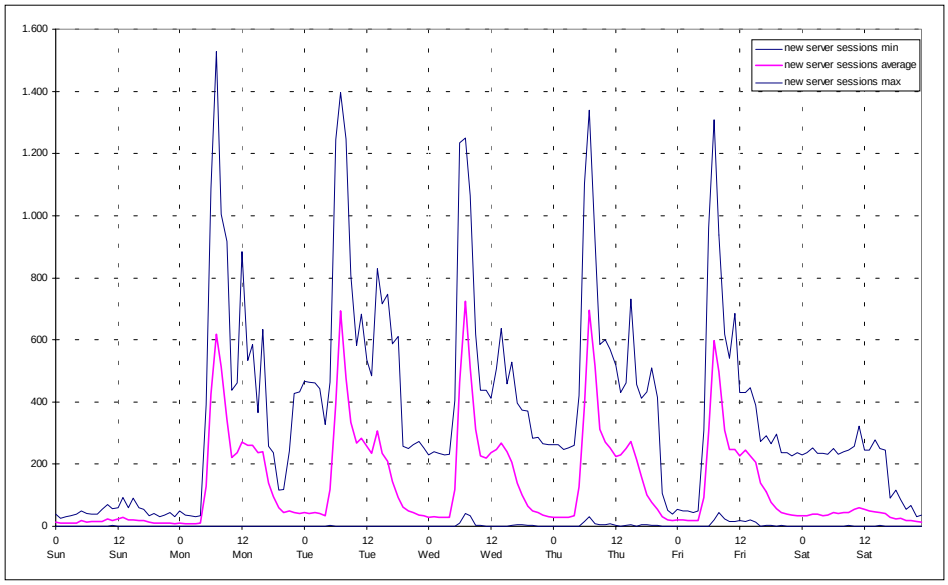
8.2.21. Printjobs Queued To Servers

The next graph shows the number of print jobs that are submitted to a spooler queue during one hour. On average the daily peak load is about 60 to 70 jobs per hour. From the server's point of view that is not much but it may be too much for the connected printers.

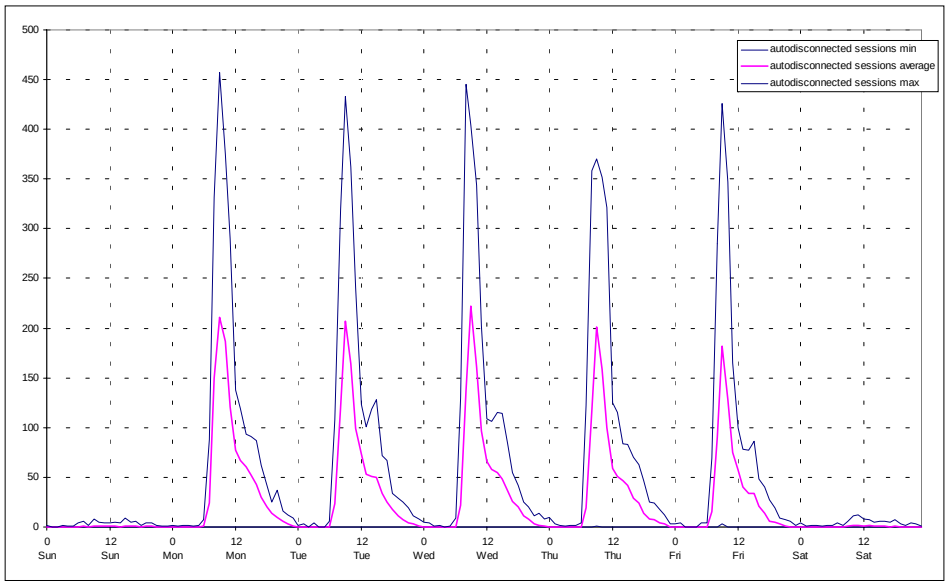


Note, that it is not possible to monitor the number of pages which have to be processed by the printers. Maybe, that would be a better measurement for the load on the printers than the number of jobs.

8.2.22. New Server Sessions

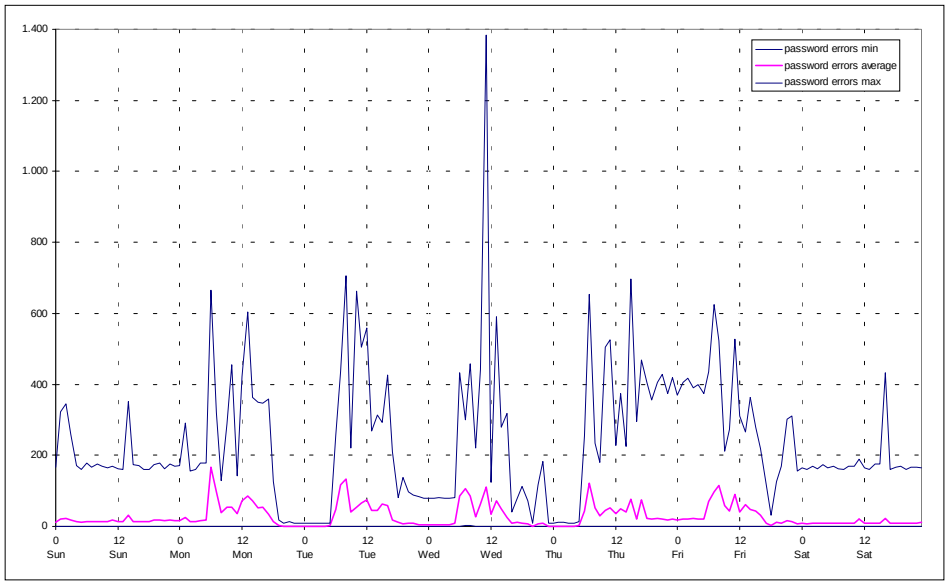


8.2.23. Autodisconnected Sessions



8.2.24. Password Errors

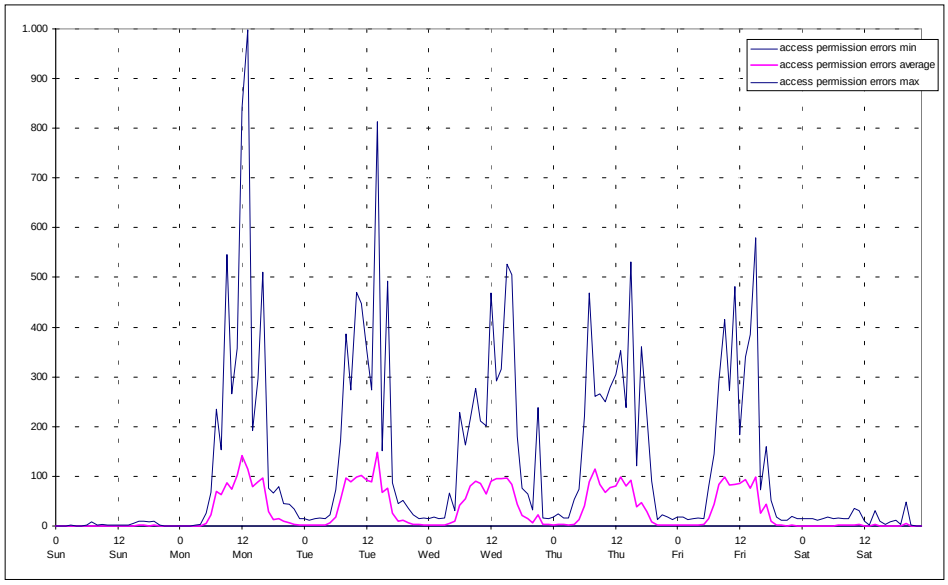
It would be interesting to understand the high number of password errors during non-working hours. A very likely reason for that may be the case if cross domain user definitions get out of sync. In that case users can generate password errors without any notice.



Obviously it is not possible to distinguish the case when a user types an incorrect password or when a cross domain password mismatch happens. The consequence is that administrators tend to ignore such information.

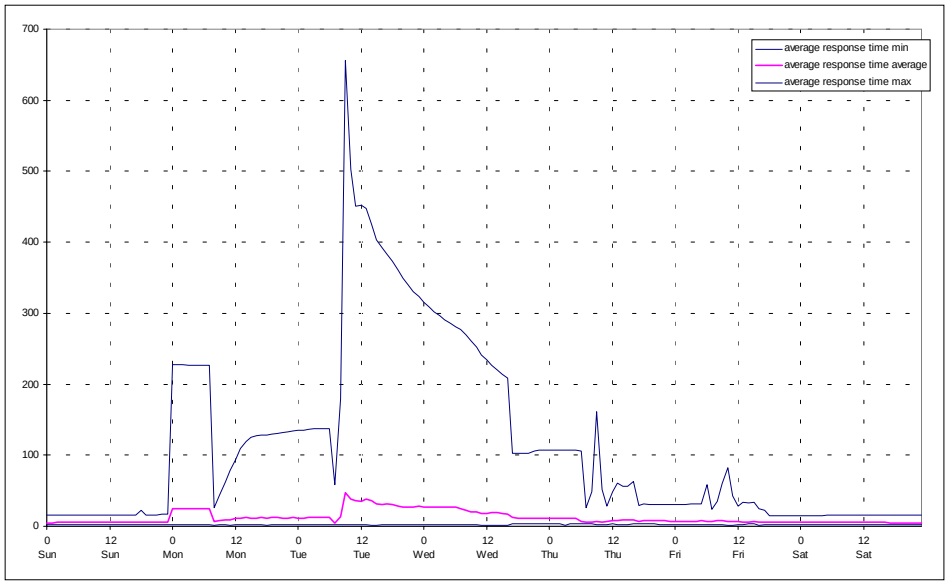
8.2.25. Access Permission Errors

A possible reason for high numbers of access permission errors are automated procedures which try to access a resource (e.g. to check their existence or the access permission of the user which uses the program).

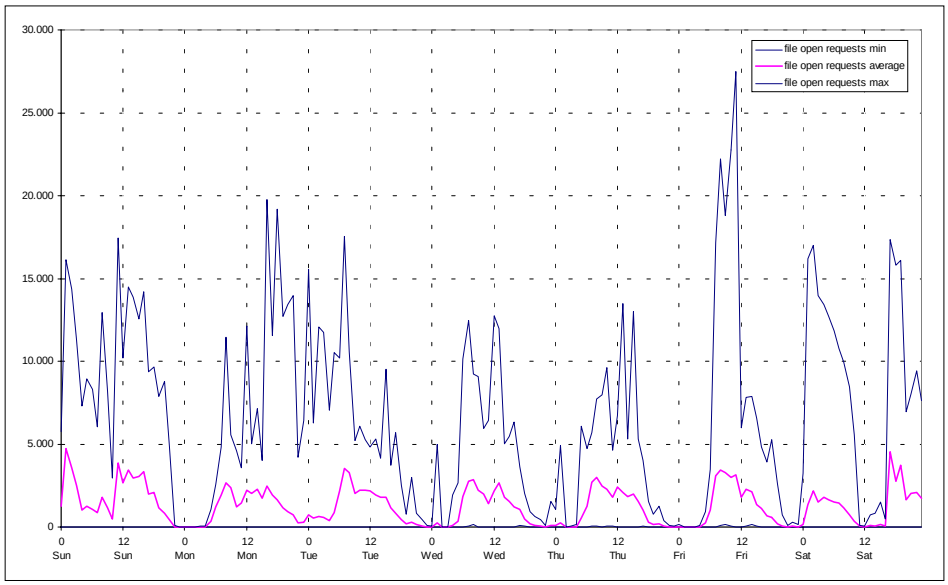


In contrast to the number of password errors access permission errors depend on the number of active users on the system.

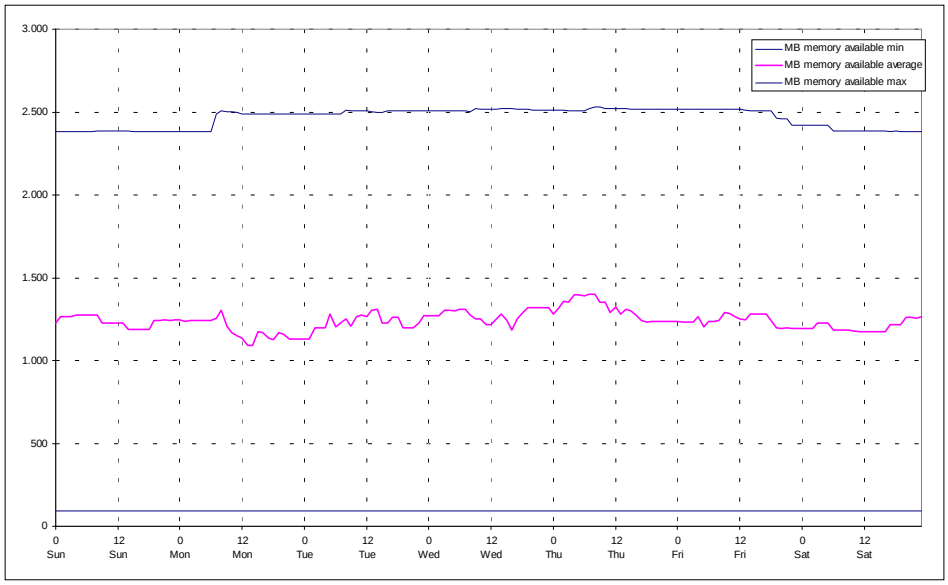
8.2.26. Average Responsetime



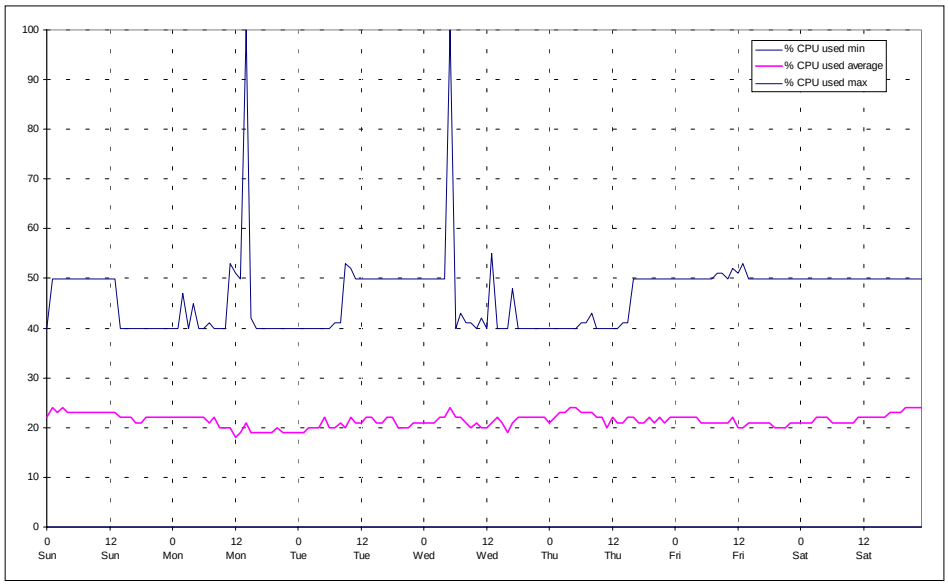
8.2.27. File Open Requests



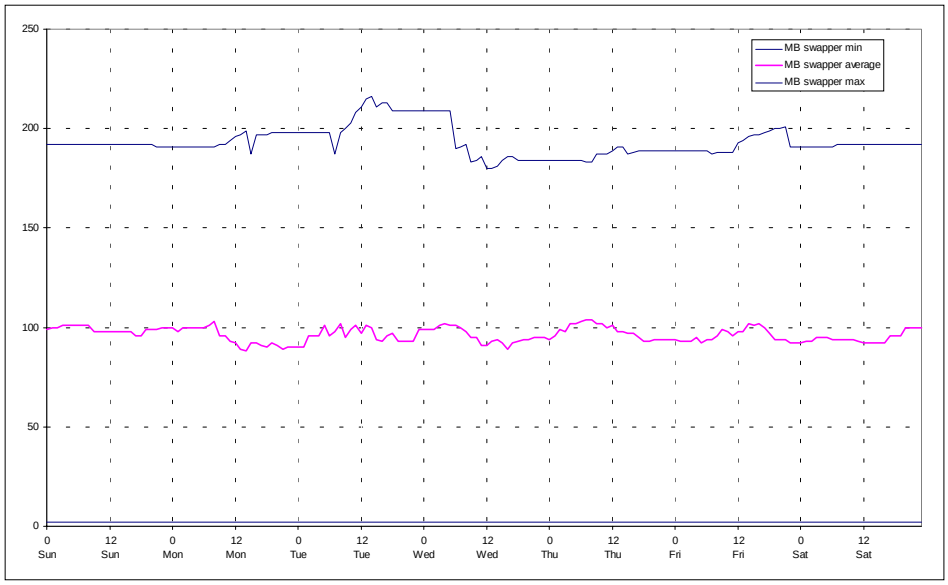
8.2.28. (Virtual) Memory Available



8.2.29. CPU Usage



8.2.30. Size Of Swappfile



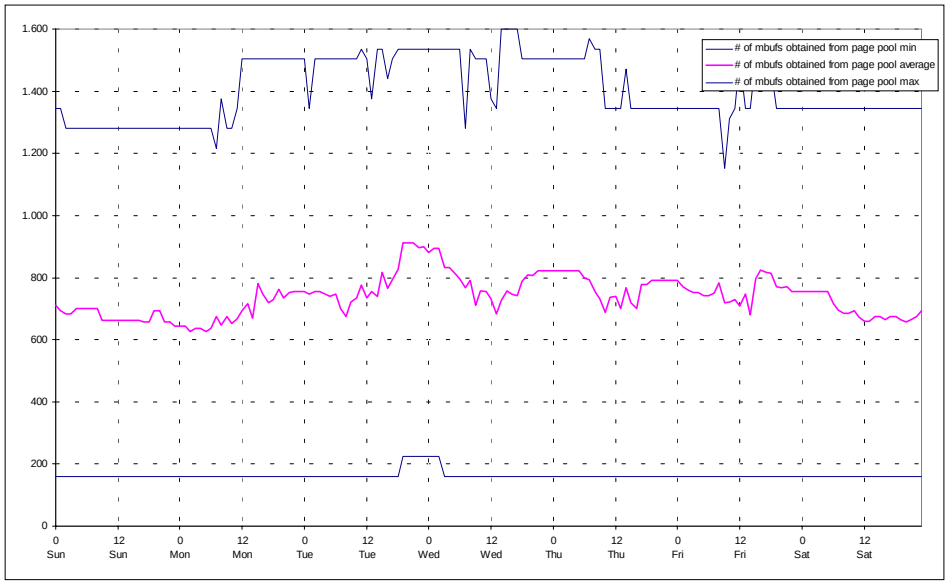
8.2.31. TCP/IP Statistics

A selection of meaningful out of more than 70 available measured resource attributes are shown on the following pages. The TCP/IP implementation on OS/2 provides much more information than NETBIOS although it does not provide an API for applications like the monitor.

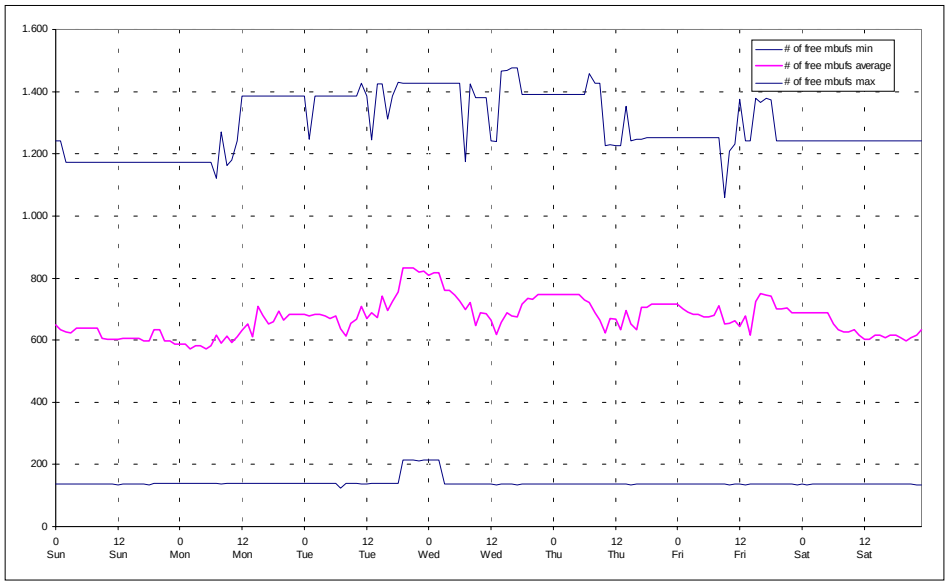
In general the use of TCP/IP resources is more dynamic but often does not correlate to the number of users. One important reason for that is, that TCP/IP is not used by the LAN server application. TCP/IP activities originate from other tasks mainly systems management activities.

# of mbufs obtained from page pool

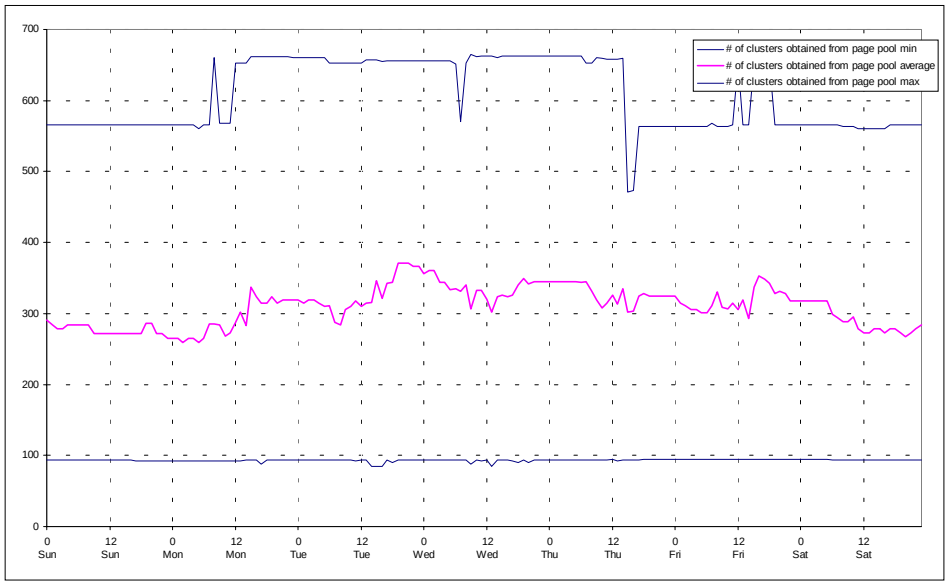
"mbuffer" are an example of internal resources. The network software needs buffers to temporary store information. Lack of buffers may decrease the performance. Managing to many buffers may decrease the overall performance of the machine as well.



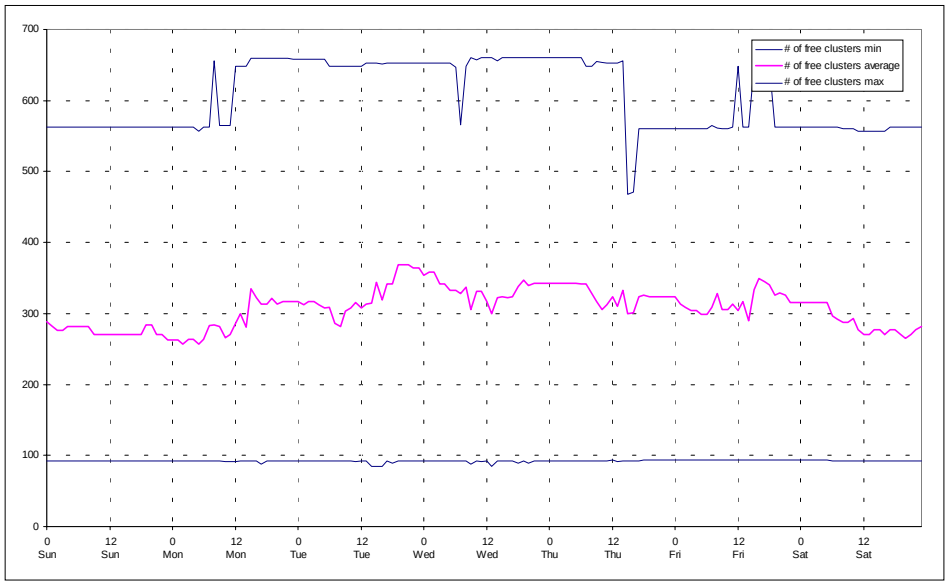
# of free mbufs



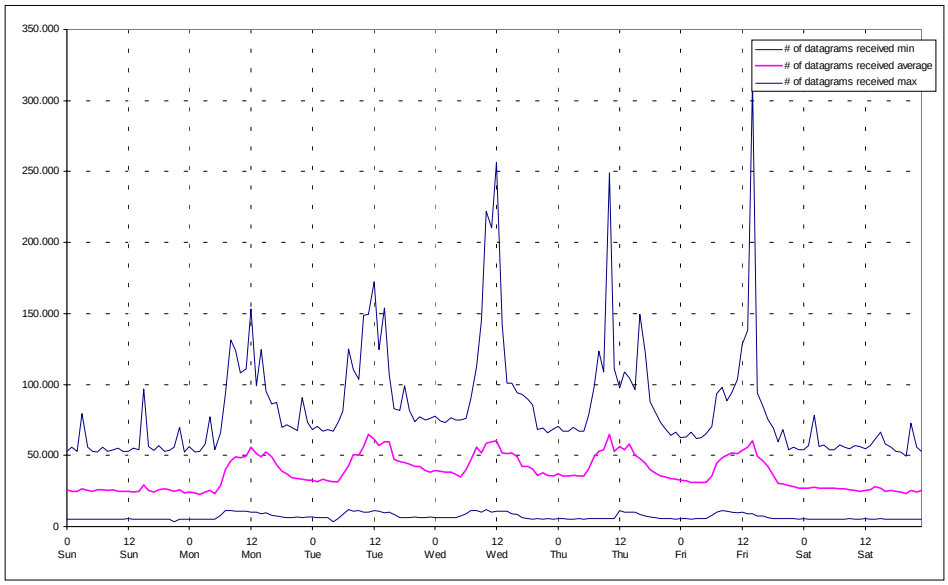
# of clusters obtained from page pool



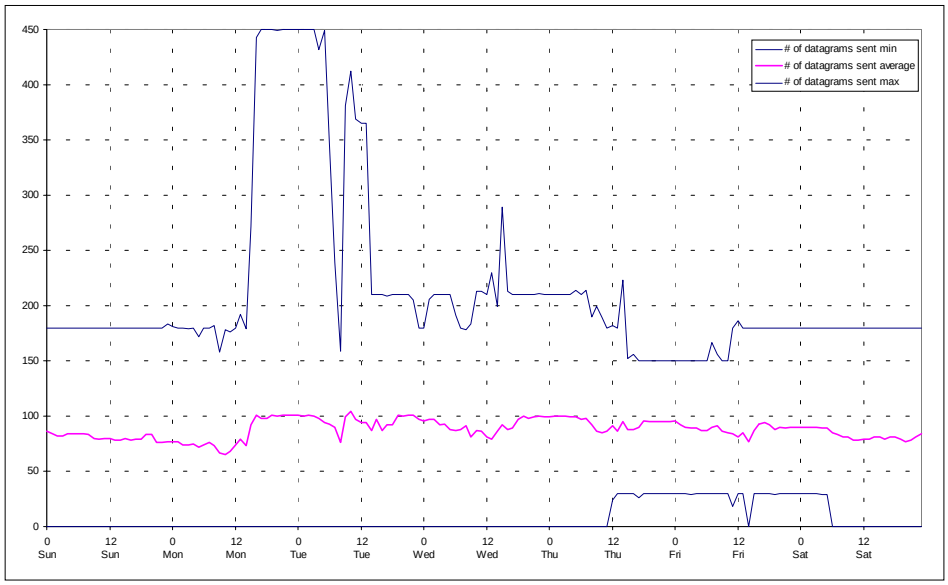
# of free clusters



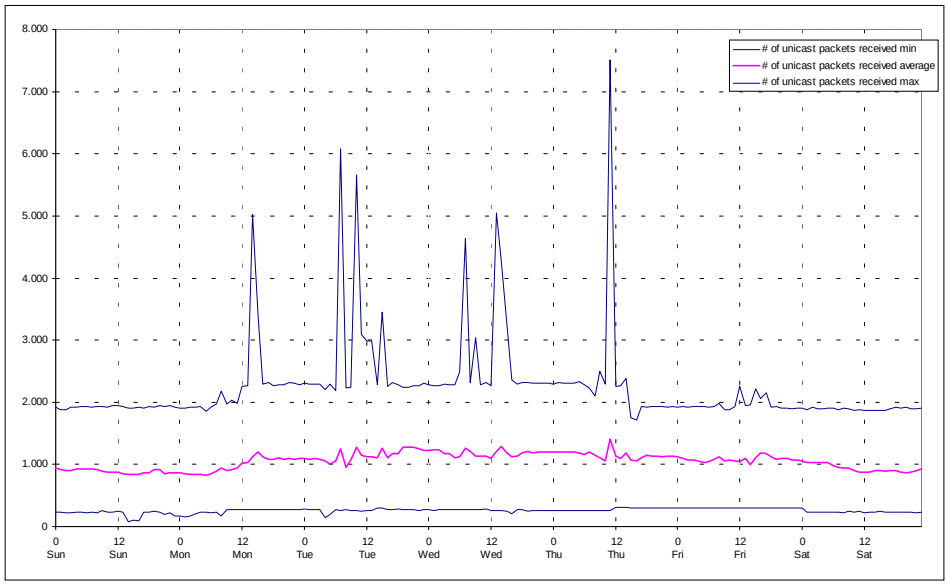
# of datagrams received



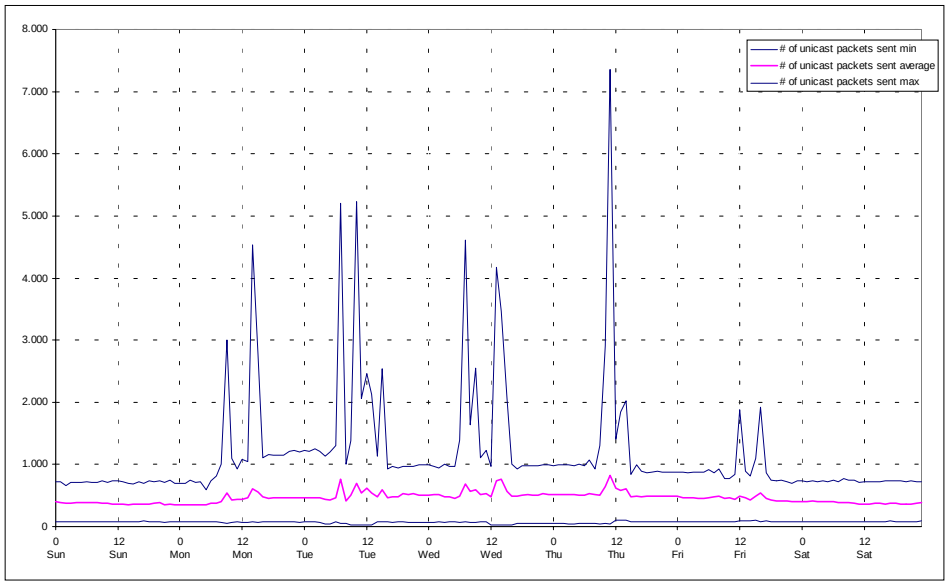
# of datagrams sent



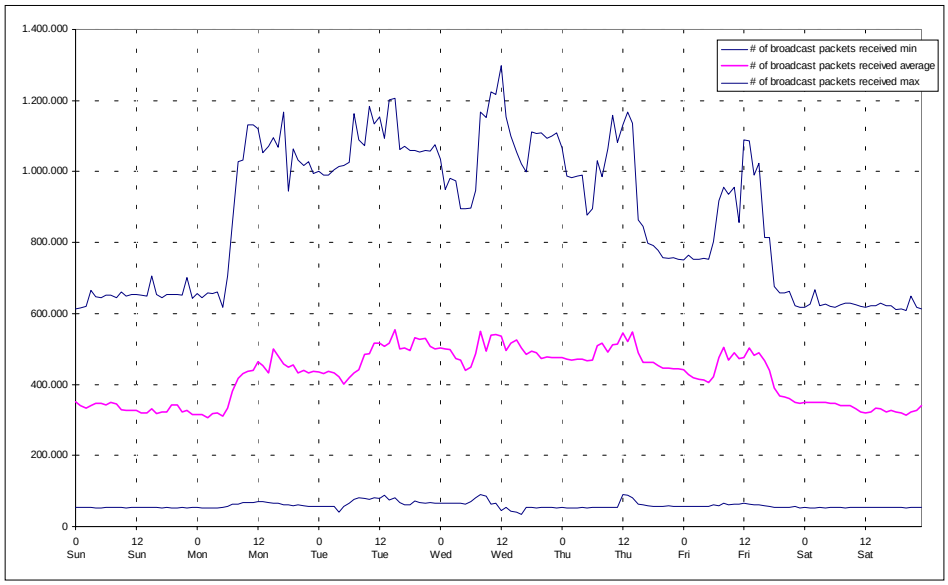
# of unicast packets received



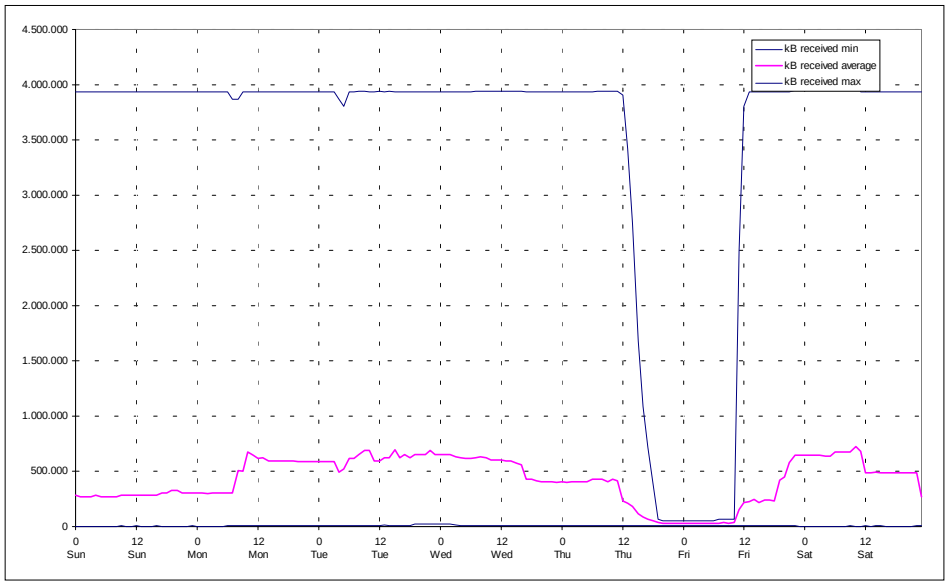
# of unicast packets sent



# of broadcast packets received

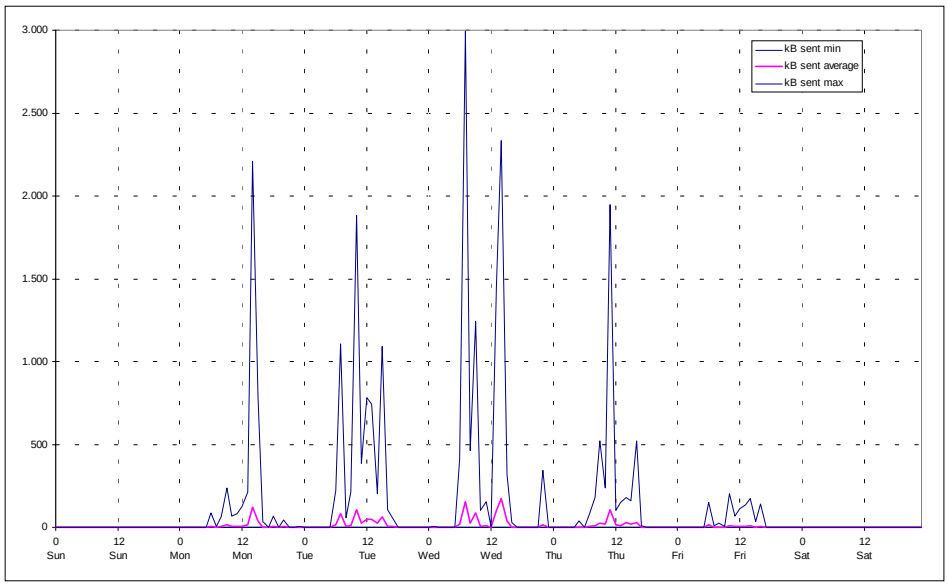


kB received



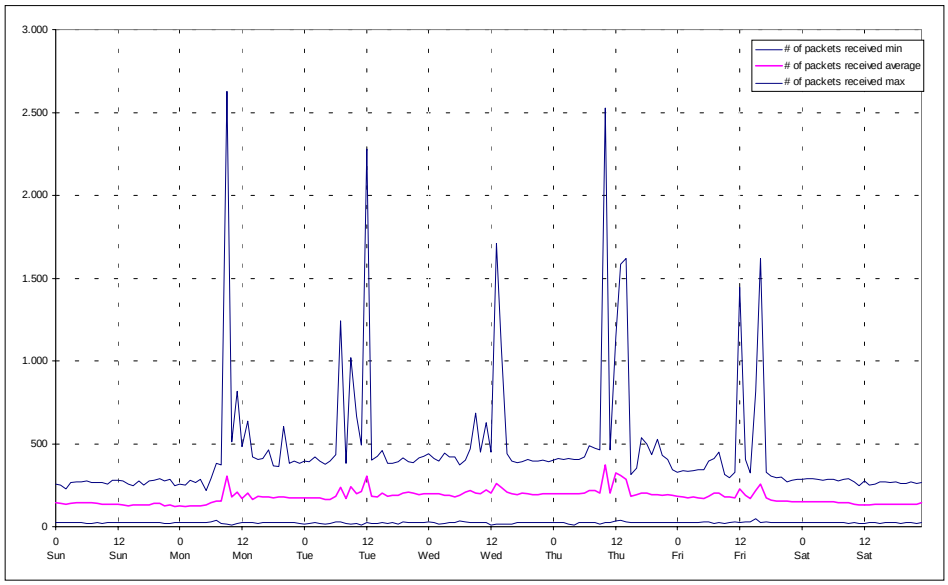
We assume that the high maximum values results from a invalid instrument readings.

kB sent



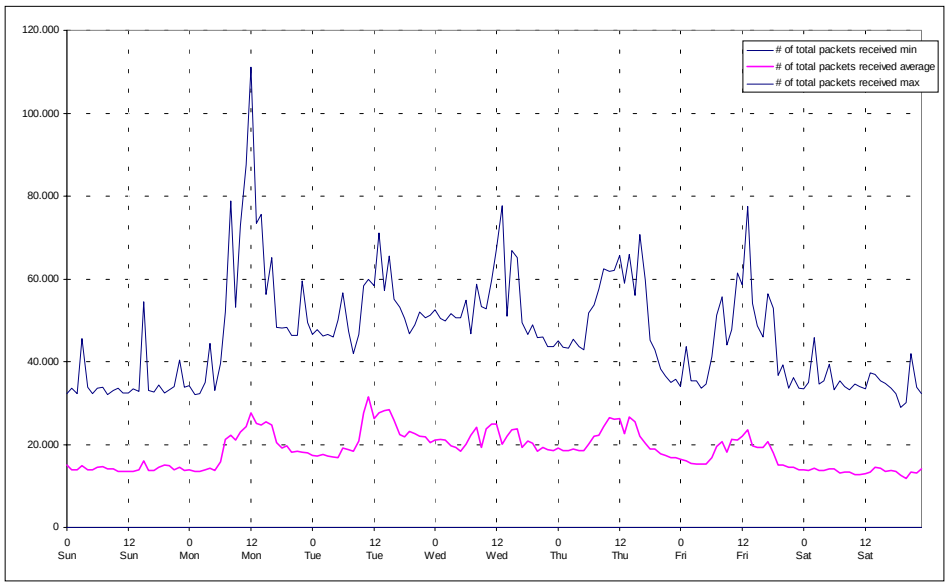
# of packets received

This is one of the major indications for network traffic related to observed machines. We can see that there is significant traffic all the time and that only a fraction of that traffic is added during office hours. We conclude that TCP/IP traffic is generated by all machines as long as they are active independent of user activity.

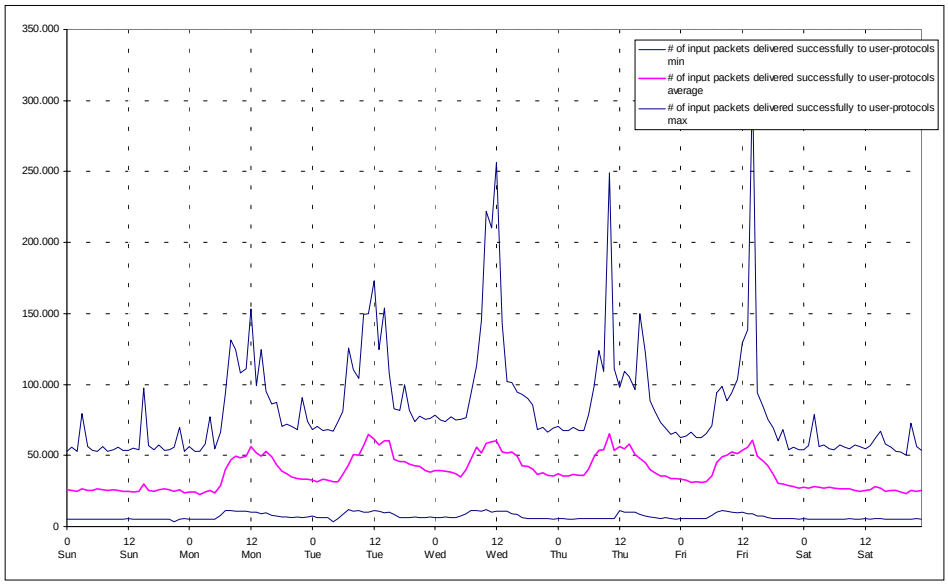


# of total packets received

The TCP/IP implementation distinguishes several types of packets. This number includes traffic of all types. Again we can see that most of the traffic is not related to work hours.



# of input packets delivered successfully to user-protocols



8.3. Results of Automated Association Detection

The following chapter contains the result of a very extensive association analyses which was performed with the data from different case studies. The main idea was to detect dependencies between servers or to identify similar behavior due to certain conditions in the network.

We added mainly those relations which happened to appear in all studies. Some data must appear as an association because they directly depend on one another (for example,. “amount of disk space allocated” and “percent of disk space free”). Such obvious associations are not documented in this paper.

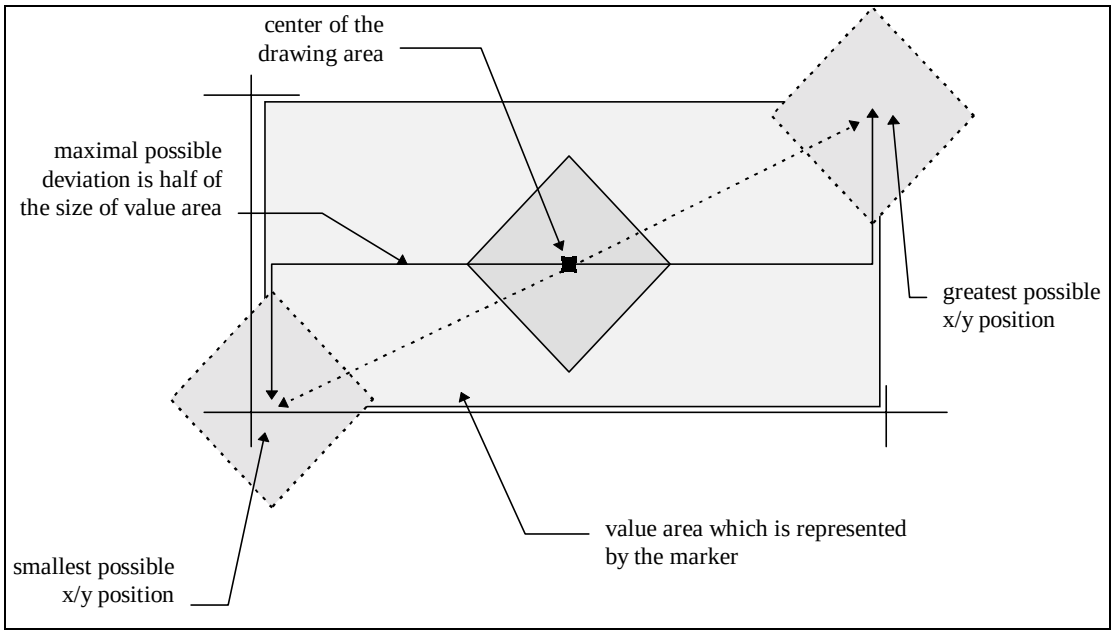
General Observation

Over the time we found no sensible associations of data from different servers. From that we assume that in the observed environments there are no direct measurable interdependencies between servers. But there are exceptions to this assumption. This exceptions point to a similar behavior or load under certain conditions but not to dependencies.

Case II (Domain FAW95)

This chapter contains the pictures for the graphical representation of some of the relations which can be found with the analyses tool. We use the form of scatter graphs which have been mentioned several times in this paper. The tool is written under OS/2 and is able to display the relations in a window. This window uses colors to show the number of hits on a certain pair of coordinates. Because there is no way to depict the OS/2 windows within this document - beside the creation of bitmaps which would be far to space consuming and would overflow the capabilities of the word processor - I used a different way to create the pictures:

For a relation of interest the pairs coordinates (x and y values within the two dimensional grid of the data items under inspection) are written to a file. Usually there are many hits for the same coordinates. In order to give a feeling for the density of hits in a certain area the exact position of each value is slightly modified by random numbers. Therefore it is possible to see dense heaps of many heaps in certain areas of the graphs.

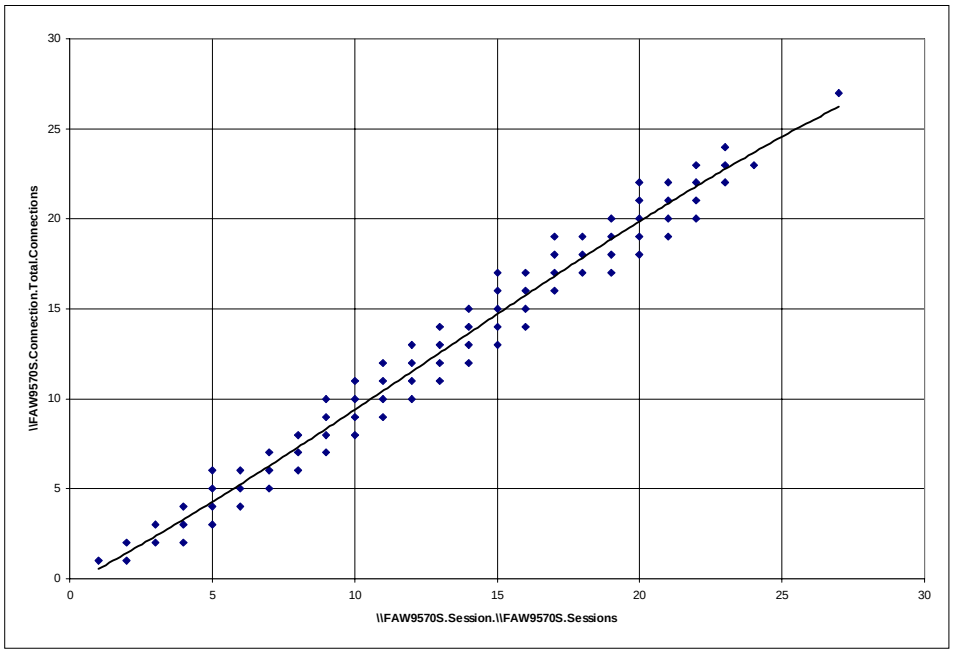


In addition to the values, which are calculated by the analyses tool, a trendline was added to the pictures. The trendline is calculated by the spreadsheet program and does not always fit the data that are represented in the graph. Due to this trendline sometimes negative coordinates are shown on the axis of the graph although no negative values can appear in the original data sets.

The following pictures are examples for what can be found with the analyses tool. Analyses for the sample domain usually detects several thousand possible associations. Many of them relate to specific servers or resources on that server. About 50% of the results are not real relations and can be ignored. For this paper only some of the associations were selected, which relate to the whole domain and which are clear examples of relations between different data items.

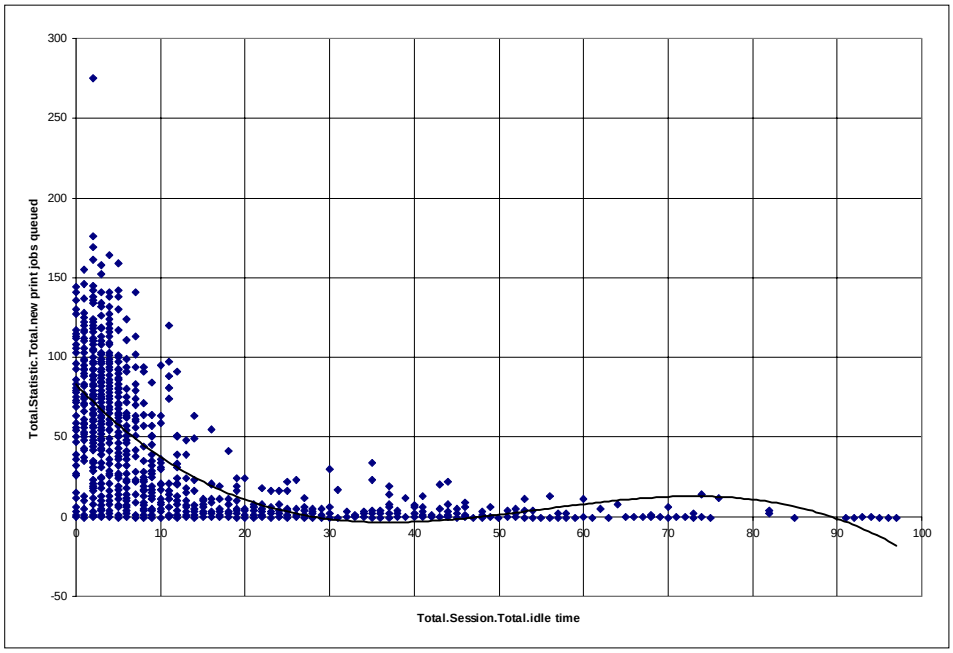
Connections vs. Sessions

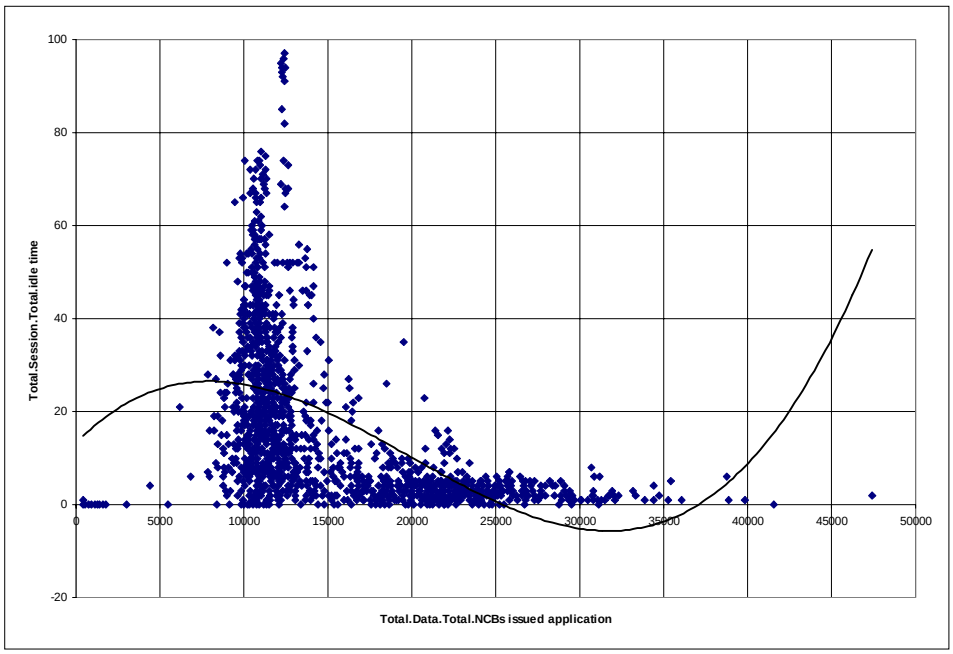
The next picture is an example of the linear association between the number of connections to a server and the number of active sessions.

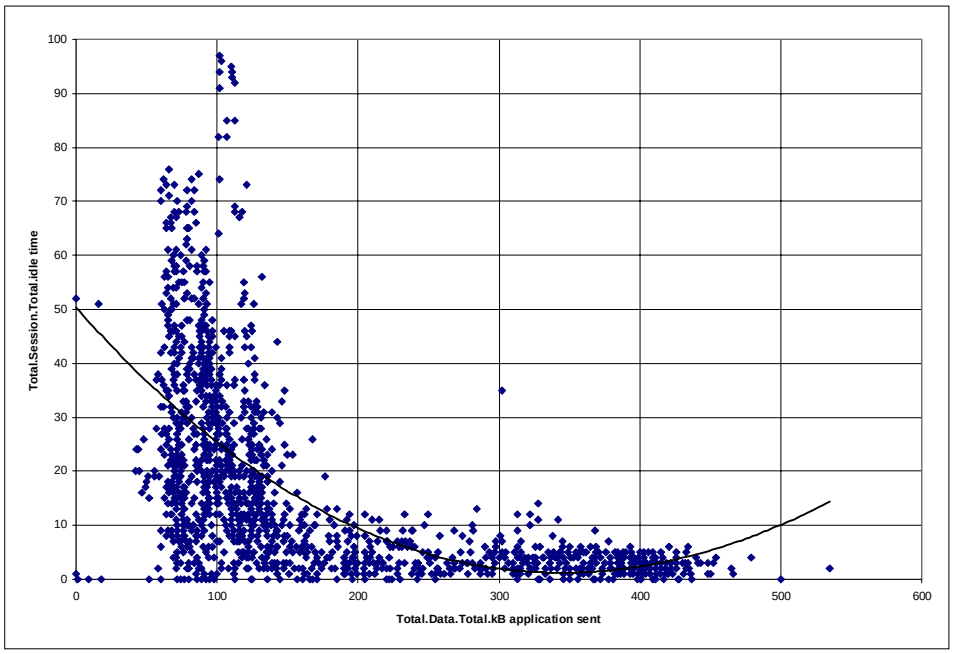


Session Idle Time

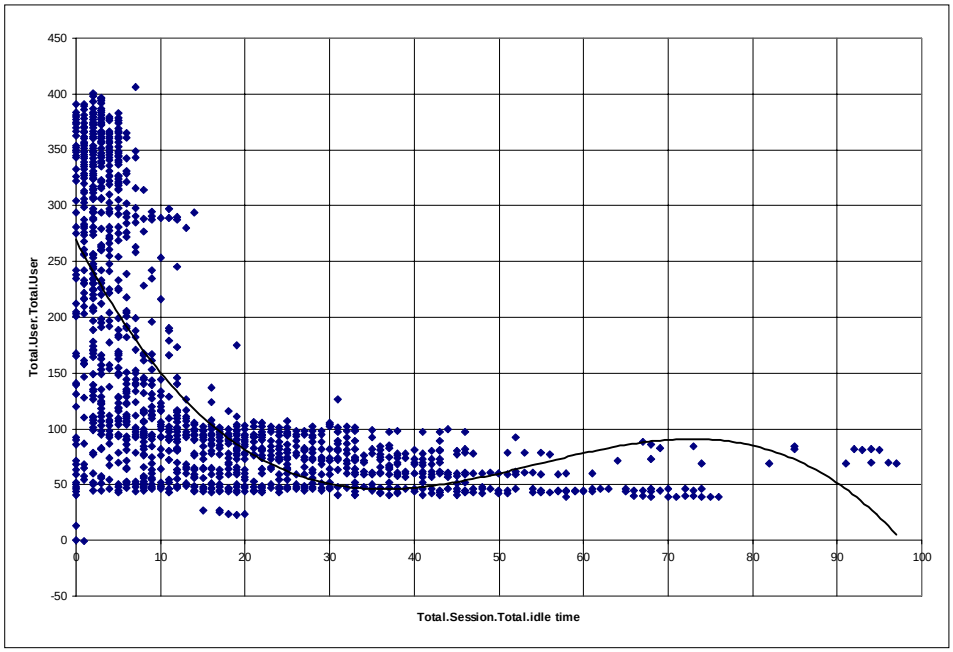
With increasing idle time of server sessions the usage of resources decrease. That comes natural as idle time is an indication for unused computers. There are several examples in the results. *Idle time* is a data item to which many certain associations can be detected.

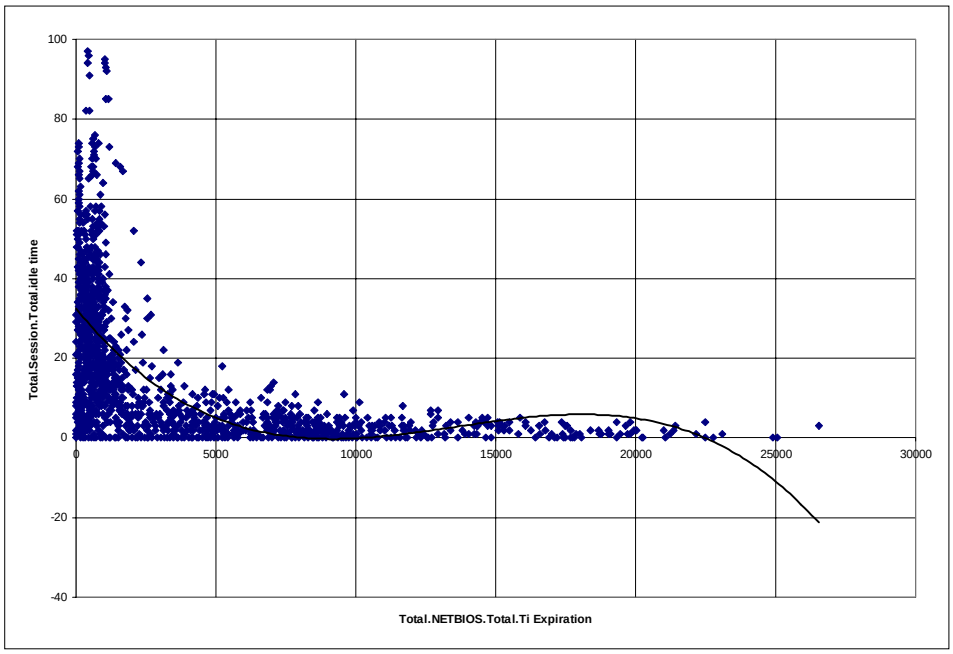


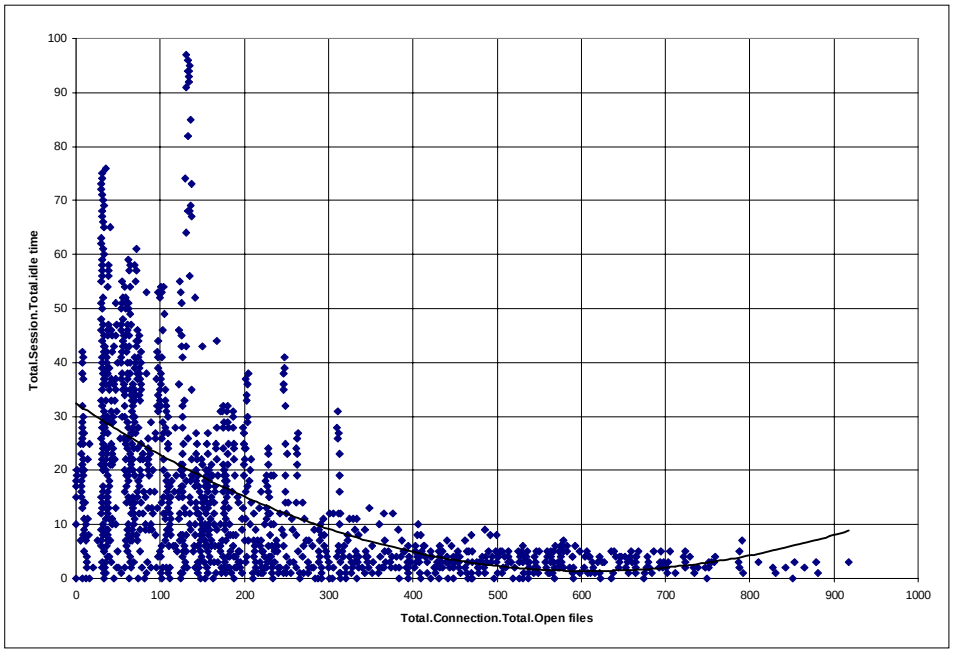


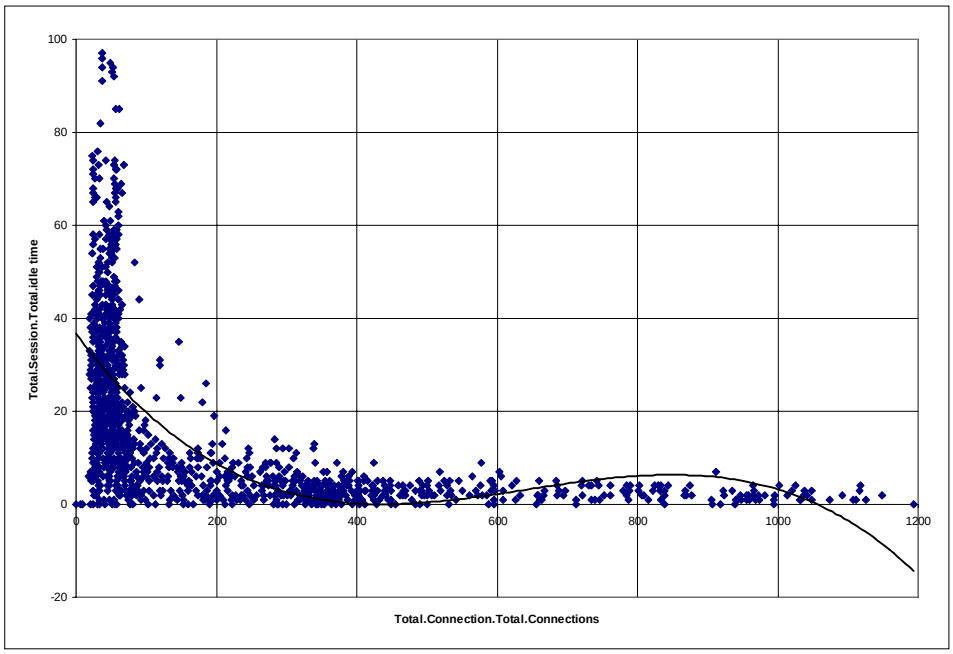


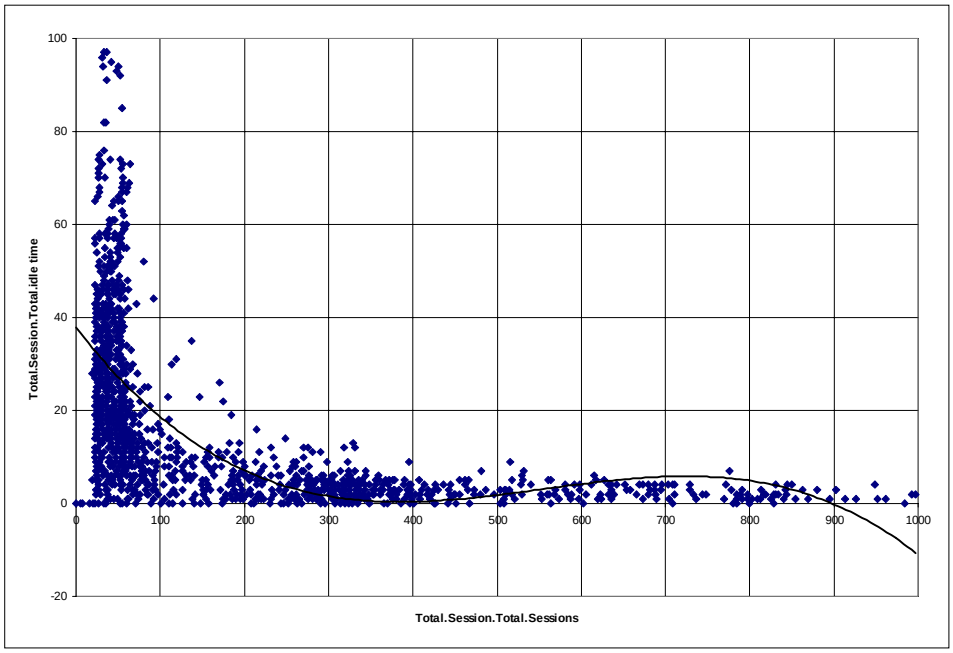
The following graph shows where the idle time comes from: about 100 user out of 400 do not turn off their machines. Sessions remain open and are not used for some time.

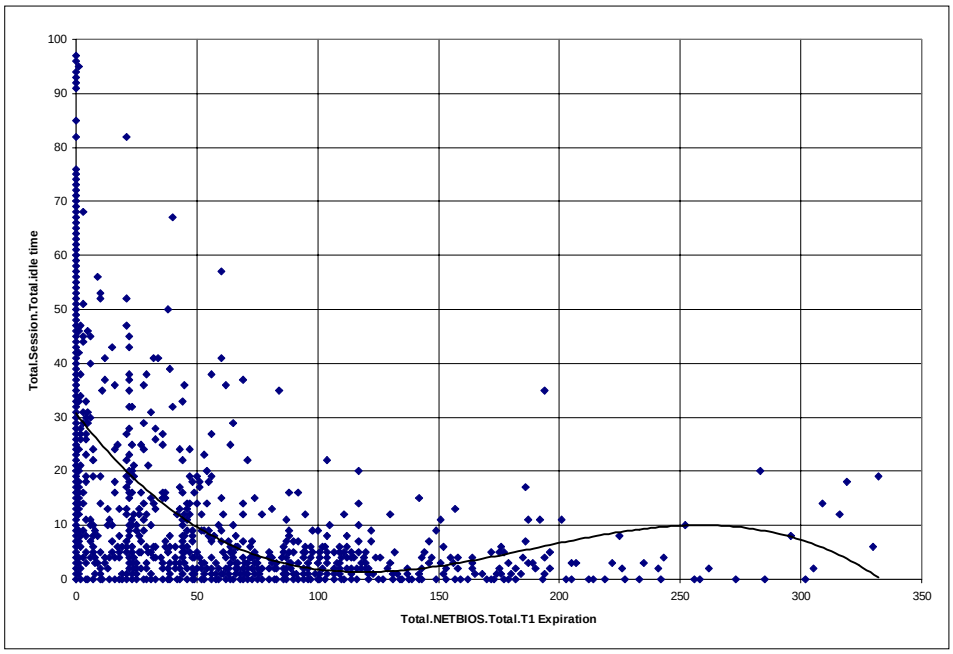


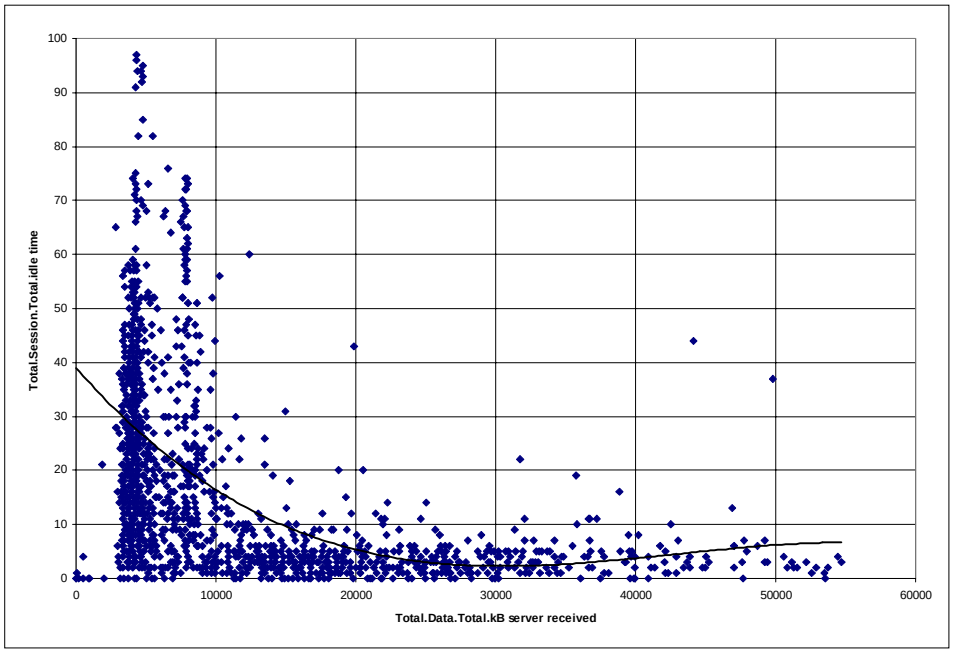

Total.Session.Total.idle time
Total.NETBIOS.Total.Ti Expiration
100
80
60
40
20
0
-20
-40
0
5000
10000
15000
20000
25000
30000

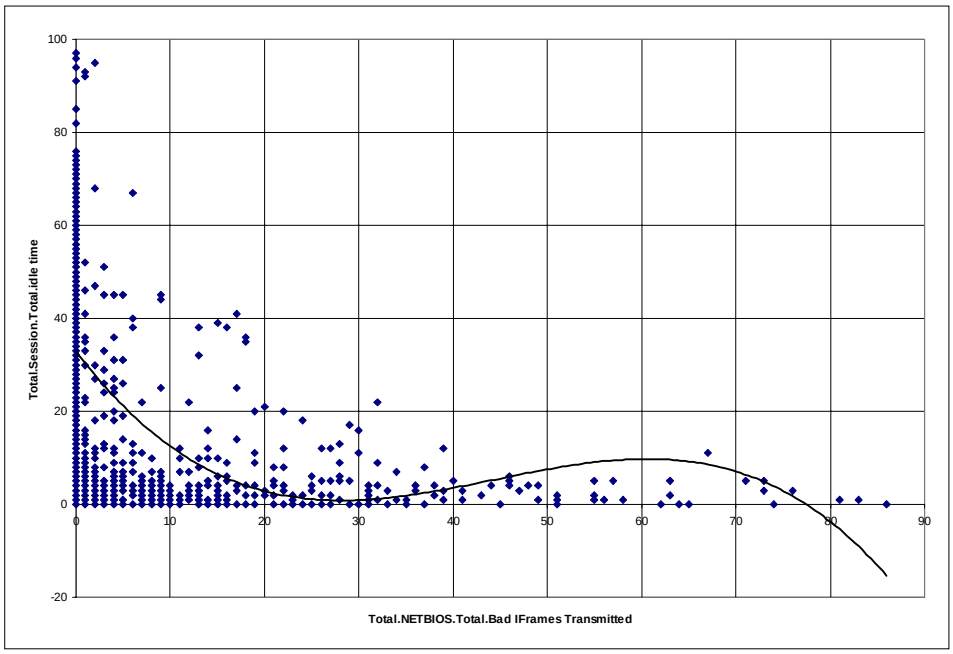

Total.Session.Total.idle time
Total.Connection.Total.Open files
100
90
80
70
60
50
40
30
20
10
0
0
100
200
300
400
500
600
700
800
900
1000


Total.Session.Total.idle time
Total.Connection.Total.Connections
100
80
60
40
20
0
-20
0
200
400
600
800
1000
1200


Total.Session.Total.idle time
Total.Session.Total.Sessions
100
80
60
40
20
0
-20
0
100
200
300
400
500
600
700
800
900
1000

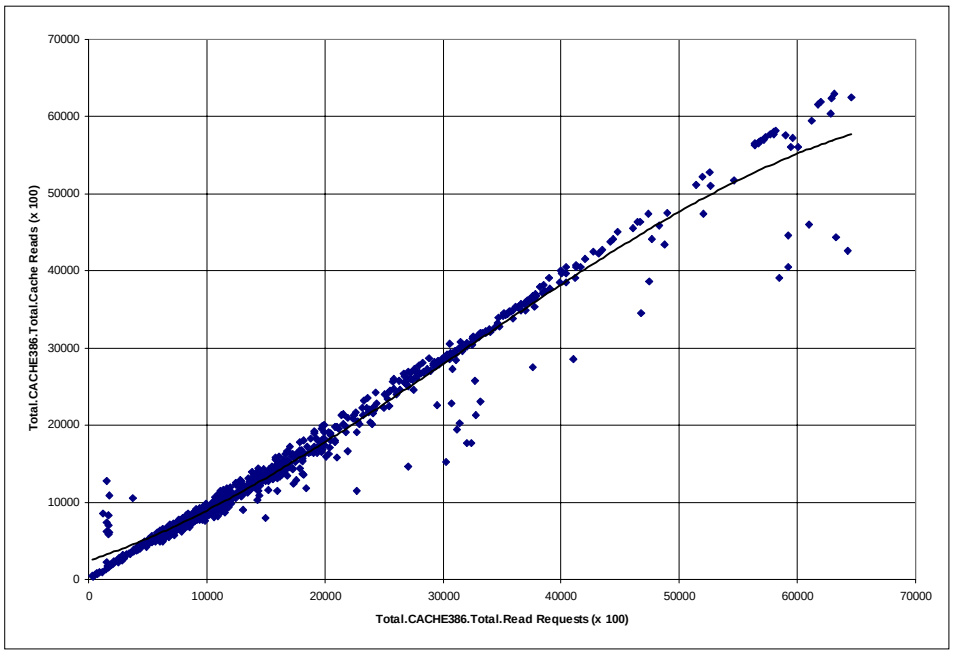

Total.Session.Total.idle time
Total.NETBIOS.Total.T1 Expiration
0
10
20
30
40
50
60
70
80
90
100
0
50
100
150
200
250
300
350


Total.Session.Total.idle time
Total.Data.Total.kB server received
0
10
20
30
40
50
60
70
80
90
100
0
10000
20000
30000
40000
50000
60000



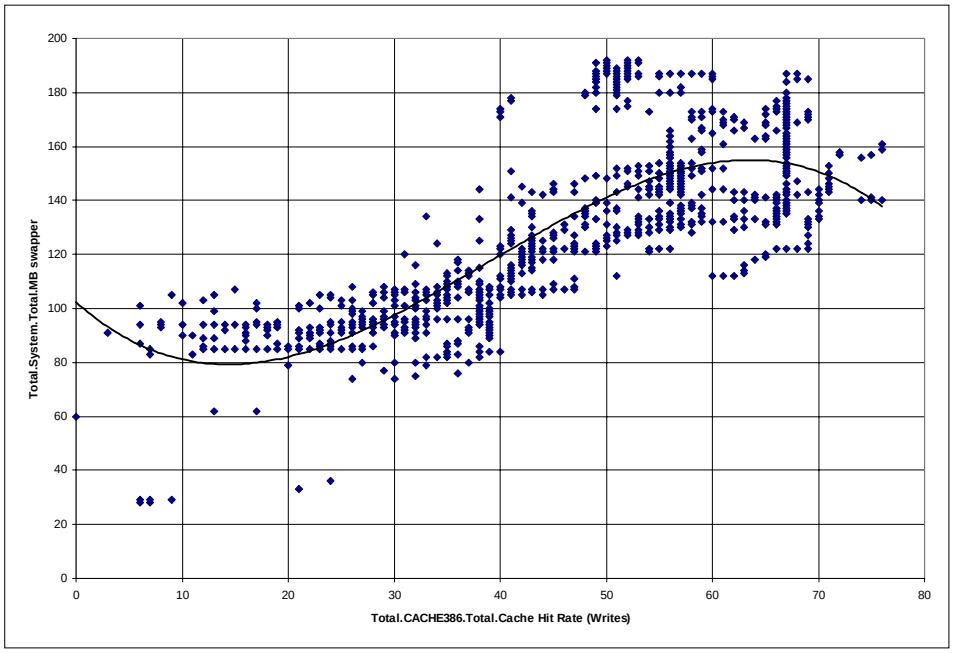
Cache Reads vs. Read Requests

The - nearly - linear association between read requests and cache reads is not a surprise, either.



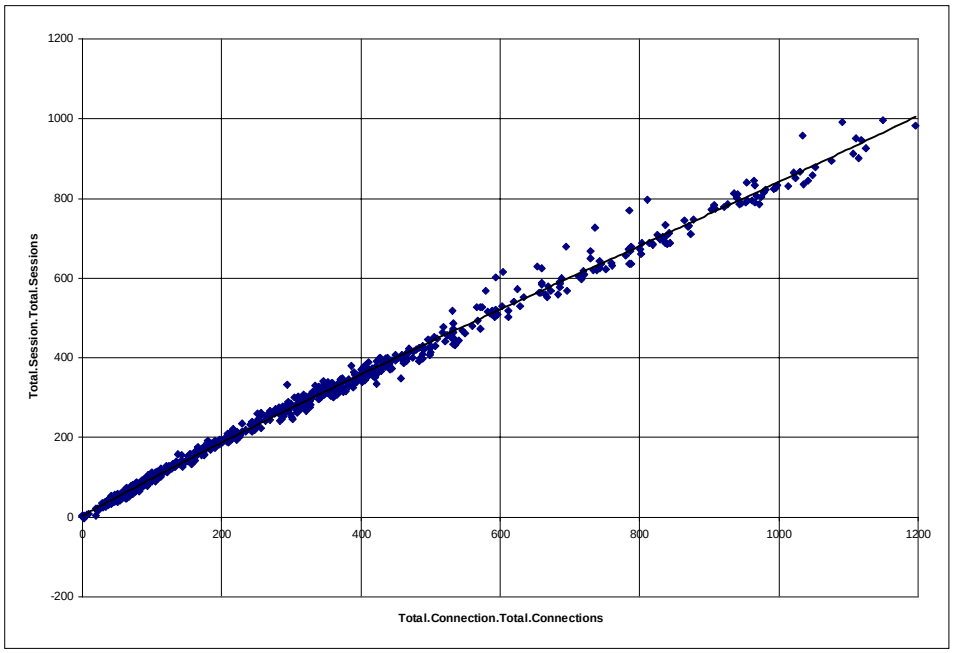
Swapper Size vs. Cache Hit Rate

An increasing size of the swap file usually means more swap activity and, thus, lead to more hits in the write cache. Depending on other disk activity, which may influence usage patterns of the cache, a certain bandwidth in the effectiveness of the cache in relation to the size of the swapper can be seen.



Sessions vs. Connections

A - nearly - linear association exists between these two data items. The number of connections increases more than the number of sessions. That means that (sometimes) two connections are established within one session.

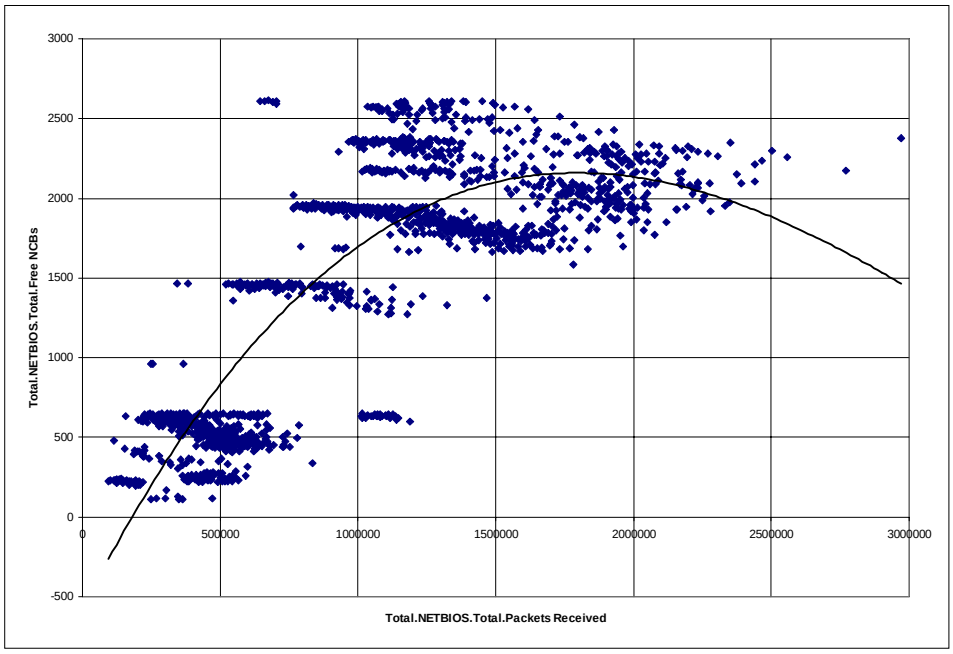


Free NCBs vs. Packets Received

Looking at the picture below it becomes clear that there is a relation between the data items, but it is not likely a steady equation. We can see clusters of markers for some values of the number of free NCBs.

First, we have to understand that the number of NCBs available to application is a fixed value which is defined by some configuration files. Most applications (like the LAN server software) allocate a fixed number of NCBs which is again defined by their configuration. Therefore the number of free NCBs is not a dynamic value that can change during hours of heavy usage.

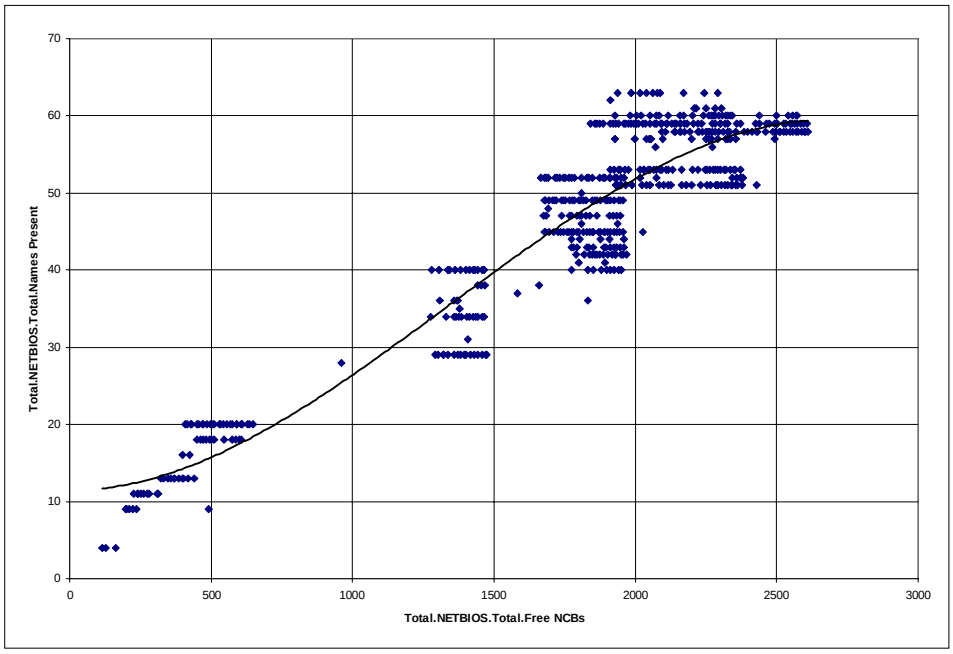
My interpretation is that there are only certain values of available NCBs because on the servers always the same application start and allocate always the same number of NCBs. With a given number of free NCBs we have another given (but not measured) number of used NCBs. For this number a bandwidth for the number of received packets exist.



Due to changes in the configuration of servers and missing monitoring data for certain servers over certain time periods (due to an unavailable machine or another problem) we see several possible values for the number of free (and used) NCBs with clusters of markers at their "height".

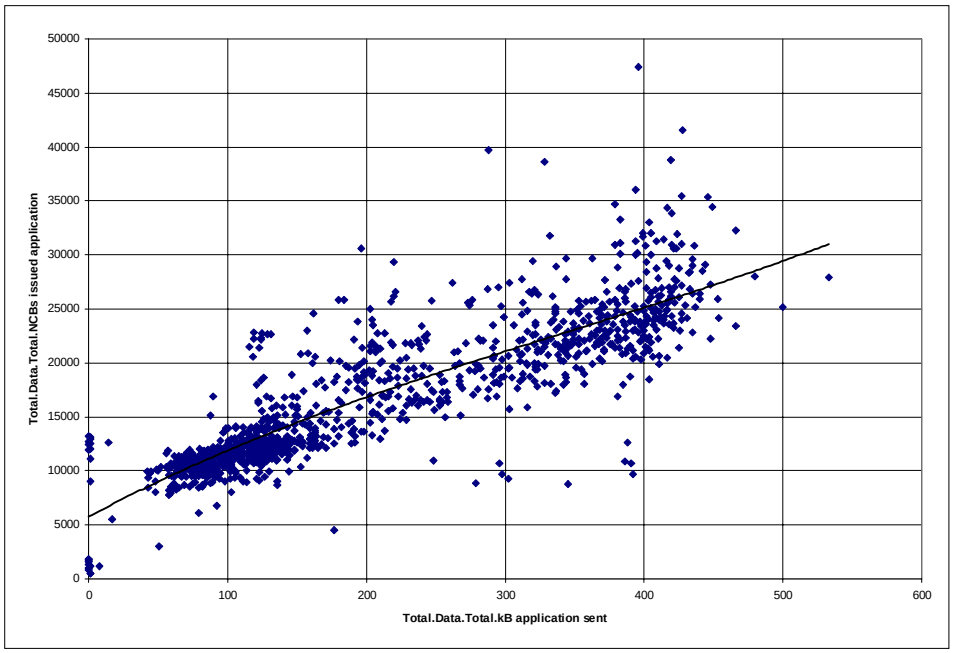
NETBIOS Names Present vs. Free NCBs

A similar picture we see for the relation between free (and used) NCBs vs. (application) names allocated by some applications.



NCBs Issued vs. kB Sent (by applications)

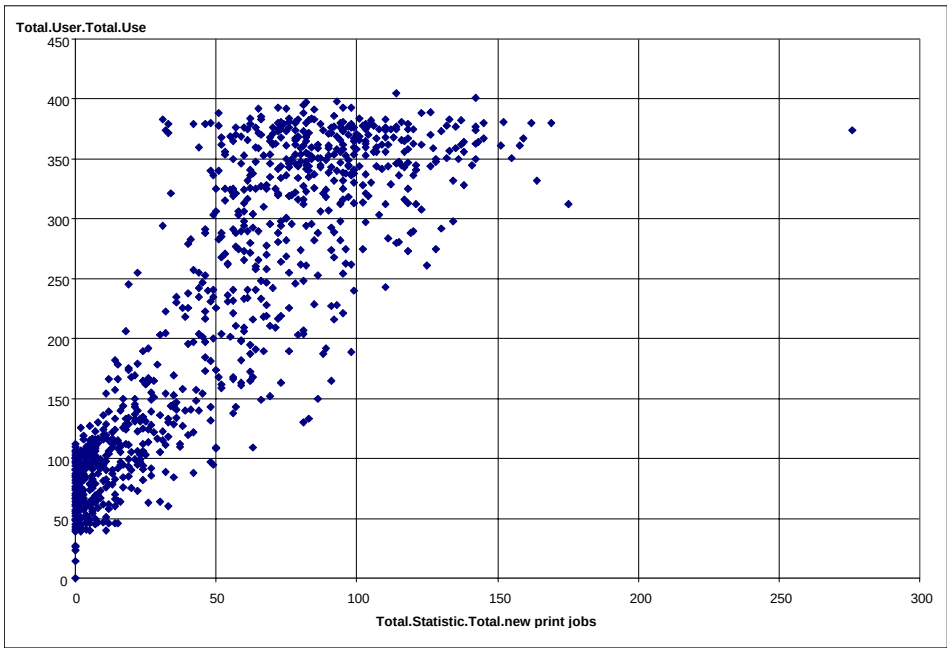
There is a linear association between the data items. The actual number of NCBs necessary to sent one kB of data varies by some amount. From the picture below we can see that about 6.000 to 8.000 NCBs are issued without much data associated to them (communication overhead and very short messages[42]). For every 100 kB an average of 5.000 NCBs are needed or about 50 NCBs for each kB.



[42] Accompanying measurements showed that current client/server software and middleware products tend to use protocols which consist of very many very small messages. Usually the overhead for each message is much greater than the effort for the data (e.g. 10:1 in one example which was analyses very extensively).

User vs. Print Job Queued

To our surprise we did not found much associations between the number of users and other data items. One rare example that such relations exists is the number of print jobs.



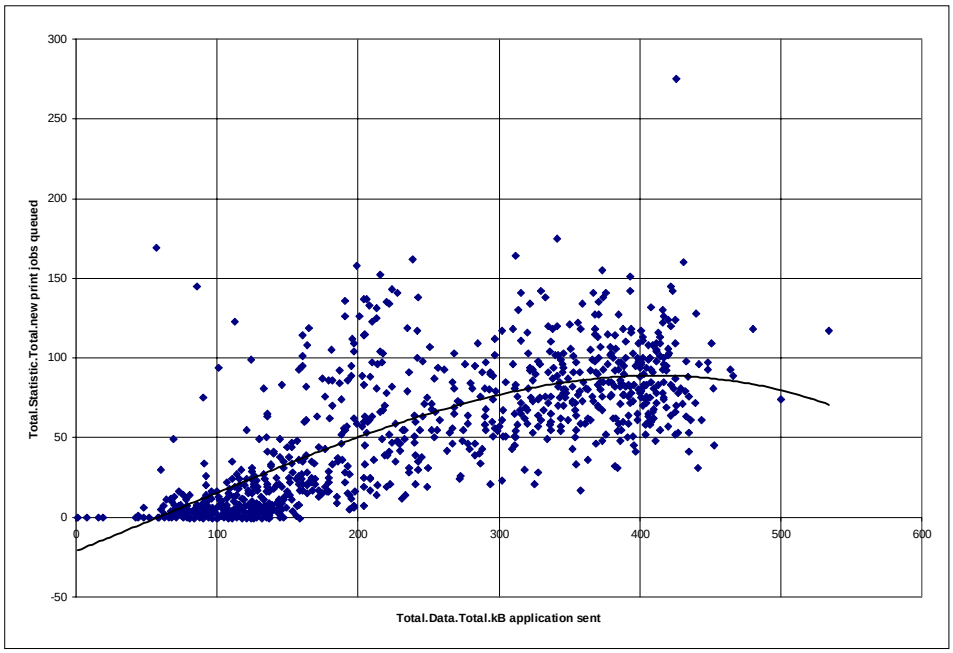
At the first look, it is no surprise that this relation exists. But on the other hand, why there are not many other relations between users and resource attributes? One assumption was that most of the activities in a client/server environment directly depends on the number of active users. Why can we find no other examples?

After several years of studying the monitoring results in different environments one conclusion is that the number of users, at it is reported by a system is not a good indication of the activity of the user. From other data we can see that only after logon most users use their computer for some time and then let it run idle.

Printing documents seems to be one of the few tasks which are done regularly by people. Consequently, with a big bandwidth, a association can be detected. On average we can draw the conclusion that for five users one print job is generated per hour.

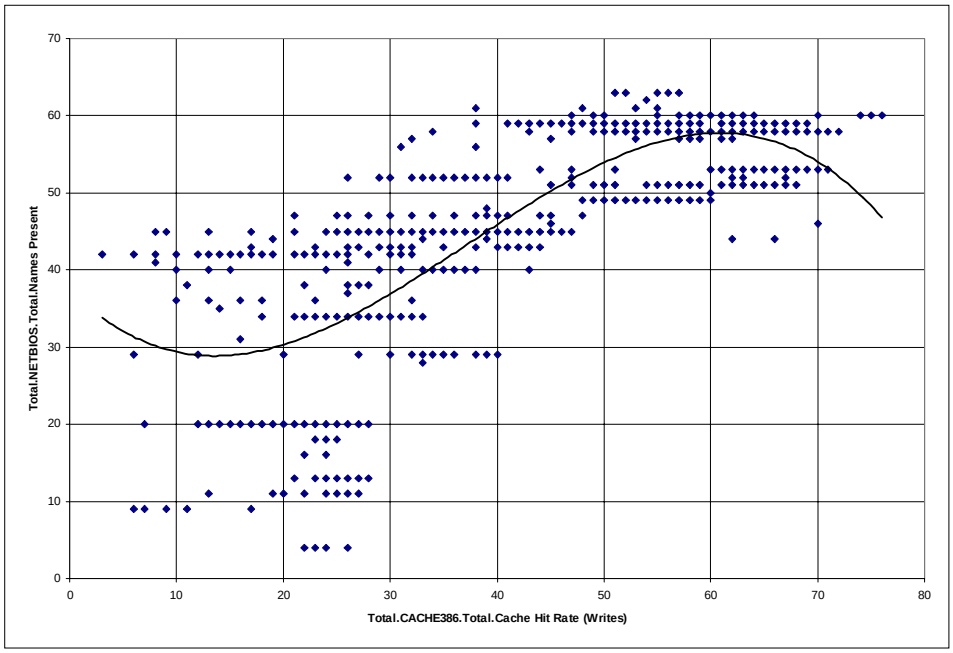
New Print Jobs vs. kB Data Sent

The analyses algorithm found the following association which is not very high. It is no surprise that number (and size) of print jobs influence the value of data sent (and received) from servers.



NETBIOS Names Present vs. Cache Hit Rate (Writes)

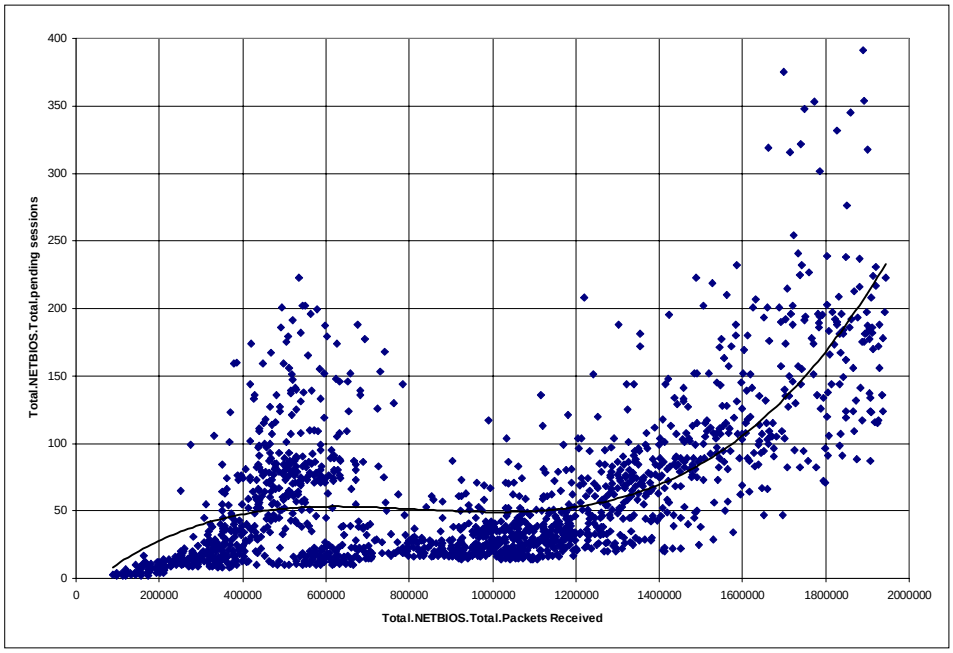
This association was not expected and its interpretation is not obvious. The cache efficiency for write operations seem to increase with the number of applications registered at the network layer.



Pending Sessions vs. Packets Received

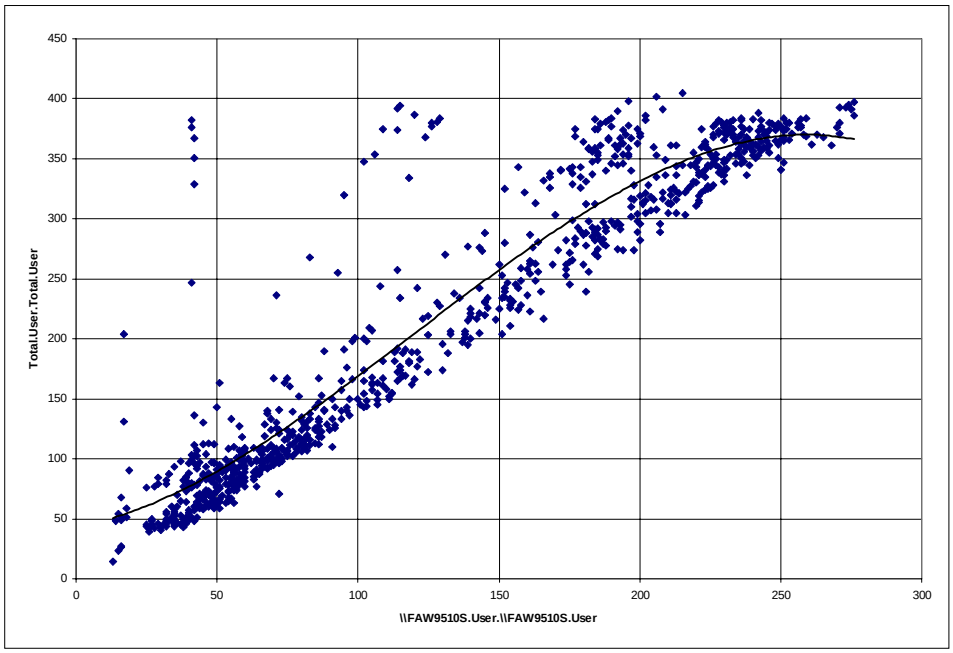
It is easier to understand that the number of packets, that travel over the network, rise with the number of established sessions. But several facts come to mind:

- Even with no active session up to 200.000 packets per hour are transmitted over the network.
- For a small number of sessions the bandwidth of packets is very big (from 200.000 up to 1.200.000). In other words, an increase of packets up to 1.200.000 packets is not associated to an increase of sessions (and users). That would mean that a small number of power users and some internal communications generate the main part of measurable traffic.
- An increasing the number of sessions does increase the number of packets up to 50% (from 400.000 to 700.000 in the area of the first maximum and from 1.200.000 to 2.000.000 in the area of the second maximum).
- There are two areas with a number of occurrences with more than 100 sessions. That underpins the assumption that beside a number of sessions which are needed for actual work, a lot of "unnecessary" sessions are created which only contribute to network overhead and consume static system resources. One reason for such sessions may be that system administrator assign certain connections to each user (e.g. printers) even if the user will never use them.

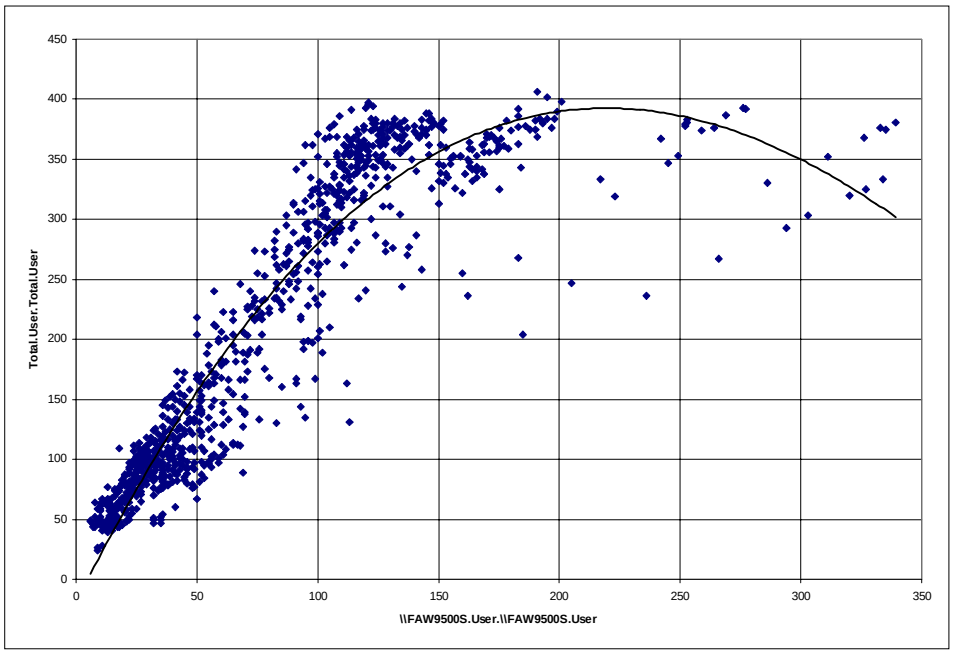


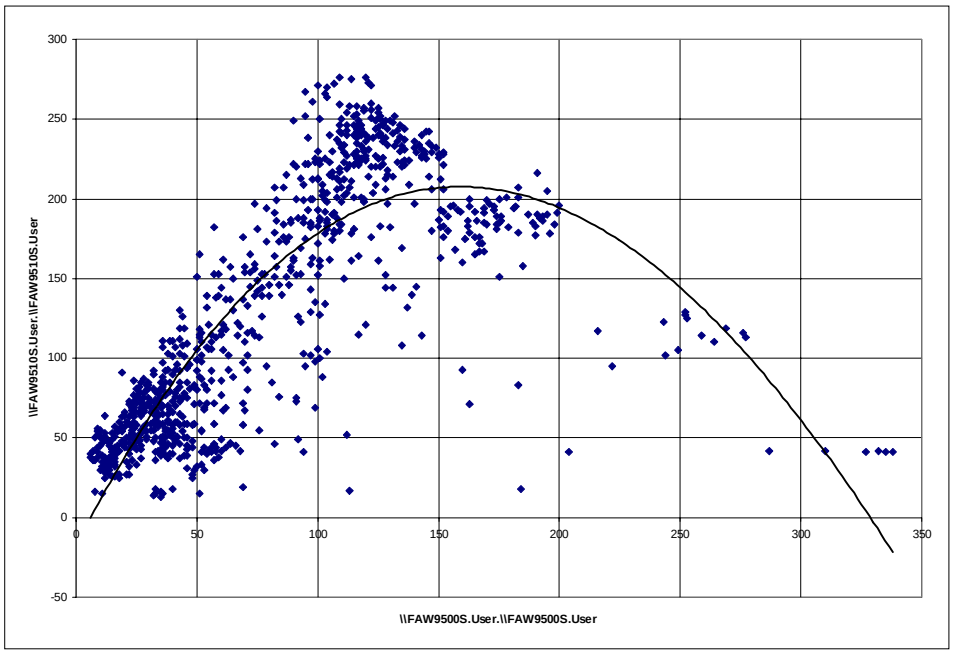
Total Users vs. Users at Server

The following picture illustrates the role of a backup domain controller. If a user logs on to a domain he may be serviced by the domain controller or one of the backup domain controller. It only depends which machine responds to the user's request first. We can see that - most of the time - the majority of users is serviced by the backup domain controller (5/8 on average).



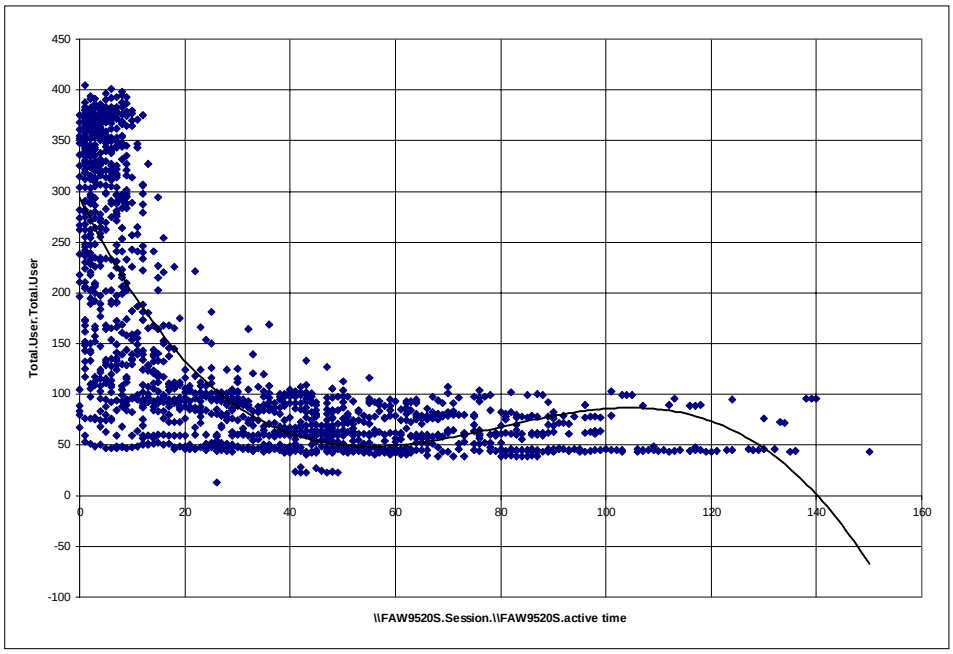
Corresponding, the picture for the domain controller shows that it only provides service to a small number of users.





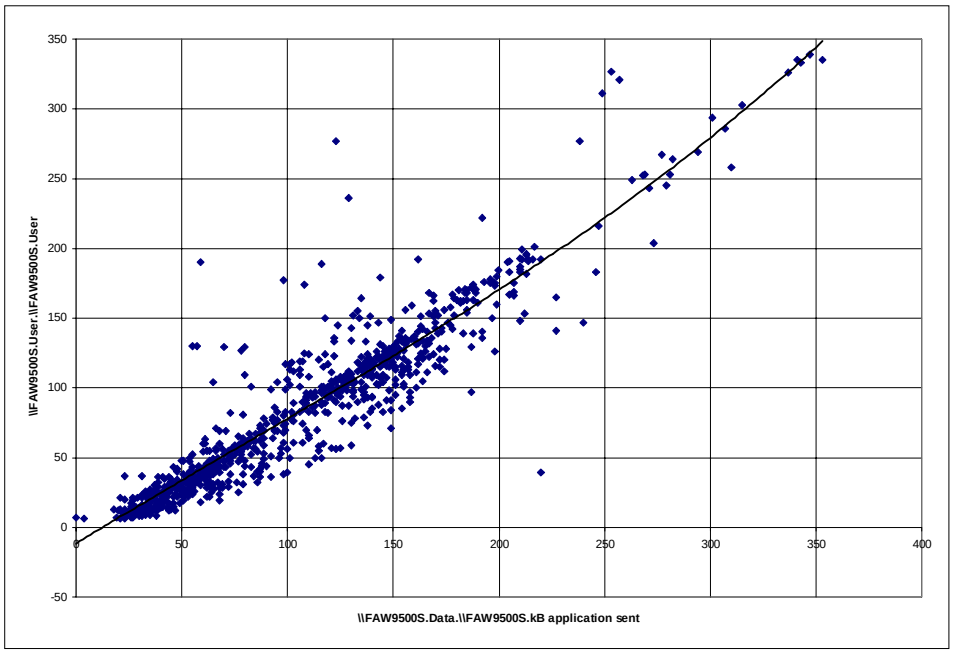
Users vs. Active Time at Server

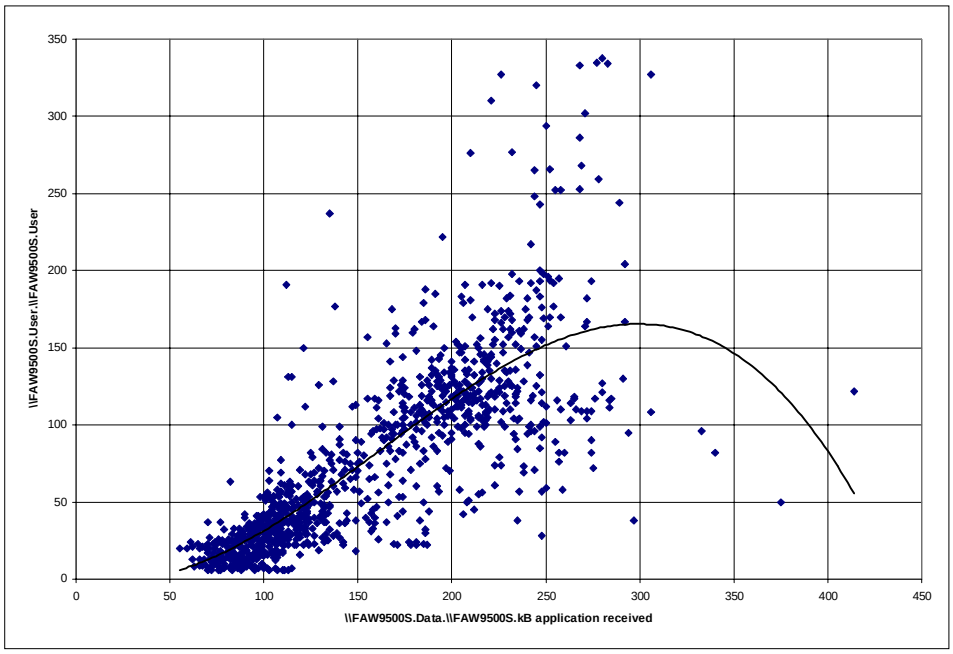
Another example of a association between the number of users and information gathered from selected servers. Again we can see that most users only use a server for a very short time. Only about 100 out of 400 users use the server over a longer period of time.



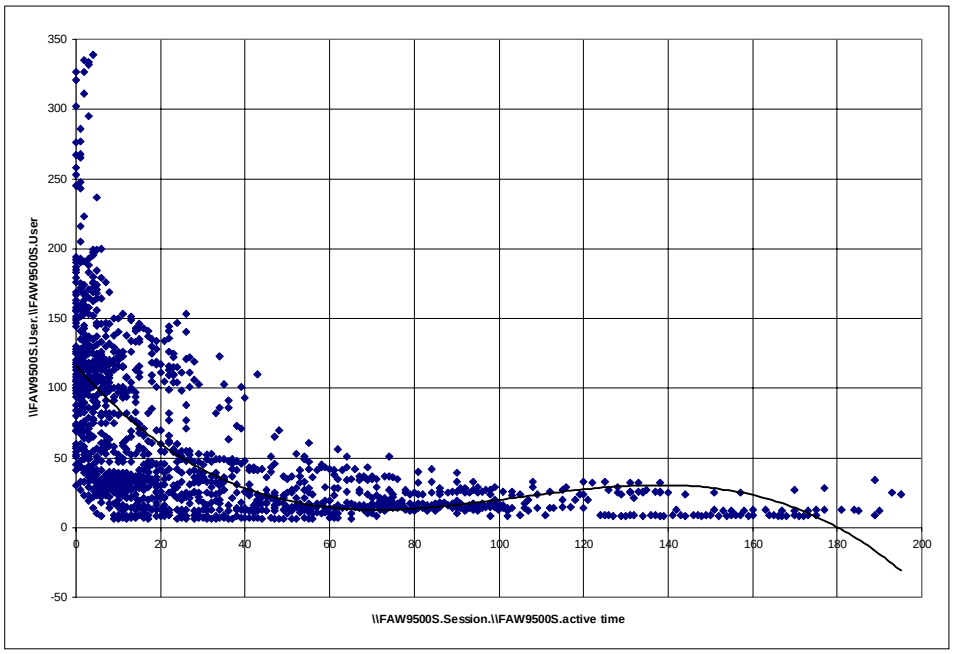
Association between Users at Domain Controller vs. Other Data Items

The following pictures represent some of the obvious associations that where detected between the number of users at the domain[43] controller and information about other monitored resources.



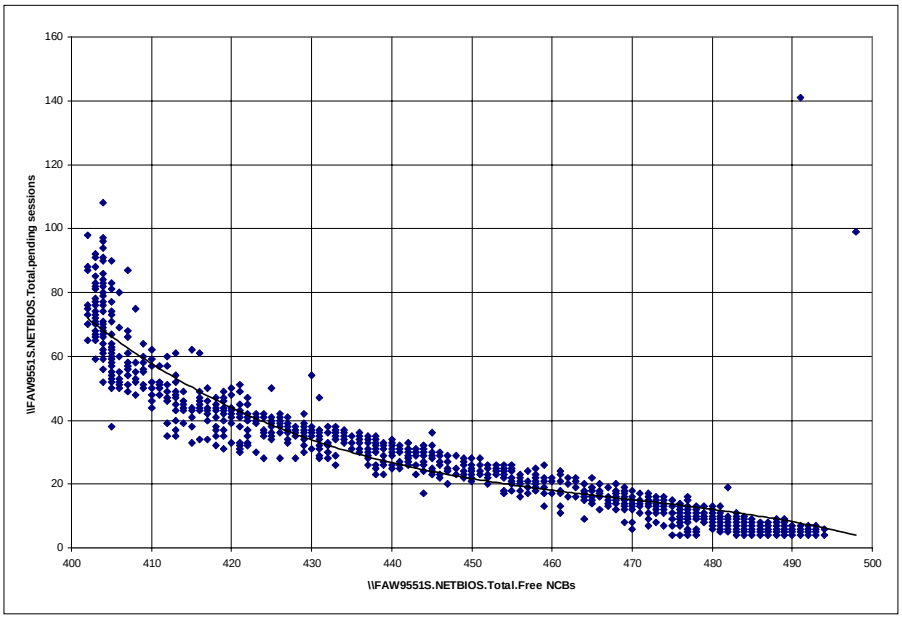


[43] Note, that the number of users is only reported for domain controllers or backup domain controllers. Therefore only for this machines a direct link between serviced users and resource consumption can be measured. For all other kind of servers it cannot be measured how many users actually use it.



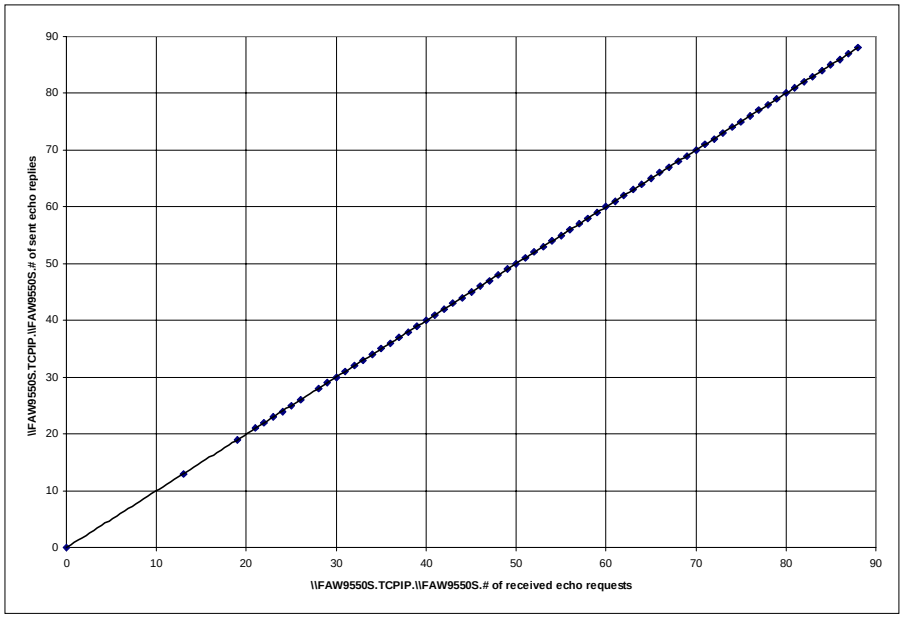
Association between number of sessions and available NCBs on a certain server

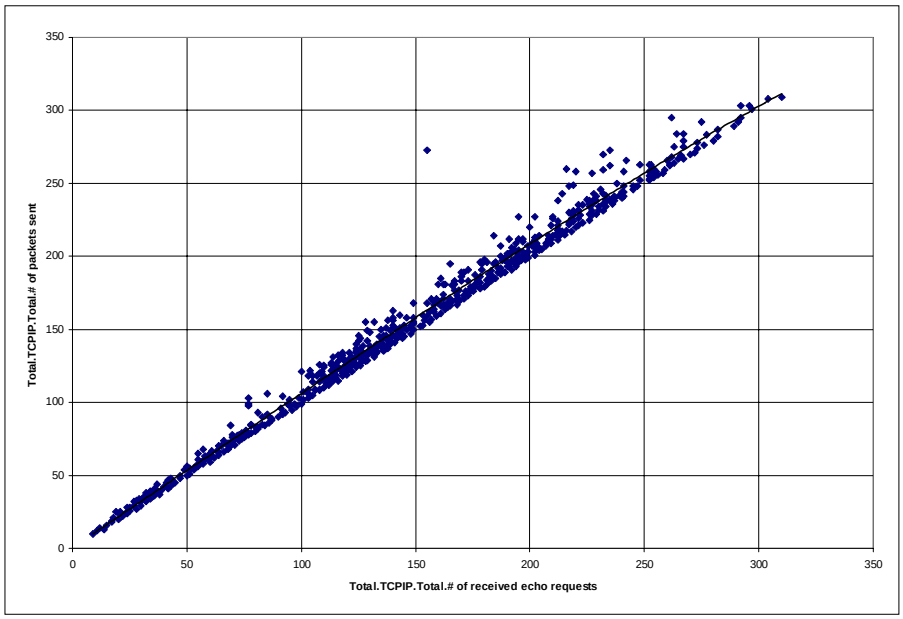
The number of available network control blocks increases as the number of pending sessions goes down. From the association we can conclude that a session needs about two NCBs. One interesting observation is that the number of NCBs does not go beyond the limit of about 400 even if the number of sessions continues to rise. Only about 20% of available NCBs are scheduled to pending sessions.

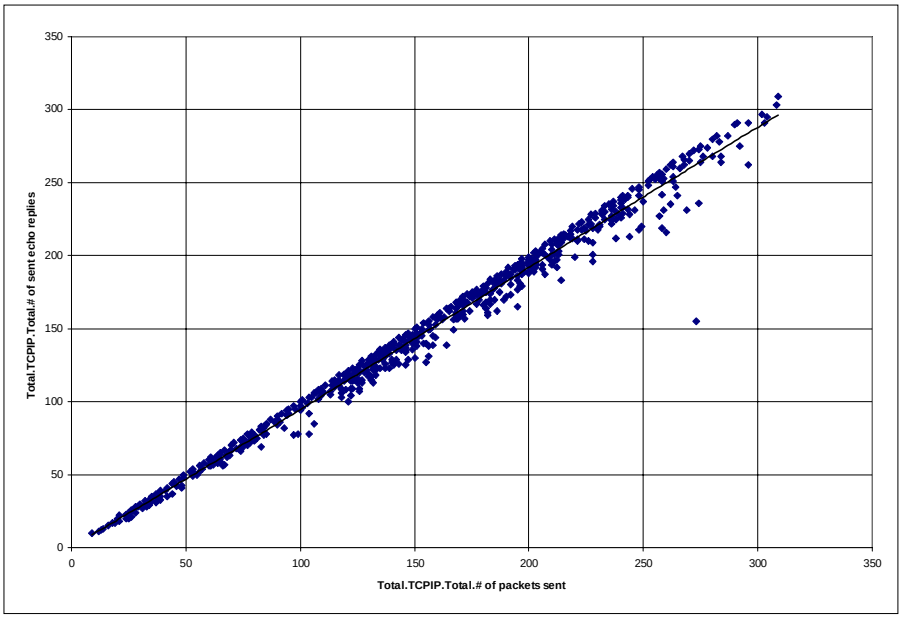


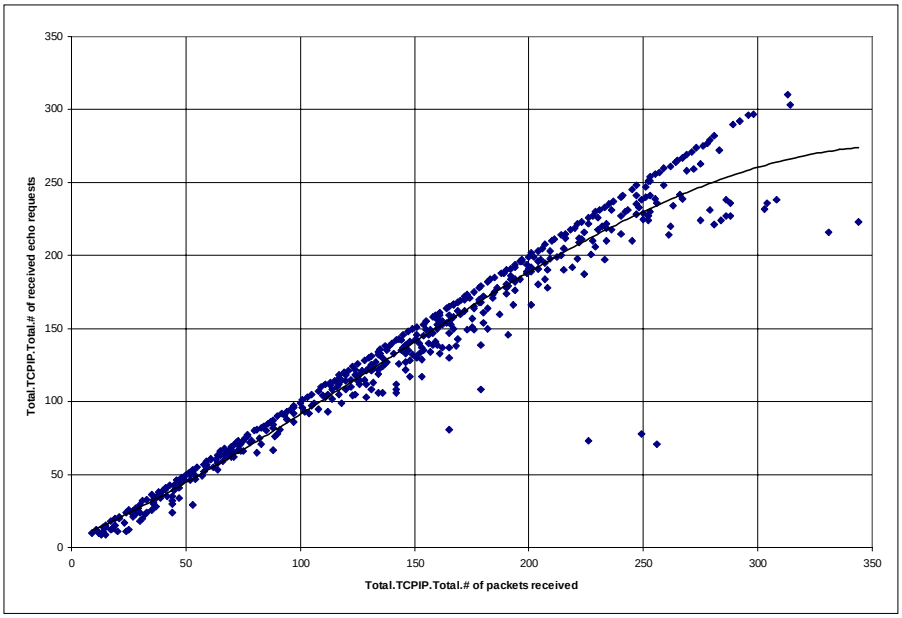
TCP/IP related associations

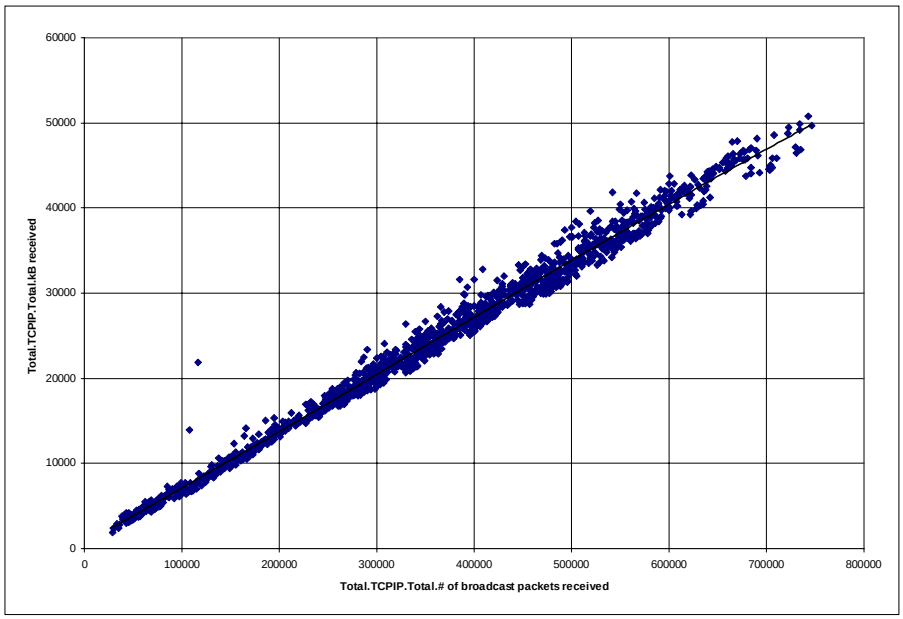
The TCP/IP protocol stack is an example of a component that offers a lot of information about it's internal and external resources. Therefore a lot of dependencies and associations could be detected. Only some very significant results were selected for this paper. However, TCP/IP is not the main communication protocol of the hosts involved. Thus, the figures below do not reflect the full network activities of the machines.

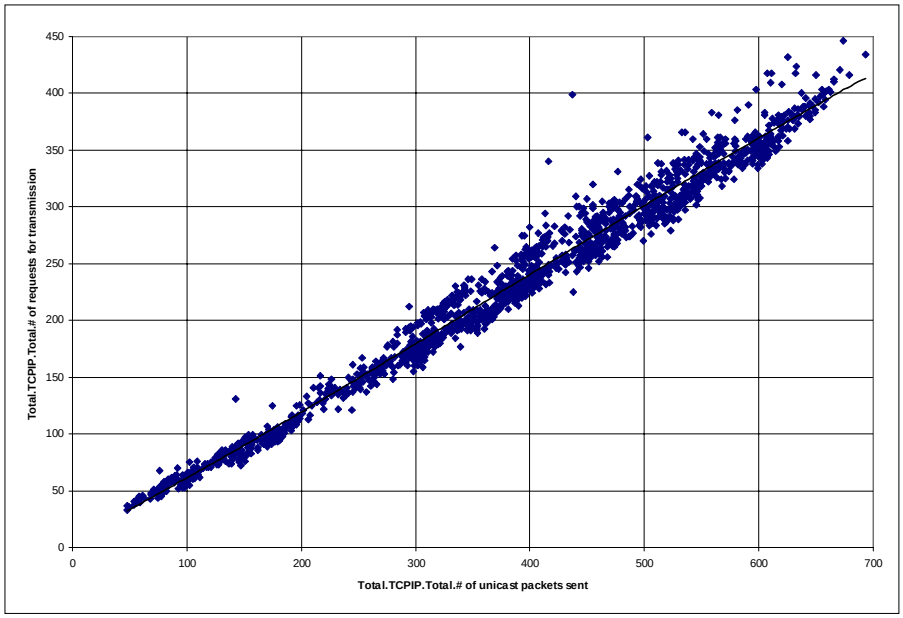


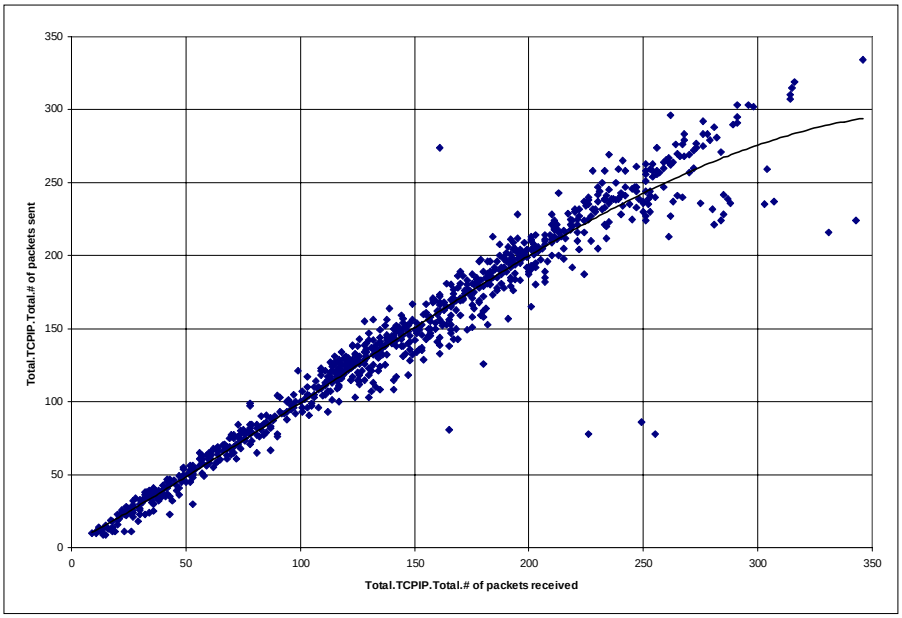


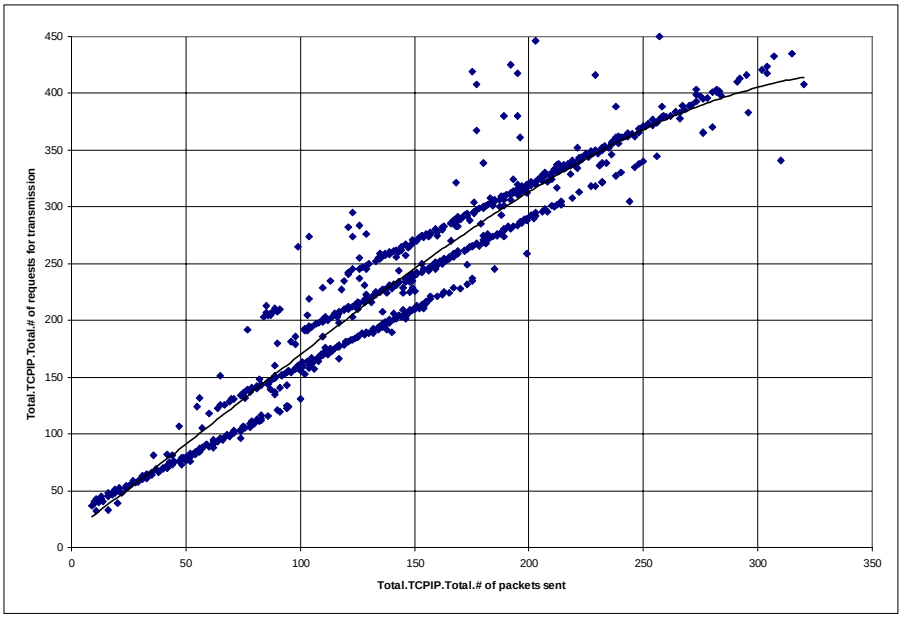

Total.TCPIP.Total.# of sent echo replies
Total.TCPIP.Total.# of packets sent
0
50
100
150
200
250
300
350

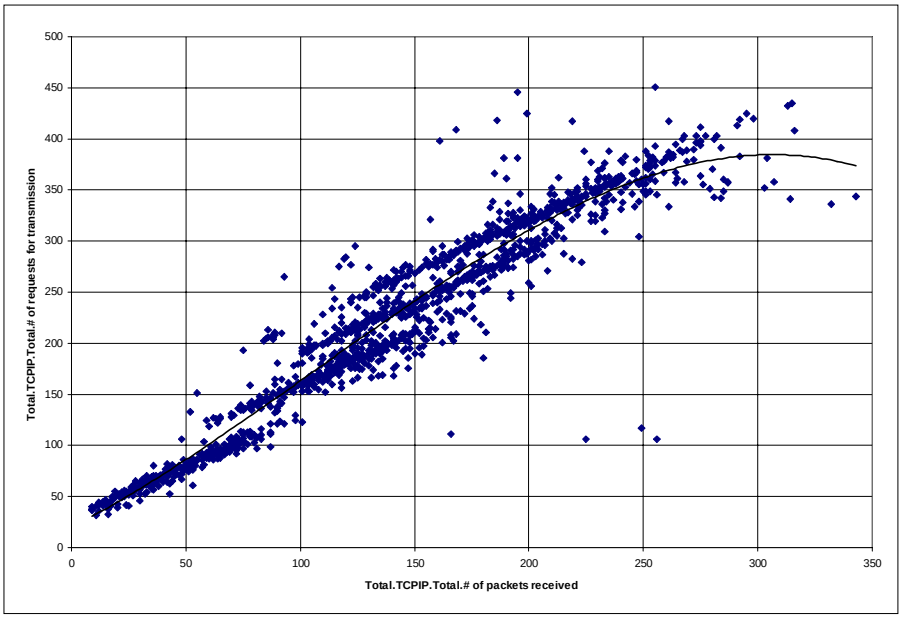

Total.TCPIP.Total.# of received echo requests
Total.TCPIP.Total.# of packets received
0
50
100
150
200
250
300
350

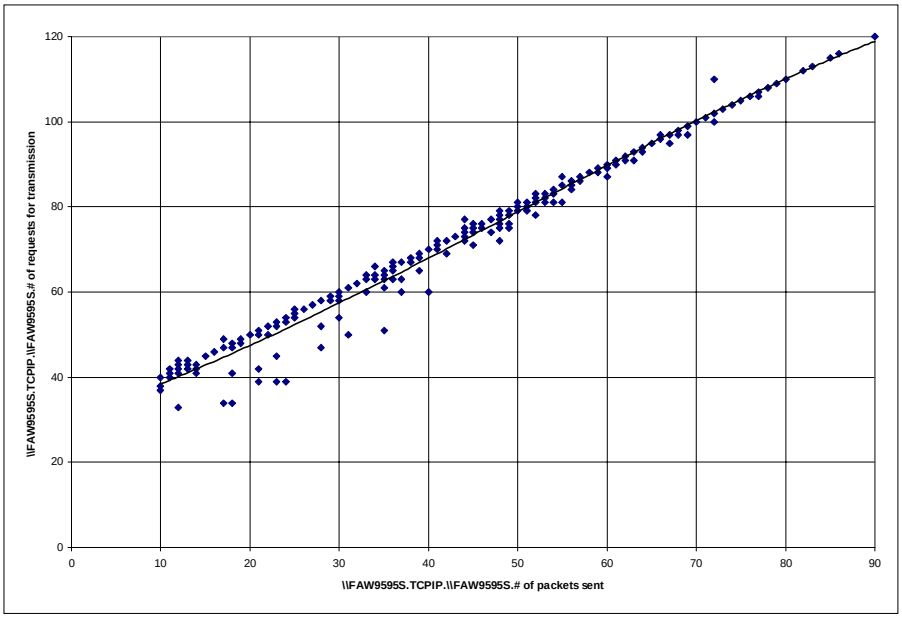

60000
50000
40000
30000
20000
10000
0
Total.TCPIP.Total.kB received
0
100000
200000
300000
400000
500000
600000
700000
800000
Total.TCPIP.Total.# of broadcast packets received

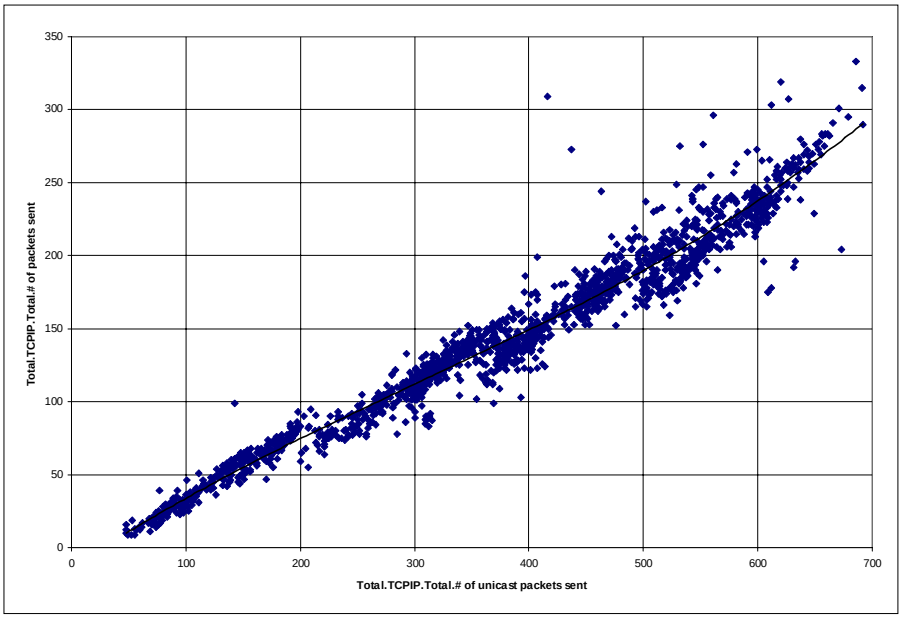

450
400
350
300
250
200
150
100
50
0
Total.TCPIP.Total.# of requests for transmission
0
100
200
300
400
500
600
700
Total.TCPIP.Total.# of unicast packets sent

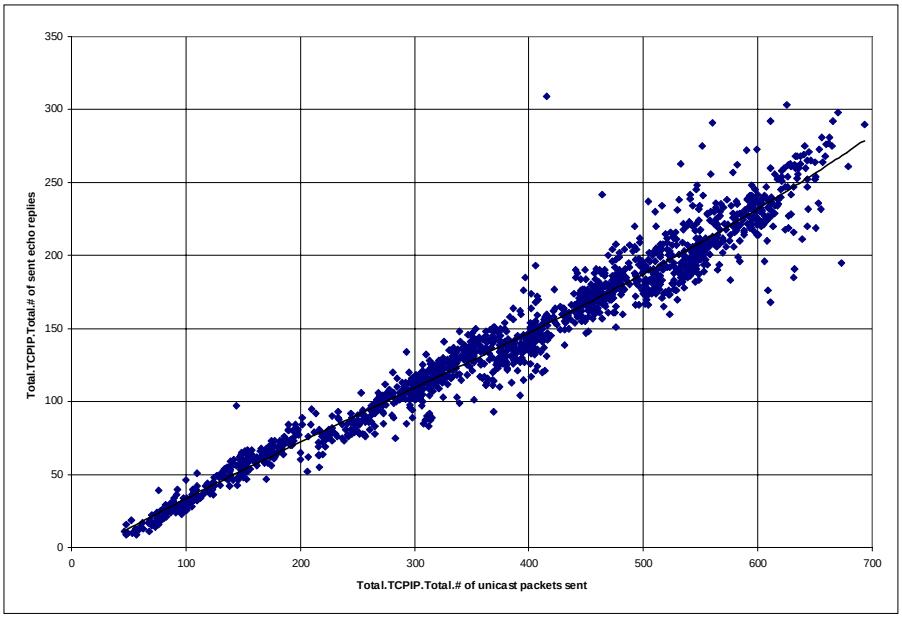

Total.TCPIP.Total.# of packets sent
Total.TCPIP.Total.# of packets received
0
50
100
150
200
250
300
350

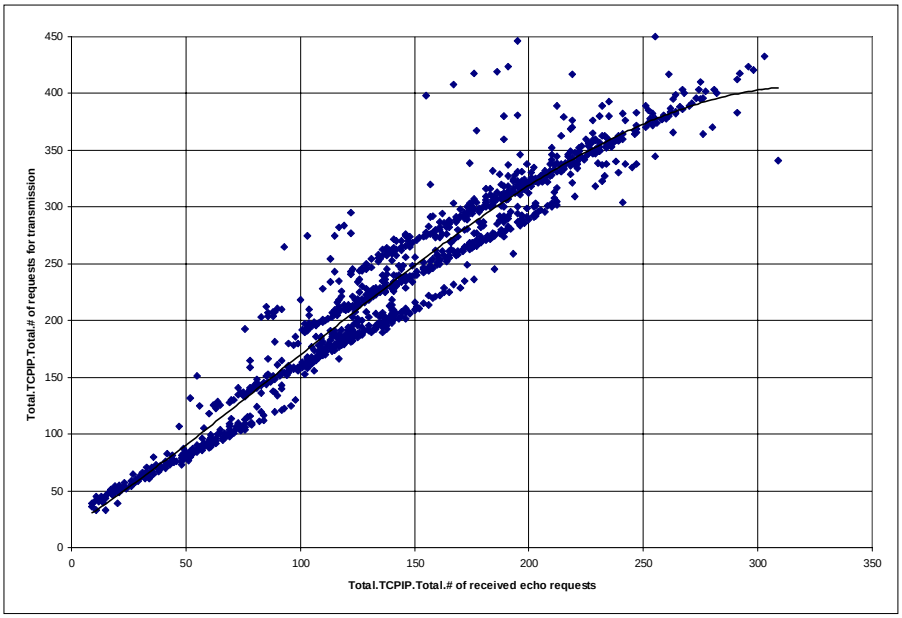

Total.TCPIP.Total.# of requests for transmission
Total.TCPIP.Total.# of packets sent
0
50
100
150
200
250
300
350
400
450

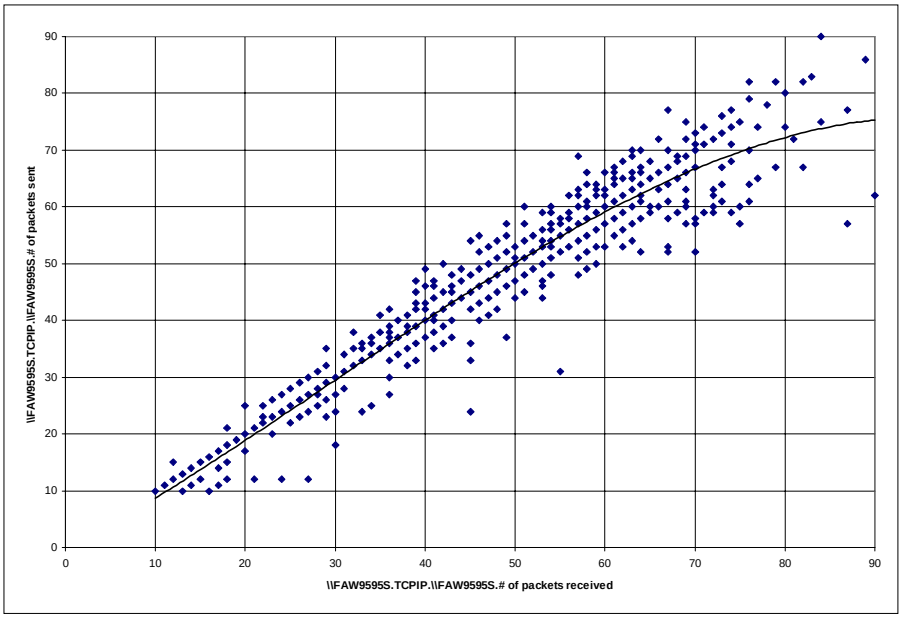

Total.TCPIP.Total.# of requests for transmission
Total.TCPIP.Total.# of packets received
0
50
100
150
200
250
300
350
400
450
500

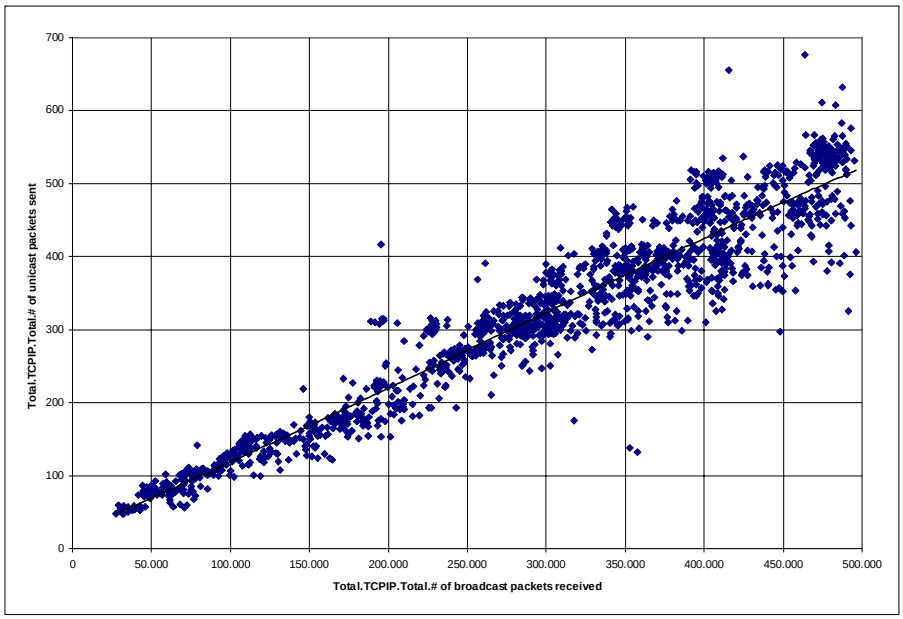

\\FAW9595S.TCPIP.\\FAW9595S.# of requests for transmission
\\FAW9595S.TCPIP.\\FAW9595S.# of packets sent
0
10
20
30
40
50
60
70
80
90
100
120

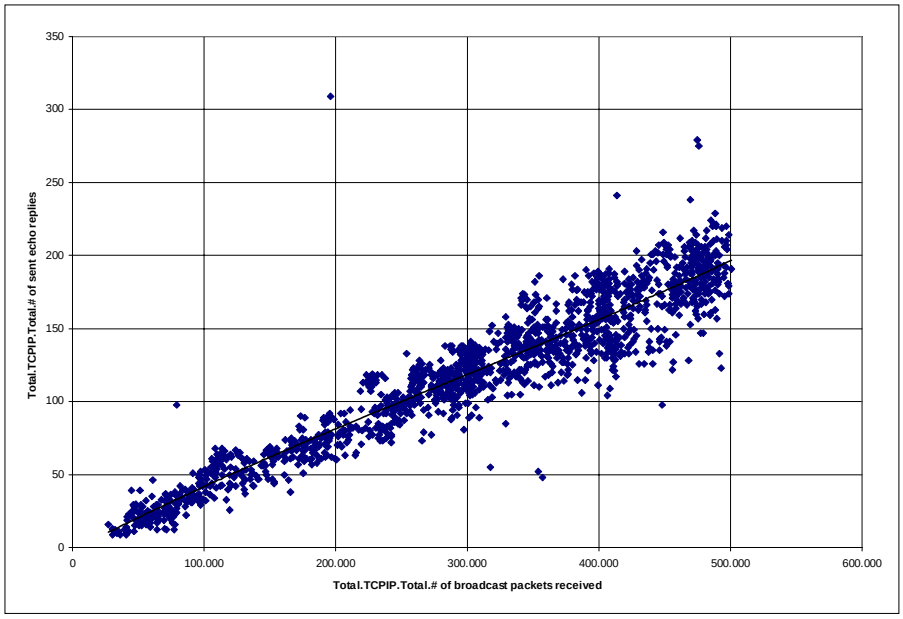

Total.TCPIP.Total.# of packets sent
Total.TCPIP.Total.# of unicast packets sent
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600
700

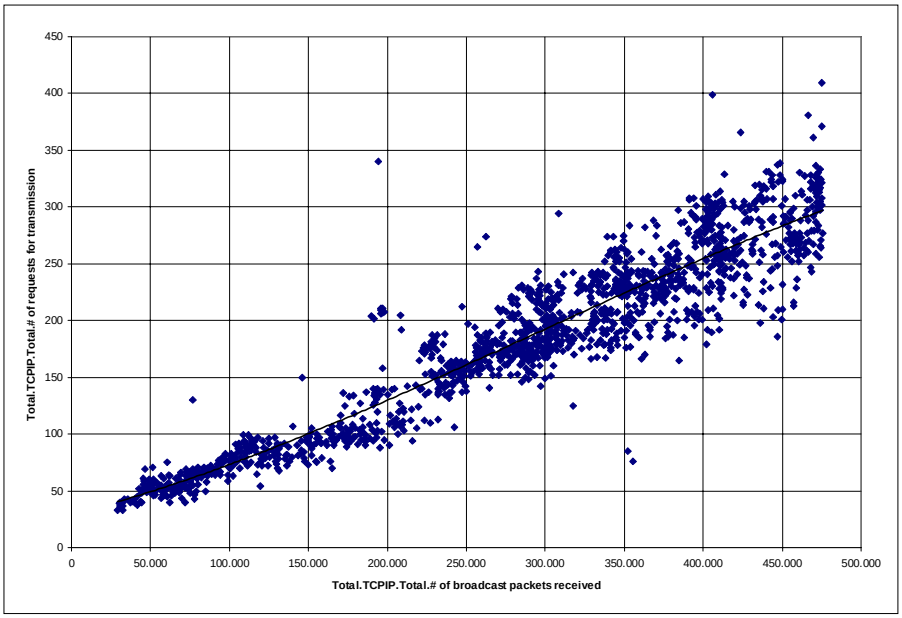

Total.TCPIP.Total.# of sent echo replies
Total.TCPIP.Total.# of unicast packets sent
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600
700

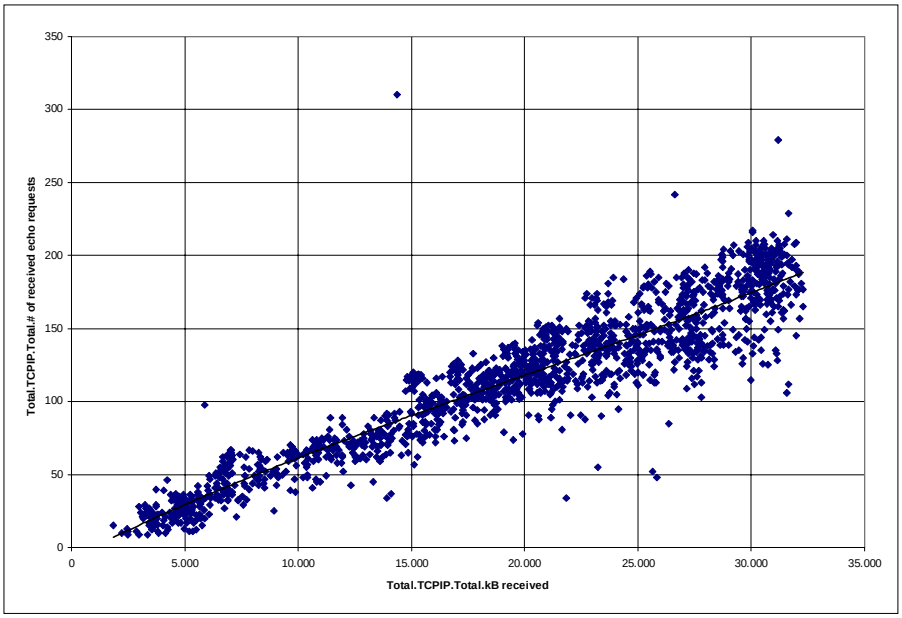

Total.TCPIP.Total.# of requests for transmission
Total.TCPIP.Total.# of received echo requests
0
50
100
150
200
250
300
350
400
450

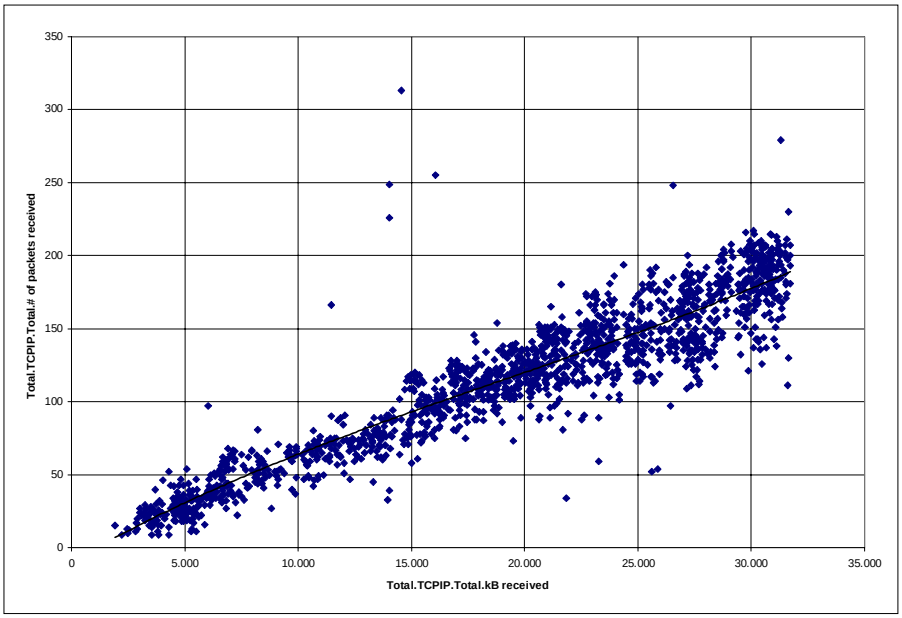

\\FAW9595S.TCPIP.\\FAW9595S.# of packets sent
\\FAW9595S.TCPIP.\\FAW9595S.# of packets received
0
10
20
30
40
50
60
70
80
90

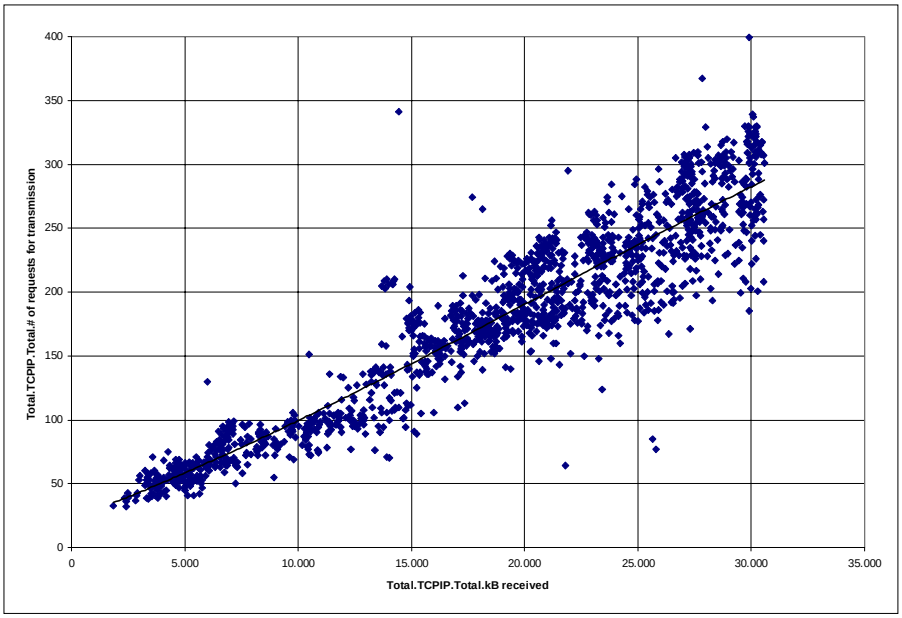

Total.TCPIP.Total.# of unicast packets sent
Total.TCPIP.Total.# of broadcast packets received
0
100
200
300
400
500
600
700
0
50.000
100.000
150.000
200.000
250.000
300.000
350.000
400.000
450.000
500.000

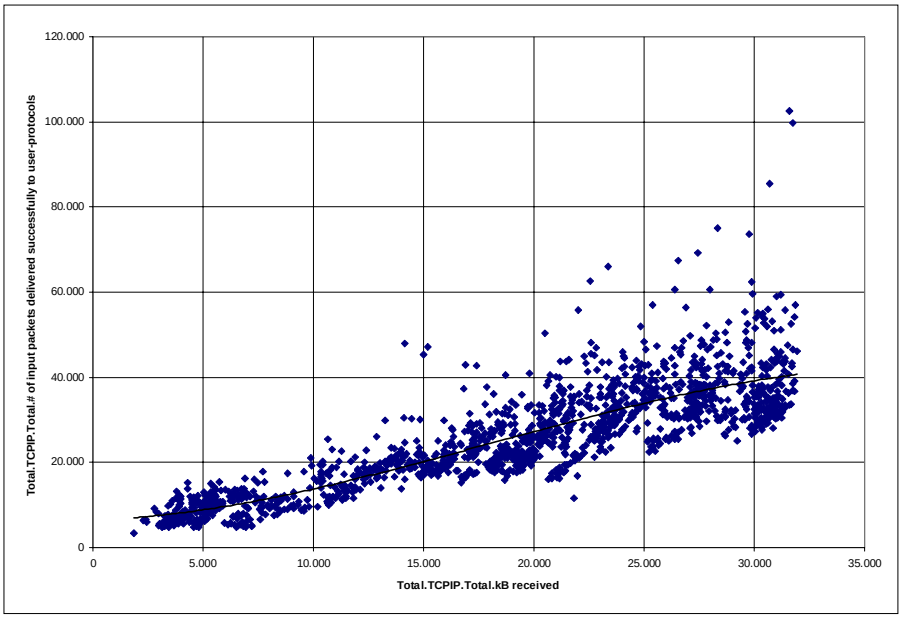

Total.TCPIP.Total.# of sent echo replies
Total.TCPIP.Total.# of broadcast packets received
0
50
100
150
200
250
300
350
0
100.000
200.000
300.000
400.000
500.000
600.000

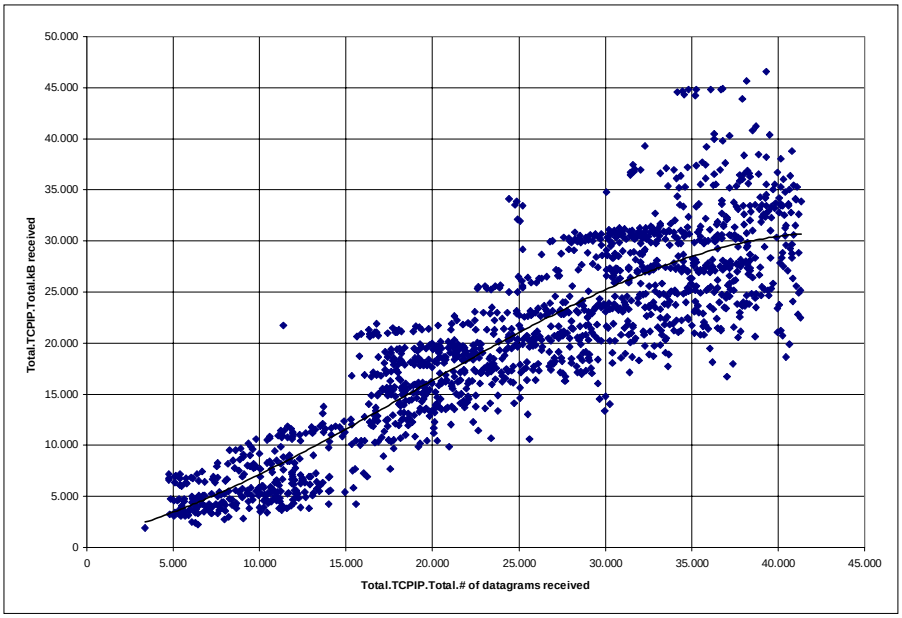

Total.TCPIP.Total.# of requests for transmission
Total.TCPIP.Total.# of broadcast packets received
0
50
100
150
200
250
300
350
400
450
0
50.000
100.000
150.000
200.000
250.000
300.000
350.000
400.000
450.000
500.000

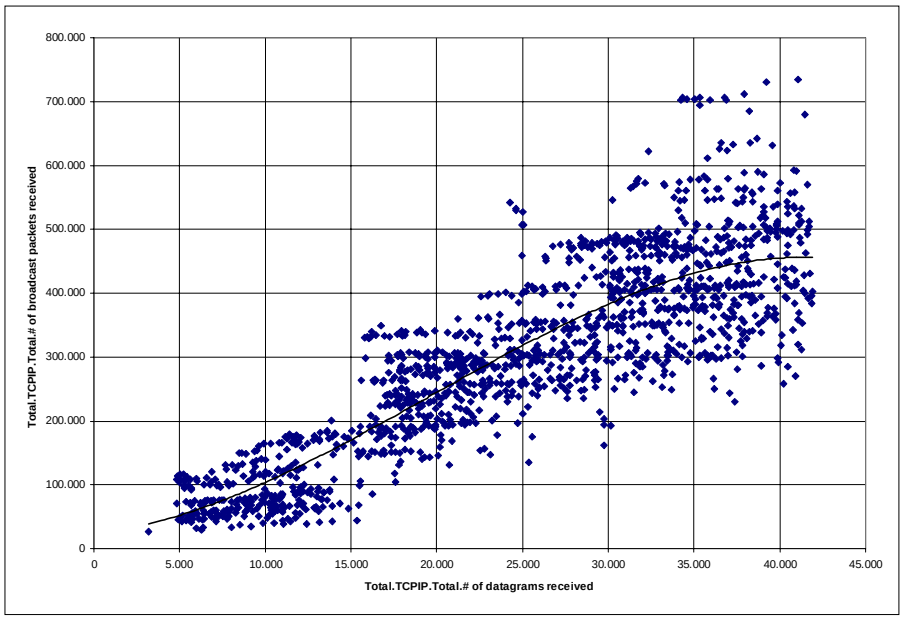

Total.TCPIP.Total.# of received echo requests
Total.TCPIP.Total.kB received
0
50
100
150
200
250
300
350
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000

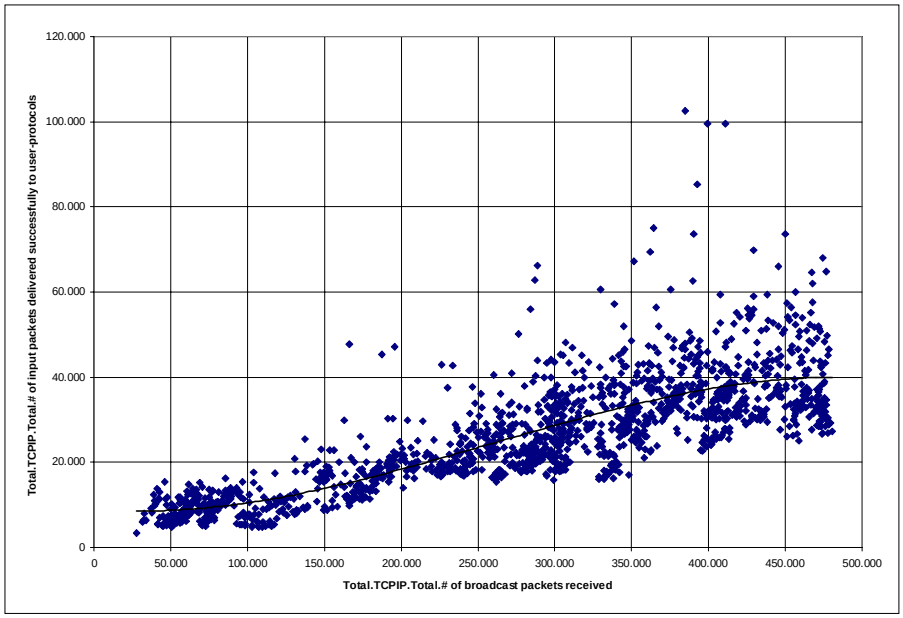

Total.TCPIP.Total.# of packets received
Total.TCPIP.Total.kB received
0
50
100
150
200
250
300
350
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000

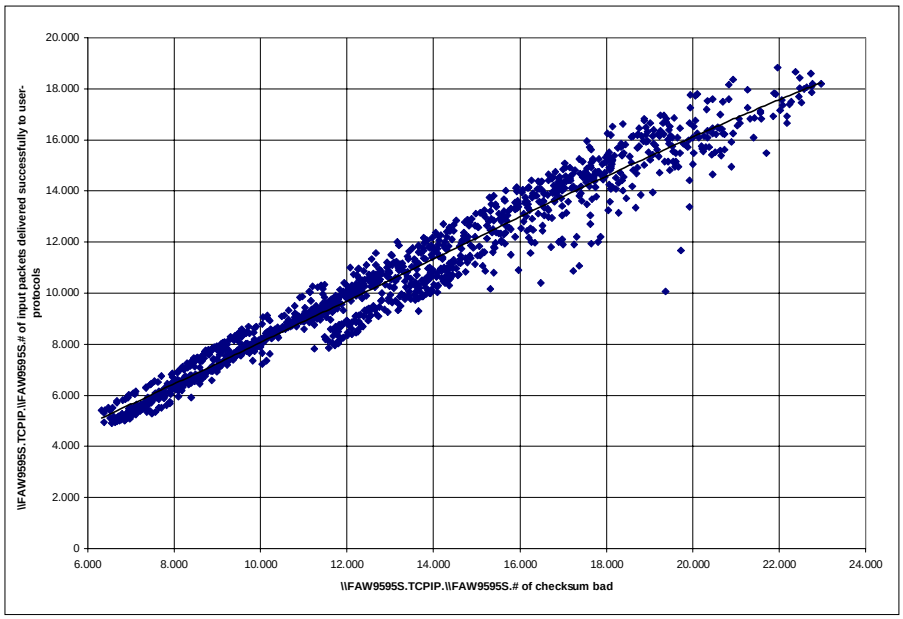

Total.TCPIP.Total.# of requests for transmission
Total.TCPIP.Total.kB received
0
50
100
150
200
250
300
350
400
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000

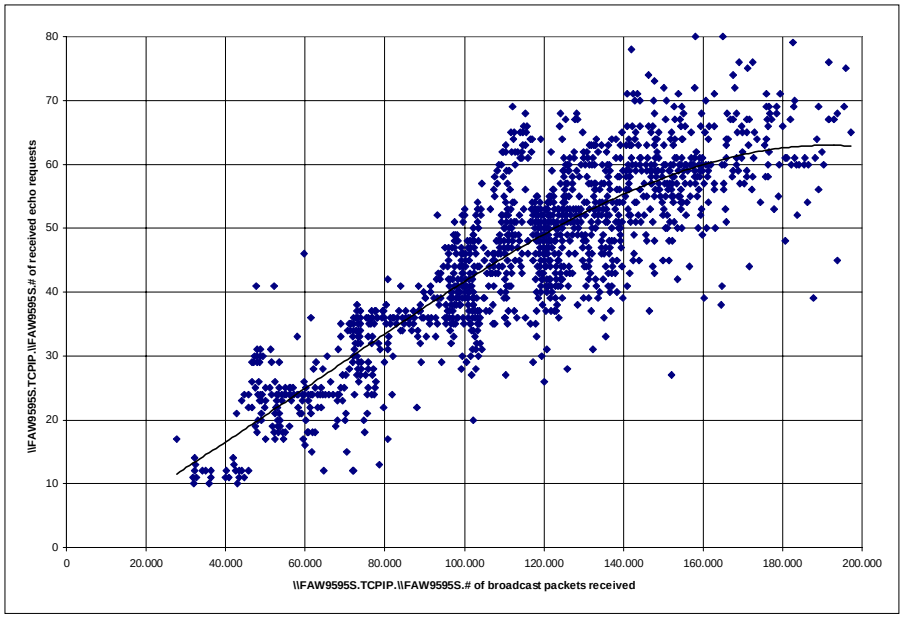

120.000
100.000
80.000
60.000
40.000
20.000
0
Total.TCPIP.Total.# of input packets delivered successfully to user-protocols
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
Total.TCPIP.Total.kB received


50.000
45.000
40.000
35.000
30.000
25.000
20.000
15.000
10.000
5.000
0
Total.TCPIP.Total.kB received
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
Total.TCPIP.Total.# of datagrams received


800.000
700.000
600.000
500.000
400.000
300.000
200.000
100.000
0
Total.TCPIP.Total.# of broadcast packets received
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
Total.TCPIP.Total.# of datagrams received

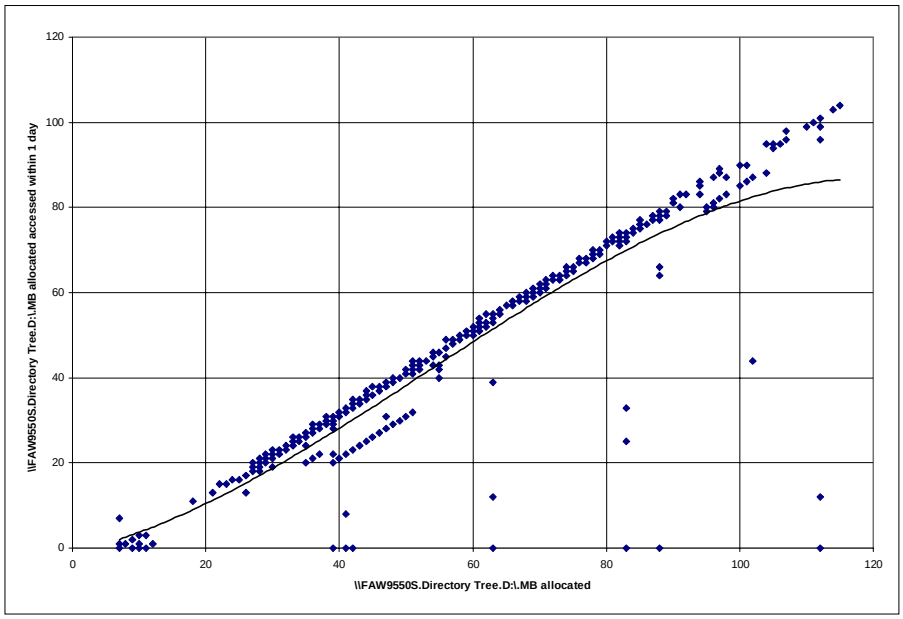

120.000
100.000
80.000
60.000
40.000
20.000
0
Total.TCPIP.Total.# of input packets delivered successfully to user-protocols
0
50.000
100.000
150.000
200.000
250.000
300.000
350.000
400.000
450.000
500.000
Total.TCPIP.Total.# of broadcast packets received


\\FAW9595S.TCPIP.\\FAW9595S.# of input packets delivered successfully to user-protocols
20.000
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0
6.000
8.000
10.000
12.000
14.000
16.000
18.000
20.000
22.000
24.000
\\FAW9595S.TCPIP.\\FAW9595S.# of checksum bad


\\FAW9595S.TCPIP.\\FAW9595S.# of received echo requests
80
70
60
50
40
30
20
10
0
0
20.000
40.000
60.000
80.000
100.000
120.000
140.000
160.000
180.000
200.000
\\FAW9595S.TCPIP.\\FAW9595S.# of broadcast packets received

Association between the amount of allocated disk space vs. used disk space

The following picture shows the special case that there is a direct relationship between the diskspace that is allocated on a logical drive and the time it was accessed. Because most of the information on the disk is always used during the last 24 hours we assume that the disk is used as temporary storage and is cleared every day.



This is an example of a finding in a special environment related case. This information cannot be generalized but may be vital for the administration of the site and for planning systems management activities or future purchase of new equipment

9. Conclusion

In this dissertation we have presented the problems associated with workload monitoring in distributed PC server systems in general and with long-term monitoring in particular. We have shown how the problems can be overcome and we discussed a full implementation of a suitable tool set. With the help of this tool set two thorough case studies have been done and we have presented a summary of the results including associations that could be found from the recorded data.

This chapter summarizes what was found over the course of the work on this dissertation and from the evaluation of the data from these studies.

Long term monitoring is vital

The project succeeded in the sense that a robust tool could be created that is able to monitor large amount of synchronized, meaningful data samples in a highly distributed environment. From that raw data aggregated log records are stored in a database. These information proved to be very useful for systems management activities. In contrast to short-term observations of a few hours long-term monitoring can reveal the cyclical up's and down's in the load that is put on the systems.

In addition it became possible to view and examine parameters that are a combination of the load from different applications and - to some degree - the data can point to relations between different software products and layers within the application structure.

Valuable information for planing and sizing decision can be retrieved: By achieving a better understanding about the number of existing and active users and their behavior SRVMONPM makes it possible to show which configuration parameters are influenced by the number of users and which are not. It can be observed and documented which resources are used near their limits during what periods of the week and it is possible to find slots where the system is capable of handling more load. That is a vital input for planing and sizing activities.

Because the tool documents the changes of parameters over a long time general trends and technical architecture issues within one computer system can be observed and visualized for several purposes. To some degree findings from that documentation can be applied to similar systems.

The tool proved to be stable and efficient

During the case studies - some of them took several years - the tool run without interruptions or faults. The stability mechanisms that were introduced over time, showed to be robust and fault-tolerant. They can be used in a real 7x24 mode. But that was not for free. In contrary, this topic was underestimated in the beginning. It took a lot of serious effort to find a way to implement fault tolerant agent code that can handle all the problems that may arise in the host system.

That is not only clean coverage of all possible error return codes of functions called by the agent. In addition that means that the code may not even expect any call to a system function or server API ever returning any value. Control processes, timers, semaphores and parallel threads had to be used in order to catch every fault.

Simple dependencies do not exist

Despite of the assumptions found in several publications that it is easy to retrieve whatever information is needed or to calculate related metrics from the parameters

that can be measured, the data from the case studies did not yield the same result. In contrary the observed systems have been behaving very different at different times. Measured parameters, even if they depend on other parameters, seem to be influenced by too many separate factors.

General model functions for deriving required metrics are either not existing or they are only true for a very limited period and a certain restricted environment. As real-world environments are much more complex than laboratory conditions it was not possible to proof dependencies as those usually used for building the basis for many models or deterministic analysis. Especially the number of active users is hardly related with measured workloads. The data confirmed the speculation of several authors that modeling techniques cannot be applied easily to distributed systems.

No "automatic" model

The consequence of the finding above is that against our initial expectation it was not possible to find enough associations between different resource attributes that it would make sense to generate a model engine for processing association tables based on the results. There are a number of reasons why this effort has not yield the expected results:

First, the behavior and relation between different components seem to change over time. Whenever they are examined they are different. The often cited and used assumptions, that important parameters of a computer system are a function of other parameters **all the time** could not be validated by the recorded data.

Second, only simple relations could be checked (one parameter depends on a second parameter). While the detection algorithm theoretically would be able to check whether a parameter is associtaed to a group of parameters that did not work in practice.

On one hand, there is the problem that some important data about the internal status of the software (including the operating system) cannot be monitored. On the other hand, PC servers (that process requests from a wide range of very individual and different PC clients) behave much more stochastic than classical main frame computers with their batch oriented data processing.

Lack of support for (load) monitoring

As mentioned before, a major problem concerning analyzing the data is the fact that server applications do not report necessary status information about their resources. What is reported and what is not reported is not thought out very well. Our impression after four years of measuring data and adapting several products to the monitoring infrastructure is that the supply with meaningful statistical data to help in managing the application is an after-thought, at best, and not an essential part of the development effort.

Often it is difficult to understand, why certain information is provided and other equally important information is not. One striking example is when an network application reports the number of buffers that were missing or could not be allocated during a request, but it does not tell how many buffers are in use or free during normal operation. Therefore an administrator never knows how critical the operational status is until a problem occurs. Being short of any other information, and because the corresponding parameter is only defined during startup, the only thing an administrator can do is to oversize the parameter and hope that the application will never hit the limit. Why not report the number of buffers used and/or the number of buffers still available for other requests?. Only that insight would give the tool an oppertunity to detect the behavior of the system, correlate this with other conditions and help to react in problem situations.

In the scientific area monitoring is most often associated with debugging distributed applications as a part of software development. The fact that software has to be operated in a complex environment and that the organization that has to do it needs support and tools is still no accepted. Exceptions to that are applications that were ported from the UNIX world, e.g. the TCP/IP protocol stack or the latest version of DB2. There you get a lot of information about the current workload and internal status. Obviously, many years in distributed multi-user environments sharpened the sense for the many problems in that area. This may explain why monitoring is totally neglected in most papers and books. Most writers, especially from universities, are very familiar with UNIX but do not know other computing platforms. In commercial environments UNIX is not very popular and you have to deal with other platforms and other mind sets. OS/2 differs significantly from UNIX and we addressed several problems with the PC based server operating system.

No relevant information about resources is provided

The simple fact that a server application provides resources to others and consumes resources from subsystems is not reflected in the information you get from applications. Based on the experiences above basic prerequisites for a successful load management are formulated in section “Monitoring Demands”.

Software not ready for 7x24 monitoring

A very annoying problem was the fact that application servers, although they may provide an interface for monitoring, are not prepared for the existence of a monitor. On certain conditions the use of an API function will not return an error code. Instead it brings up a pop-up window on the screen and blocks all further processing waiting for a user to respond. This are occasions when the single-user, "personal computing" paradim of most PC software shows up in server software. Robust data processing on a server (and monitoring as part of that) must be able to execute unattended that is without the assumption that a user will look at and react on such messages. Another problem area is that a server cannot perform certain operations as long as the monitor is active and uses the interface of the software.

OS/2 is a suitable platform

Even with low-level OS/2 machines departmental environments can be handled easily. Current PCs can handle large enterprise environments. The resource needs of SRVMONPM are no match compared to mainframe or UNIX-based solutions. Thus, this work shows how a complex systems management application can be implemented in a way that does not demand lots of expensive hardware to monitor a large amount of "clients" and process their data.

Though OS/2 is no longer popular in the marketplace most results can be applied to Windows NT based server systems as well. Besides the close relationship between the two operating systems, both imply the use of a large number of (relatively) cheap machines which are distributed over the enterprise. Both handle resource management, networking and server application control in very much the same way.

Over the course of this work a general decline in the importance of departmental servers could be observed. Many functions are move back into centers where more sophisticated server applications based on internet technology (web servers with CGI and Java) and groupware software (like Lotus Domino) offer more control and management for enterprise data.

Technical Observations

The following paragraphs summarize major findings from the analysis of the data from the case studies (see chapter 2.3 for major findings and hypoteses about the reasons for the results of the measurements):

- A number of data items depend on office hours. They are of a cyclic nature. That came to no surprise. However, two things come to mind: it is not true that everybody comes between 7:00 and 9:00 in the morning and starts to work with the PC. In reality the load of "begin of day" work is distributed over the whole morning. The maximum is reached during 12:30 and 14:00 and then begins to decrease again. The same behavior could be noticed in several environments, even with relatively strict working time regulations. To that respect this load behavior is different to the statistics that are available for some internet servers that show two peaks: one around 10:00 and the second around 14:00.

  One possible consequence is that a significant percentage of people do not start their PC in the morning but do some other work first. A related experiment showed that this kind of "workload balancing" is an important factor for the operation of file servers that share application code. If all employees of a medium or large location would start their favorite applications (which range in the size of few to many MBs today) nearly at the same time, even fast LAN hardware would become a bottleneck that would paralyze the operation of the system for hours.

- Only a few resource attributes show a direct association (correlation) with the number of users. It follows that only a few metrics depend on the number of active users. The number of users does not seem to be a relevant parameter for sizing server capacity. Active users are users that have logged on to the domain and that means that at least one session to one server has been established by each user. Because enduser work on client PCs it is not possible to measure the number of people actually using one of the servers in a domain.

- The majority of data items do not even correlate with work hours. Thus, it is not likely that they do depend on user activities.

- The load that is put on file and print servers by end users is rather low and has even been decreasing over the course of the study. There are two explanations for that: First, loading applications usually happens once a day; often there is no further access to the server while the application is running. Second, there is a massive trend to the use of laptop computers; therefore all the application code is installed at the laptop and no server is needed anymore.

- The load on the domain controller and the backup domain controller is shared equally. This can be seen from the number of users which are handled by each of the hosts. There is no measurable resource consumption (e.g. CPU) on these kind of servers.

- Monday is the day of highest server usage. The number of connections decrease over the week with Friday being the working day with the lowest number of connections. Other values, like amount of data sent over the network indicate the same.

- Within each day the time from 10:00 to 12:00 o'clock has the most activities. Note, that this is not the time with the maximum of active users.

- The main resource that a server provides to remote machines is its disk space. CPU, memory and network resources usually are less significant and their use is rather stable.

- Automated maintenance activities put more load on a system then normal user activities.

- Blocked printer queues often remain blocked for hours; obviously people do not care about their printouts.

- Connections to servers are used mainly after the connection was established. Then the connection becomes idle and, eventually, is closed by the server.

- The number of real users of a server system is (usually) much lower than the expected or planned number. Sometimes even administrators were astonished by the real numbers. Not everybody uses the system all the time.

- No dependencies between resources on different hosts were detected. Each host seems to be independent from the status/activity of other hosts.

9.1. Monitoring Demands

Concerning a smooth operation of a tool like SRVMONPM, a number of open problems still exist:

- API functions that are called against a server, assume an interactive application. Some of them display an error window on the screen of the host machine and wait for user input instead of returning an error code. That blocks the agent and in some cases the process cannot be terminated because of the open error window.

  As mentioned above robust server software must be able to run unattended. That includes the ability to handle any error situation.

- Some server software cannot perform all of its function while a monitor is active (e.g. backup).

  Server software has to be able to operate all the day. It must not be influenced by a monitor that uses its interface.

- Interfaces to retrieve workload data are commonly end-user oriented: sometimes there is no suitable interface for a software monitor. The information that is provided should be consistent and give necessary insight to the status of the software and its resources.

Request For Improvements

With several years of experiences with monitoring and the behavior of server software and its users we still see the need for improvements in the area of server software in order to enable profound workload monitoring and - as a later step - workload management.

First, server software must clearly define the resources it needs from other subsystems and what (internal or external) resources it offers to others. Status information about both types should be maintained. A monitor must be able to access that data via an API. Whether this API is used and the way how this is done must not influence the server application in any aspect.

Given that prerequisites, a workload monitor, like the one described in this paper, can do its task. Dependencies and correlations between different server applications can be discovered and modeled.

Second, the server applications should support some external control about resource consumption. Again an API should exist that lets a server application allocate or free needed resources or make available more or less of the resources its provides.

Third, server applications should support the construction of a "server cluster" that can be used to ship service requests to a machine out of a set of possible choices. Of course, without the proper support by the operating system it would be extremely difficult to accomplish that goal.

An external tool - the workload balancing control mechanism - must be in charge to decide where to route an incoming request.

References

[1] Bruce Elbert, Bobby Martyna, *Client/Server Computing*, Artech House Inc., 1994

[2] IBM Inc, *Local Area Network Technical Reference*, SC30-3383-03, 4th edition, Dec. 1990

[3] Wolfgang Becker, *Dynamische adaptive Lastbalancierung für große, heterogene konkurrierende Anwendungen*, Doktorarbeit an der Universität Stuttgart, 1995

[4] M. Haas, W. Zorn, *Methodische Leistungsanalyse von Rechensystemen*, R. Oldenbourg Verlag, 1995

[5] P.G. Harrison, N.M. Patel, *Performance Modelling of Communication Networks and Computer Architectures*, Addison-Wesley, 1993

[6] H. Weber, *Einführung in die Wahrscheinlichkeitsrechnung und Statistik für Ingenieure*, Teubner 1983

[7] D. Menascé, V. Almeida, L. Dowdy, *Capacity Planning and Performance Modeling*, Prentice Hall, 1994

[8] D. Ferrari, G. Serazzi, A. Zeigner, *Measurements and Tuning of Computer Systems*, Prentice Hall, Englewood Cliffs, N.J., 1983

[9] Computer Measurement Group (CMG), at www.cmg.org

[10] C. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990

[11] R. Jain, *The Art of Computer System Performance: Analysis, Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991

[12] S. Lam, S. and K. Chan, *Computer Capacity Planning: Theory and Practice*, Academic Press, 1987

[13] H. Letmanyi, *Guide on Workload Forecasting*, Special Publication 500-123, Computer Science and Technology, National Bureau of Standards, Washington, D.C., 1985

[14] C. Rose, *A measurement procedure for queueing network models of computer systems*, ACM Computing Surveys, Vol. 10, No. 3, 1978

[15] I. Borovits, S. Neumann, *Computer System Performance Evaluation*, Lexington Books, 1979

[16] J. Cady, B. Howarth, *Computer System Performance Management and Capcity Planning*, Prentice Hall, Australia, 1990

[17] M. Loukides, *System Performance Tuning, A Nutshell Handbook*, O'Reilly & Associates, 1991

[18] Sun Microsystems, *The BSD 4.2 System, Programmer's Guide*, Sun, 1986

[19] U. Lackner, *Entwicklung eines wissensbasierten Systems zur Leistungsnalyse von DV-Systemen*, PIK - Praxis der Informationsverarbeitung, Nr. 3, 1986

[20] R. Hofman, R. Klar, N. Luttenberger, B. Mohr, G. Werner, *An approach to Monitoring and Modeling of Multiprocessor and Multicomputer Systems*, Proceedings of the IFIP TC 7/WG 7.3 International Seminar on Performance of Distributed and Parallel Systems, Elsevier Publishing Company, 1988

[21] M. Raynal, J.-M. Helary, *Synchronization and control of distributed systems and programs*, Wiley, 1990

[22] G. Nagl, F. Maybaum, H. U. Struve, M. Haas, *VERONIKA - Verteiltes BS2000-Monitorkonzept Karlsruhe*, FZI Karlsruhe, 1992

[23] Candle, OMEGAMON für MVS, *Effizientes MVS-Performance Management und OMEGACENTER*, 1990

[24] SNI, *SM2 V10.0A Software Monitor, User's manual*, SNI, 1991

[25] SNI, *SM2-PA V1.0 SM2-Programmanalysator, User's manual*, SNI, 1991

[26] Distributed Management Taskforce (DMTF), Whitepapers on *the Common Information Model*, www.dmtf.org

[27] IBM, *IBM Systems Monitor: Anatomy of a smart Agent*, Second Edition, IBM ITSO, 1996

[28] A. Hoetzel et al., *Understanding RS/6000 Performance and Sizing*, IBM ITSO Redbook, 1997

[29] S. Suhy, *Performance Tuning Windows NT*, Microsoft, 1998, (www2.slac.stanford.edu/winnt/perftune.htm)

[30] H. Merrill, *MXG Guide - Merril's Expanded Guide to Computer Performance Evaluation using the SAS System*, Merrill Consultants, 1997

[31] M. Woodside, C. Schramm, *Complex Performance Measurements with NICE*, Carleton University, Canada, 1995

[32] W. G. Pope, *Planning Domino Email Servers using Notes Transactions*, IBM Watson Research, 1998

[33] W.G. Pope, *A Planning Model for Lotus Notes Applications*, IBM Watson Research, 1998

[34] T. Winkelmann, G. Strasser et. al., *Distributed Systems Monitoring*, IBM ITSO, 1999/2000

[35] M. G. Kendall, *Rank Correlation Methods*. Fourth edition. London: Griffin, 1970

[36] M. Johnson, *The Application Response Measurement API, Version 2*, CMG, www.cmg.org/regions/cmgarmw/marcarm.html, 1997

[37] M. Katchabaw, S. Howard, H. Lutfiyya, A. Marshall, M. Bauer, *Making distributed applications manageable through instrumentation*, The Journal of Systems and Software 45, p81-97, Elsevier Science, 1999

[38] J. Hartung, B. Elpert, K.H. Klösener, Lehr- und Handbuch der angewandten Statistik, 10. Auflage, Oldenburg, 1993

Further Readings

X.400 Systems Management Standards

Robert Orfali, Dan Harkey, *Client/Server Survival Guide*, VNR, 1994

IBM Inc, *Workload Manager Presentation Guide*, ZZ81-0335-00, May 1994

Shun Yan Cheung, Vaidy S. Sunderam, *Performance of Barrier Synchronization Methods in a Multi-Access Network*, Emory University, Atlanta

Paul A. Fishwick, *SIMPACK: Getting Started with Simulation Programming in C and C++*, University of Florida, Gainsville, 1995

IBM Inc, *CICS Workload Management Using CICSPlex SM and the MVS/ESA Workload Manager*, GG24-4286-00, December 1994

B. Walke, O. Spaniol, *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, Springer Verlag, (1993)

J. W. Wong, *Performance Modeling of ATM-Based Networks*, University of Waterloo, Ontario, Canada

H Wabing, G. Kotsis, G. Haring, *Performance Prediction of Parallel Programs*, University of Vienna

B. Meyer, C. Popien, *Modellierungs- und Bewertungskonzepte für ODP-Architekturen*, RWTH Aachen

T. Heinrichs, B. Bärk, J. c. Strelen, *Wartezeiten für Pollingsysteme mittels numerischer Modelle*, Rheinische Friedrich-Wilhelms-Universität Bonn

M. N. Huber, V. H. Tegtmeyer, *Bandwidth Management of Virtual Paths - Performance Versus Control Aspects*, Siemens AG

R. Hofmann, *Gemeinsame Zeitskala für lokale Ereignisspuren*, Universität Erlangen

V. Klinger, *Ein hybrider Computerbus-Monitor*, Universität Hamburg-Harburg

J. Aman, C.K. Eilert, D. Emmes, P. Yocom, D. Dillenberger, *Adaptive algorithms for managing a distributed data processing workload*, Systems Journal, Vol. 36, No.2, IBM, 1997

Hans-Ulrich Heiß, *Überlast in Rechensystemen*, Informatik Fachberichte, Springer 1988

L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, John Wiley and Sons, NY 1975

Shui F. Lam, K. Hung Chan, *Computer Capacity Planing*, Academic Press, Inc., 1987

U. Herzog, M. Paterek, *Messung, Modellierung und Bewertung von Rechensystemen*, Springer Verlag 1987

Modelling Techniques and Tools for Performance Analysis, Elsevier Science Publishers 1986

IBM Inc., *OS/390 Resource Management Facility Report Analyses*, SC28-1950-02, IBM, 1997

K.-L. Wu, P.S. Yu, A. Ballmann, *SpeedTracer: A Web usage mining and analyses tool*, IBM Systems Journal, Vol.37, No. 1, 1998

T. Lo, *Computer workload forecasting techniques: a tutorial*, Proceedings of the International Conference on Computer Capacity Management, 1980

R. Weicker, *An overview of common benchmarks*, IEEE Computer, 1980

I. Schulte, *SINIX-Benchmarking - aktuelle Entwicklungen and applications*, SAVE Tagung, 1993

J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann, 1991

A. M. Law, W. D. Kelton, *Simulation Modelind and Techniques*, 2nd ed., McGraw-Hill, 1990

M. H. MacDougall, *Simulating Computer Systems: Techniques and Tools*, MIT Press, 1987

S. M. Ross, *A Course in Simulation*, Macmillan, 1990

L. Kleinrock, *Queueing Systems, Volume I*, John Wiley and Sons, 1975

L. Kleinrock, *Queueing Systems, Volume II*, John Wiley and Sons, 1976

J. D. C. Little, *A proof of the queueing formula* $L = \lambda W$, Operations Research, Vol. 9, 1961

P. J. Denning, J. P. Buzen, *The operational analysis of queueing network models*, Computing Surveys, Vol. 10, No. 3, 1978

E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, *Quantitaive System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, 1984

M. Resier, S. S. Lavenberg, *Mean value analysis of closed multichain queuing networks*, Journal of the ACM, Vol. 27, No. 2, 1980

K. K. Chandy, U. Herzog, L. S. Woo, *Parametric analysis of queueing networks*, IBM Journal of Research and Development, Vol. 19, No. 1, 1975

R. R. Muntz, J. W. Wong, *Asymptotic properties of closed queueing network models*, Proceedings of the 8th Princeton Conference on Information Sciences and Systems, 1974

J. Zahorjan, K. C. Sevcik, D. L. Eager, B. I. Galler, *Balanced job bound analysis of queueing networks*, Communications of the ACM, Vol. 25, No. 2, 1982

F. Baskett, K. Chandy, R. Muntz, F. Palacios, *Open, closed, and mixed networks of queues with different classes of customers*, Journal of the ACM, Vol. 22, No. 2, 1975

J. C. Lowery, *Calibration and predictive modeling of computer systems*, Ph.D. dissertation, Vanderbilt University, 1992

J. P. Buzen, A. Shum, *Model calibration*, Proceedings of the 1989 CMG Conference, 1989

J. Flowers, L. W. Dowdy, *A comparison of calibration techniques for queueing network models*, Proceedings of the 1989 CMG Conference, 1989

J. Maierhofer, H. Schmitt, *Introduction to BORIS - a block-oriented Interactive Simulation System*, Siemens, 1982

B. Page, *Diskrete Simulation*, Springer Verlag, 1992

ITR: SNAPL/1 Language Reference Manual, Instituut vir Toegepaste Rekenaarwetenskap, University of Stellenbosch, South Africa, 1982

Siemens, *Modellierung und Simulation informationstechnischer Systeme*, Siemens, 1985

C. H. Sauer, E. A. MacNair, J. F. Kurose, *The Reasearch Queueing Package V2: CMS user's guide*, IBM Research Division, 1982

Fromm, *Modellierung von Rechnersystemen*, Universitäts Karlsruhe, 1990

C. Hrischuk, J. Rolia, C. M. Woodside, *Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype*, Carleton University, Canada, 1995

P. Zave, *An insider evaluation of paisley*, IEEE Transactions on Software Engeneering, 1991

V. Berzins, Luqi, *Rapidly prototyping real time systems*, IEEE Software, 1988

G. M. Karam, *The MLog user's guide*, Carleton University, 1992

C. M. Woodside, J. E. Neilson, D. C. Petriu, S. Majumdar, *The stochastic rendezvous network model for performance of synchronous multi-tasking distributed software*, IEEE Transactions on Software Engeneering, 1994

J. A. Rolia, *Predicting the Performance of Software Systems*, University of Toronto, 1992

D. Ferrari, *Computer Systems Performance Evaluation*, Prentice Hall, 1978

J. W. Boyse, D. R. Warn, *A straightforward Model for Computer Performance Prediction*, Computing Surveys, Vol. 7, No. 2, 1975

J. P. Buzen, *Fundamental operational laws of computer system performance*, Acta Informatica 7, Vol. 2, 1976

J. P. Buzen, P. J. Denning, *Operational Teatment of Queue Distributions and Mean Value Analysis*, Computer Performance, Vol. 1, No. 1, 1980

D. Menascé, V. Almeida, *Capacity Planning for Web Performance, Metrics, Models and Methods*, Prentice Hall, 1998

Appendices

Appendix A. Terms Used In This Paper

Term	Description
7x24	continous (uninterrupted) operation of a computer system. That includes all measures taken to keep up operation during maintainance and service activities.
Class Definition	is the data structure that describes the (common) attributes of all *monitor objects* of the same class. Beside other things it contains the *field definitions*, which describe all attributes (data items) that are provided by one monitor object about an associated resource.
Collection Server	is the component that is used to **store and forward** information. Workload data are received from *monitoring agents* and are sent to an *intermediate server*.
Field Definition	describes one attribute about a give *resource*. This contains the name of the attribute, it's unit and the type (normalized, delta or absolute)
Host	is the machine where *server applications* and *monitor agents* execute.
Infrastructure Design	is the task of understanding the requirments against the needed infrastructure of a system, insulating the components that will make up the system, planning the capacity and sizing the components, choosing available products and compose everything to a running system. In contrast to software design there are no common, publically known methods. Some companies, like IBM, have developed their own methods.
Intermediate Server	is a collection server that can be accessed by the front end application (*workload manager*)
Machine Definition	is the data structure that is used to store and organize all workload data for a certain host.
Module Presentation Object	is the primary means of communication between *agent* and *monitor module*. It provides a list of *monitor factory objects* to the agent.
Monitoring Agent	is the program that executes at a target host and which controls the operation of *resource monitor objects.*
Monitor Class	represents the knowledge on how to retrieve and process workload information about a particular resource. An important task of a monitor class is its object factory. The class is responsible for creating monitor objects for each monitored resource. Thus,

Term	Description
	it is an important part during the resource discovery process on a monitored client.
Monitor Domain	is the set of all agents that report their data to the same physical collection (or intermediate) server
(Monitor) Factory Object	is the object that implements the knowledge about detection of resources and the creation of *monitor objects.*
Monitor Module	is a dynamic link library that contains all neccessary code to detect and access resource information and it provides some information to control presentation and analyses; the code in the module is structured in monitoring classes
Monitor Object	abreviation for *resource monitor object*
Resource Monitor Object	is a C++ object which has the knowledge and data to query and process the workload information about an existing instance of a computer resource; e.g. partition index C on the disk array of server A.; it monitors a resource at the local host
Resource Proxy	is a resource monitoring object that monitors a resource on a remote host
Server Application Target Application	is a program which runs on a server and which is monitored (that is, load and performance information is retrieved from that application)
Workload Agent	is a small program which runs on a machine, which is to be monitored, and which is the platform for monitor objects.
Workload Manager	is the central front end of the application. It collects and processes workload data.

Appendix B. Abbreviations

Abbreviation	Explanation or full wording
ARM	Application Response Measurement
CIM	Common Information Model of the DMTF
DLL	dynamic link library
DMTF	Distributed Management Taskforce
FFST	First Failure Support Technology
GUI	graphical user interface
IPC	inter process communication
NCB	network control block; a data structure which is used in the NETBIOS communication stack to issue commands to control basic network operations
NOS	network operation system e.g: the IBM LAN SERVER product
OS	operating system
PM	Presentation Manager that is the window system of OS/2
RMF	IBM Resource Management Facility
sar	UNIX System Activity Reporter
SIP	Unisys Software Instrumentation
SMF	IBM System Management Facility
WMI	Windows Measurement Instrumentation

Appendix C. Adding New Monitor Classes

At many places throughout this document it was mentioned that new monitor modules can be added to the system and that they will be integrated seamlessly. This chapter describes how this can be accomplished by a C++ developer.

Developer's Toolkit

In order to enable a developer to write a new module a developer's kit is part of the monitoring package. It consist of a number of include and library files and two example modules which demonstrate the creation of some simple classes and which may be used as a boilerplate for new modules.

Core Classes And Their Methods

The following paragraphs demonstrate the usage of one sample class and what a developer has to do to create a new monitor module. Code fragments from the example file SRVMLSM1.CPP and SRVMLSM1.HPP are used in the examples below.

While the examples keep the class declarations in separate include files this is not a prerequisite of the application. If the declarations are not used in other source files they may as well reside in the source file of the monitor module.

Note that two DLLs will be created: one that will be used at the agent and which has to be bound to any libraries of the application to be monitored; and one which will be used at the manager and which must not be bound to such libraries (if that creates the need to link to application specific DLLs that will not be available at the manager station). A developer may choose to write to different source files for that purpose. But it is recommended that only one source file is maintained and that precompiler directives are used to control the generation of the necessary code. Therefore the descriptions below will point out in which DLL the code fragment is needed.

First we will look at the declarations which are needed to build the module. Two classes must be provided for monitoring: the retriever object, which is a subclass of the core class **RETRIEVER**, and a monitor class, which is subclass of the core class **CLASS_DEF**.

```
#ifdef AGENT

typedef class MONSMP_LOCAL_STATIC: public RETRIEVER {
   unsigned short CurrValue;
public:
   MONSMP_LOCAL_STATIC (PMACHINE pS, PCLASS_DEF pClass);
   virtual void  InitData (void);
   virtual void  NewValue (DATAITEM aVal[VALUE_ARRAY]);
} *PMONSMP_LOCAL_STATIC;


typedef class MONSMP_LOCAL_DYNAMIC: public RETRIEVER {
   unsigned short CurrValue;
public:
   MONSMP_LOCAL_DYNAMIC (PMACHINE pS, PCLASS_DEF pClass, char *n);
   virtual void  InitData (void);
   virtual void  NewValue (DATAITEM aVal[VALUE_ARRAY]);
} *PMONSMP_LOCAL_DYNAMIC;

#endif
```

The example defines two types of retriever objects. Because retriever objects are used only by the agent the *ifdef* directive is used to control the generation of this code

fragment. Each retriever object class **must** declare a constructor method and the virtual method **NewValue**, which is responsible for retrieving the load data. All other methods are optional. The use of the method **InitData**, which is called once before monitoring begins, may be useful for initialization that cannot be done during the creation of the object but must be done before the actual monitoring.

```
typedef class CLASS_SMP_LOCAL_STATIC: public CLASS_DEF {
public:
   CLASS_SMP_LOCAL_STATIC (void);
#ifdef AGENT
   virtual PRETRIEVER BeginEnumObjects (PMACHINE pServer);
   virtual PRETRIEVER GetNextObject (PMACHINE pServer);
#else
   virtual int getDefaultViewDefinition (int Index,
                                         DEFAULT_VIEW_DEF
&DefViewData);
#endif
} *PCLASS_SMP_LOCAL_STATIC;


typedef class CLASS_SMP_LOCAL_DYNAMIC: public CLASS_DEF {
public:
   CLASS_SMP_LOCAL_DYNAMIC (void);
#ifdef AGENT
   virtual PRETRIEVER BeginEnumObjects (PMACHINE pServer);
   virtual PRETRIEVER GetNextObject (PMACHINE pServer);
#endif
} *PCLASS_SMP_LOCAL_DYNAMIC;
```

For each retriever object a monitor class must be defined as well. The class definitions are shared between agent and manager code. Each class must define the following methods:

- The constructor must be defined in any case (agent and manager).
- At the agent the virtual methods **BeginEnumObjects** and **GetNextObject** must be defined; These methods return retriever objects. GetNextObject will be called as long as it returns a pointer to an object. If no more objects can be created (there are no more resources to be monitored) the method returns NULL.
- At the manager the virtual method **getDefaultViewDefinition** should be defined. It returns defaults for the view definition (see below). If the method is not provided **no** view definition will be created and the user must do that manually.

```
#define INCL_WIN

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <os2.h>

#include <srvmdevl.hpp>
#include "srvmlsm1.hpp"
```

It is very likely that a monitor module needs the include files in the example above.

```
static FIELD_DEF
      LocalStaticFlds[] =  {
                              { FLDDEF_ABSOLUTE, "Item A1" },
                              { FLDDEF_ABSOLUTE | FLDDEF_NORMALIZED, "Item
B1" } },
      LocalDynamicFlds[] = {
                              { FLDDEF_ABSOLUTE, "Item A2" },
                              { FLDDEF_ABSOLUTE, "Item B2" },
                              { FLDDEF_ABSOLUTE, "Item C2" },
                              { FLDDEF_ABSOLUTE, "Item D2" } };

static char *DemoRole = "Demonstration";
static char *ClassRole = DemoRole;
```

The **FIELD_DEF** structure defines data items of a monitor class. The definitions consists of a number of flags and the name of the data item. The flags are important for the processing and display of the data and it is very important that the developer defines them correctly. Following flags are available:

- **FLDDEF_DELTA**:

  The value of the data item is the delta of the real data within one sample interval. Usually running counters are represented by deltas.

- **FLDDEF_ABSOLUTE**:

  The value of the item contains the real data as they are retrieved from/about the resource. No deltas are calculated.

  FLDDEF_DELTA and *FLDDEF_ABSOLUTE* are exclusive - only one of the can be used for one item.

- **FLDDEF_NORMALIZED**:

  The value of the item was normalized into a certain value range. For example percentages are normalized values. The value range is defined by the retriever code and it is not known to the monitoring application.

  Normalized values are treated differently in total pages of view notebooks: the manager does not calculate the sum of all items but an average instead.

The field definitions are used for the monitor class construction (see below). The same is true for the definition of the role (**ClassRole**).

The next code fragment contains all the code to implement one monitor class.

```
/*****************************************************************************
**
** class LOCAL_STATIC
**
** example for a monitor class which provides one retriever object for a
server.
** The object is added to the "global server" views. Note the following things:
**
** + the class attribute CLASS_TYPE_GLOBAL is used; consequently the data are
**   displayed in global views
**
** + only one instance of a retriever object must be created per server;
**   therefore the method BeginEnumObjects creates and returns one object and
**   the method GetNextObject returns NULL to signal that no more objects of
**   this class will be created for the server
**
*****************************************************************************/


CLASS_SMP_LOCAL_STATIC::CLASS_SMP_LOCAL_STATIC (void):
   CLASS_DEF ("smp_local_static",               // internal class key
              "Local & Static",                 // name in the user interface
              CLASS_TYPE_GLOBAL,                // class attributes
              2,                                // number of data items
              LocalStaticFlds,                  // description of data items
              ClassRole)
{
}
```

The constructor of the core class needs the following arguments:

- the class key; the key must not contain blanks (white space character)
- the name of the class, which is used in the user interface
- class flags (see below)
- the number of data items that are provided by retriever objects of that class
- an array of definitions of the data items
- the role to which the class belongs

Following class flags can be used:

- **CLASS_TYPE_DYNAMIC**:

  The monitor class is written for a number of resources which may vary in number and which may appear and disappear over time. Therefore the class is called *dynamic*. The application expects that the method *GetNextObject* returns several objects. Nevertheless it is possible that the class does not return any object if no resource does exist at a certain moment.

  The methods *BeginEnumObjects* and *GetNetxObject* will be called in regular intervals to check whether new resources appeared in the system or existing objects became obsolete.

  An example of such a class are the LAN server (shared) net names (connections): new net names can be created dynamically and may disappear as well. The number of net names cannot be foreseen and is dynamic.

- **CLASS_TYPE_GLOBAL**:

  The monitor class is the opposite of a dynamic class: it monitors certain aspects of an application or system, which is independent of the existence of resources. Therefore exactly one retriever object is created. Views of this object are accessed via the menu of the "global" icon in the monitor selection.

The method *BeginEnumObjects* is called once. *GetNextObject* is called for consistency reasons but it is expected that the method returns NULL.

An example of such a class is the DB2 Database Engine: if the database engine is active on a machine, it provides load information independent of the existence of databases (while monitoring *database instances* requires a dynamic monitor class).

```
#ifdef AGENT

PRETRIEVER
CLASS_SMP_LOCAL_STATIC::BeginEnumObjects (PMACHINE pServer)
{
   return (new MONSMP_LOCAL_STATIC (pServer, this));
}


PRETRIEVER
CLASS_SMP_LOCAL_STATIC::GetNextObject (PMACHINE pServer)
{
   return (NULL);
}

#else
```

BeginEnumObjects creates and returns a retriever object. *GetNextObject* returns NULL because it is a **global** class.

```
#else

#define WMID_DEMO    1540

int
CLASS_SMP_LOCAL_STATIC::getDefaultViewDefinition (
                            int Index,
                            DEFAULT_VIEW_DEF &DefViewData)
{
   switch (Index) {
   case 0:
      strcpy (DefViewData.ViewName,  "Demo");
      strcpy (DefViewData.MenuEntry, "Demo");
      strcpy (DefViewData.Key, "A");
      DefViewData.MenuID = WMID_DEMO;
      DefViewData.GraphNum = 2;
      DefViewData.GraphDef[0].DataIndex = 0;
      DefViewData.GraphDef[0].LensType  = VIEW_LENS_NORMAL;
      DefViewData.GraphDef[1].DataIndex = 1;
      DefViewData.GraphDef[1].LensType  = VIEW_LENS_NORMAL;
      return TRUE;
   default:
     break;
   } /* endswitch */
   return FALSE;
}

#endif
```

The method **getDefaultViewDefinition** is used at the manager when it detects a class for which no view definition does exist and no entry in the view creation table can be found. The view creation table contains an entry for each class for which the default views were created. If the user removes the views later the manager will not recreate the views again.

The method is called several times while incrementing the parameter **Index** until it returns FALSE. During each invocation the method has to fill the view definition structure. The structure consists of:

- the name of the view
- the menu entry by which the view is accessible to the user
- menu id (it must be unique within one PM menu)
- a unique[44] "key"
- the number of graph lines; maximal 6 lines can be used in a view
- up to six line definitions which consist of:
  - the index of the data item which is represented by the line
  - the lens which is used to display the item (use either **VIEW_LENS_NORMAL** or **VIEW_LENS_AVERAGE**; other settings should be done by the user via the view definition window)

```
/*****************************************************************************
**
** MONSMP_LOCAL_STATIC
**
** is a retriver object. It has to implement the way how data about the network
** or server will be retrieved
**
*****************************************************************************/

MONSMP_LOCAL_STATIC::MONSMP_LOCAL_STATIC (PMACHINE pS, PCLASS_DEF pClass):
      RETRIEVER (pS, pClass, pS -> Name)
{
} /* MONSMP_LOCAL_STATIC */


void
MONSMP_LOCAL_STATIC::InitData (void)
{
   CurrValue = 0;
} /* InitData */


void
MONSMP_LOCAL_STATIC::NewValue (DATAITEM aVal[VALUE_ARRAY])
{
   aVal[0]   = (CurrValue + rand () % 100) % 3000;
   aVal[1]   = (CurrValue + rand () % 100) % 3000;
   CurrValue = (CurrValue + rand () % 50)  % 3000;
   ErrorCode = 0;
} /* NewValue */
```

The implementation of the retriever object is extremely simple in this example. In reality the logic to retrieve the required information will be placed in the method **NewValue**.

[44] The key has to be unique within the same monitor class. Views of different monitor classes can reuse the same keys.

```
/*****************************************************************************
**
** class LOCAL_DYNAMIC
**
** example for a monitor class which provides a number of retriever object for a
** server.
** The object is added to the "server" views. Note the following things:
**
** + the class attribute CLASS_TYPE_GLOBAL is NOT used; consequently the data are
**   displayed in global views
**
** + the class attribute CLASS_TYPE_DYNAMIC is used; that means that the resources
**   on the network, which are monitored by this class, may appear or disappear
**   dynamically. Therefore from time to time the class will be asked to check
**   for new objects. Retriever objects of resources which do not longer exist are
**   not removed from the monitor engine. They have to return VALUE_UNDEFINED to
**   signal that no connection to resource does exist.
**
** + several instances of a retriever object will be created per server;
**   therefore the method BeginEnumObjects initialises the enumeration of objects
**   and uses the method GetNextObject to return the first retriever object.
**   The method GetNextObject creates more objects. Finally it returns NULL to
**   signal that no more objects of this class will be created for the server
**   Note:
**   Servername and class name and object name are used as keys for each object.
**   If the same object is created again during a refresh the monitor engine
**   checks for any existing key. If it finds a matching key the new object is
**   destructed immediatelly. Therefore you do not have to bother with the
**   existance of "old" retriever objects while you are creating new objects
**
*****************************************************************************/


CLASS_SMP_LOCAL_DYNAMIC::CLASS_SMP_LOCAL_DYNAMIC (void):
   CLASS_DEF ("smp_local_dynamic",              // internal class key
              "Local & Dynamic",                // name in the user interface
              CLASS_TYPE_DYNAMIC,               // class attributes
              4,                                // number of data items
              LocalDynamicFlds,                 // description of data items
              ClassRole)
{
}


PRETRIEVER
CLASS_SMP_LOCAL_DYNAMIC::BeginEnumObjects (PMACHINE pServer)
{
                                       // query number of entries
   Entries = 3;
   NextIndex = 0;
   return (GetNextObject (pServer));
}


static void
QueryNextObject (int Index, char *NameBuffer)
{
   sprintf (NameBuffer, "Obj %d", Index);
}


PRETRIEVER
CLASS_SMP_LOCAL_DYNAMIC::GetNextObject (PMACHINE pServer)
{
   char Buffer[20];
   PRETRIEVER pRet;

   while (NextIndex < Entries) {
                                       // query data of next retriever object
```

```
      QueryNextObject (NextIndex, Buffer);
      NextIndex++;
      pRet = new MONSMP_LOCAL_DYNAMIC (pServer, this, Buffer);
      if (pRet)
         return (pRet);
   }
   return (NULL);
}


MONSMP_LOCAL_DYNAMIC::MONSMP_LOCAL_DYNAMIC (
                              PMACHINE pS,
                              PCLASS_DEF pClass,
                              char *n):
      RETRIEVER (pS, pClass, n)
{
} /* MONSMP_LOCAL_DYNAMIC */


void
MONSMP_LOCAL_DYNAMIC::InitData (void)
{
} /* InitData */


void
MONSMP_LOCAL_DYNAMIC::NewValue (DATAITEM aVal[VALUE_ARRAY])
{
   int i;

   for (i = 0; i < 4; i++) {
      aVal[i]   = (CurrValue + rand () % 100) % 3000;
   } /* endfor */
   CurrValue = (CurrValue + rand () % 50)  % 3000;
   ErrorCode = 0;
} /* NewValue */
```

The implementation of a dynamic class is more complex - even in this very simple example. Note the following differences to the first example above:

1. *BeginEnumObjects* typically initializes the search for existing resources and then calls *GetNextObject*, which implements the resource lookup and monitor creation.

2. *GetNextObject* will rely on some C code to communicate with the target application and get information about available resources. In the example the function *QueryNextObject* simulates such a C code.

3. *GetNextObject* creates a retriever object. It must provide all relevant information to the constructor of the object. At the minimum this is the name of the new object because the objects needs that for its construction. It should be avoided that the new retriever objects has to access the target application again to ask for information about the resource. Usually *GetNextObject* will have that information right at hand.

4. The target application may provide a list of available resources as one piece of information. Because *GetNextObject* returns one retriever object with each invocation and it must be avoided that it asks for the same information many time, the class must use some static memory to store information about other resources for future use. This is simple because only one instance of the monitor class will exist in the system.
   At the same time it should be avoided to mix different classes. Each class must act as if there are no other monitor classes. It is not allowed that a class relies on information which has to be provided by another class.

5. Server (machine) name, class key and resource name are used to create a unique key for retriever objects (and related monitors). Therefore it is very important that this names are chosen in a way that guarantee uniqueness.

After the monitor classes and retriever objects are defined the only thing missing is the mechanism to export them to the load monitor application:

```
/*****************************************************************************
**
** class MODULE_PRESENTATION
**
** is used to provide the interface of the DLL to the monitor engine.
** One instance of this class must be created. It contains the list of monitor
** classes which are provided by the DLL.
**
*****************************************************************************/


class SMP_MODULE_PRESENTATION: public MODULE_PRESENTATION {
public:
   virtual void getClassDefinitions (PMODULE_CLASS_DEF pModClassDef);
   virtual int  getModuleCopyright  (char *TextBuffer, unsigned long
BufferSize) const;
#ifndef AGENT
   virtual int  getRoleDefinitions  (PMACHINE_ROLE pMachineRole[]);
#endif
} SmpModulePresentation;
```

The module must provide an object which is a subclass of **MODULE_PRESENTATION**. The virtual methods, which are shown in the example, may be overwritten.

```
static CLASS_SMP_LOCAL_STATIC  ClassLocalStatic;
static CLASS_SMP_LOCAL_DYNAMIC ClassLocalDynamic;


void
SMP_MODULE_PRESENTATION::getClassDefinitions (PMODULE_CLASS_DEF
pModClassDef)
{
   pModClassDef -> pClassArray[0] = &ClassLocalStatic;
   pModClassDef -> pClassArray[1] = &ClassLocalDynamic;
   pModClassDef -> nClasses = 2;
}


int
SMP_MODULE_PRESENTATION::getModuleCopyright  (char *TextBuffer, unsigned
long BufferSize) const
{
   sprintf (TextBuffer, "Sample Data Query, Version 1.0\n   by G nther
Strasser\n     (C) 1994, 1995\n");
   return TRUE;
} /* getModuleCopyright */


#define ICON_SERVER  2
#define ICON_STAT    3


int
SMP_MODULE_PRESENTATION::getRoleDefinitions  (PMACHINE_ROLE
pMachineRole[])
{
   pMachineRole[0] = new MACHINE_ROLE (DemoRole,
                                       (HPOINTER) WinLoadPointer
(HWND_DESKTOP, hMod, ICON_SERVER),
                                       (HPOINTER) WinLoadPointer
(HWND_DESKTOP, hMod, ICON_STAT));
   return 1;                            // returns number of defined roles
}
```

The method *getClassDefinitions* fills an array of pointers to classes. The module must make sure that the class instances still exist after the method returns. It is recommended that the class instances are created as static objects - like in the example. It would be possible to build them dynamically by the use of the **new** operator. This method **must** be implemented by each module (otherwise it would not export its class definitions).

getModuleCopyright fills some text into the text buffer. When any part of the monitor application is started the method is called for each class and the text is written to *stdout*.

getRoleDefinitions may be used to define new roles. A role consist of a name (**DemoRole**) and two icons: one for the global information and one which is used for each machine instance. The variable **hmod** is the module handle of the module and is provided as public member of MODULE_PRESENTATION.

```
/*****************************************************************************
**
** define module interface
**
** The function SmpQueryPresentation must be exported by the DLL.
** The name of the DLL and the name of the function have to be added to the
** file SRVMCODE.MOD in the working directory of SRVMONPM.EXE.
** After restarting the tools the new DLL will be loaded.
**
** Note:
** You may use any name for this function.
** Use C calling/naming convention for this function.
**
*****************************************************************************/

extern "C" {
   PMODULE_PRESENTATION _Export SmpQueryPresentation (void);
}


PMODULE_PRESENTATION _Export
SmpQueryPresentation (void)
{
   return (&SmpModulePresentation);
} /* SmpQueryPresentation */
```

The last part is the definition of a function - with C calling convention - that does nothing else then returning the address of the module presentation object. The name of the module and the function must be registered in the file SRVMCODE.MOD:

```
local          SRVMDFLT          DefQueryPresentation
remote         SRVMDFLT          DefQueryPresentation
local          SRVMLRMT          RemQueryPresentation
client         SRVMRRMT          RemQueryPresentation
local          SRVMLACS          RemQueryPresentation
client         SRVMRACS          RemQueryPresentation
local          SRVMLSRV          RemQueryPresentation
client         SRVMRSRV          RemQueryPresentation
local          SRVMLDB2          RemQueryPresentation
client         SRVMRDB2          RemQueryPresentation
local          SRVMLD22          RemQueryPresentation
client         SRVMRD22          RemQueryPresentation
```

Two entries are necessary: one with the keyword **local** which signals to the application that the module has to be used at the manager; the second with the keyword **client** if the module is used at the agent for monitoring of the agent's machine, or **remote** if the code accesses information about remote machines (the agent was started with one of the /s arguments).

Cookbook To Build Your Own Module

When creating a new module follow the steps below:

1. Define what data you want to add to the monitoring system and think about the way you can efficiently access them.

2. Encapsulate the data access in a thin C or C++ layer which may be used for testing and binding to the monitor module.

3. Copy a template.

4. Define the monitor classes by subclassing the base class and fill the required virtual functions as described above.

5. Build a module for use at the agent (with your data access layer) and a module for use at the manager (with resource and view definitions)

6. Add the names of your modules to the module registration file (SRVMCODE.MOD).

7. Test the module.

8. Check the ranges of the data items and calibrate them if necessary.

Distribute the new monitor module to all agents.

Appendix D. Important Classes of the Monitor Agent

DATAITEM

This class represents one piece of information that is returned by a monitor object for each item of the class. It is implemented as an unsigned short integer and some methods enforce the meaning of *overflowed* and *undefined* mode of the sample data.

Possible values are in the range of 0 and 65533 (hex 0xFFFD). The integer value hex 0xFFFE is interpreted as *overflowed* (VALUE_MAXIMUM) and hex 0xFFFF has the certain meaning *undefined* (VALUE_UNDEFINED) and, therefore, both do not represent a valid number.

There is a problem when the possible value range is much larger than 65533 and scaling is not applicable for all types of target machines. A good example is the number of files on a server: it depends on the number disks and the size and usage of each disk. It may be below 10.000 as well greater then 500.000. The only solution I see at the moment would be that the monitor agent decides about a scaling factor for each item and provides it together with the monitor object definition. The difficult part is the math at the manager: it would need to consider the different scaling values for each item of each object and all calculations and logging would have to be adjusted in some way. In the current implementation that is not done. Before a new class becomes operational I use it on several production server and "calibrate" it. That means that I add the scaling factor hardcoded in the retriever class.

```
typedef class DATAITEM {
   unsigned short Value;

public:
   DATAITEM (void)           { }
   DATAITEM (unsigned long x)
            { Value = (x == VALUE_UNDEFINED ?
                VALUE_UNDEFINED :
                (x > VALUE_MAXIMUM ? VALUE_MAXIMUM : x)); }

   operator unsigned short (void)
            { return Value; }
   DATAITEM & operator =  (unsigned long x)
            { Value = (x == VALUE_UNDEFINED ?
                VALUE_UNDEFINED :
                (x > VALUE_MAXIMUM ? VALUE_MAXIMUM : x));
              return *this; }
   DATAITEM & operator += (unsigned long x)
            { *this = Value + x; return *this; }
   DATAITEM & operator -= (unsigned long x)
            { *this = Value - x; return *this; }
} *PDATAITEM;
```

RETRIEVER

This class is the parent for all kind of objects which retrieve data from a system or application. A controller (usually part of the monitor agent) calls the method **NewValue** and the method fills the array of DATAITEMs - **aVal**. Each monitor object builds a unique key which is used for storage, lookup and availability checks.

A forward reference to CLASS_DEF is necessary because each retriever has a pointer to the class which defines the layout of the item that it returns.

```
typedef class CLASS_DEF *PCLASS_DEF;


typedef class RETRIEVER {
protected:
   STRING     Key;
   PCLASS_DEF pClass;
public:
   PMACHINE pServer;
   char     Name[OBJ_NAME_SIZE];
   int      ErrorCode;

   RETRIEVER (PMACHINE pS, PCLASS_DEF pCl, char *n);
   virtual void          InitData (void) = 0;
   virtual void          NewValue (DATAITEM aVal[VALUE_ARRAY]) = 0;
   virtual void          UpdateInfo (RETRIEVER *pObj) { }
           PCLASS_DEF    getClass (void) { return pClass; }
           STRING const& getKey (void) { return Key; }

   friend STRING const& key (RETRIEVER * const&);
} *PRETRIEVER;
```

FIELD_DEF

This structure defines one data item. It consists of the name of the item, which is used in GUI, and flags, which control the processing of the item data:

- FLDDEF_DELTA:

  the item is the difference between the current and the most recent value of the information, which can be retrieved from a system; this is used for counters which are constantly increased (e.g. number of bytes received since the system started)

- FLDDEF_ABSOLUTE:

  the inverse of DELTA; the value is stored as it is returned from the system (e.g. current number of free buffers)

- FLDDEF_NORMALIZED:

  the value was forced into a certain range; this is important for the calculation of sums: normalized values cannot be added to a sum (e.g. disk usage in percent - in this cases instead of sums the average value is calculated during analyses); this flag can be combined with ABSOLUTE

```
#define FLDDEF_ABSOLUTE       0x0001
#define FLDDEF_DELTA          0x0000
#define FLDDEF_NORMALIZED     0x0002


typedef struct FIELD_DEF {
   ULONG flFlags;                     // FLDDEF_* flags
   char *Name;                        // name of the field
} *PFIELD_DEF;
```

DEFAULT_GRAPH_DEF

This structure is used to define default views for a class. That enables the manager component to automatically create view and menu definitions for a new class or on a new machine. A single view can contain up to six graphs (= lines).

```
typedef struct DEFAULT_GRAPH_DEF {
   int   DataIndex;
   int   LensType;
} *PDEFAULT_GRAPH_DEF;


typedef struct DEFAULT_VIEW_DEF {
   char  ViewName[256];
   char  MenuEntry[256];
   ULONG MenuID;
   char  Key[4];
   int   GraphNum;                     // range is 1 to 6
   DEFAULT_GRAPH_DEF GraphDef[6];
} *PDEFAULT_VIEW_DEF;
```

CLASS_DEF

This class is the parent for all classes which build the interface between a general purpose engine (the monitor agent for monitor objects,. the manager for views and monitors) and the special purpose code to access a certain system or application.

Two class flags are important for a developer:

- CLASS_TYPE_GLOBAL:

  The class will return exactly one monitor object because it returns global information; there do not exists several resources for which this information could be retrieved.

- CLASS_TYPE_DYNAMIC:

  The resource(s) which is(are) monitored by monitor objects, returned by the class, may appear or disappear over time. Therefore the monitor agent must ask the class in regular intervals whether existing objects expired and new objects were discovered. If this flag is not set the monitor agent will ask for monitor objects once.

```
#define CLASS_TYPE_GLOBAL     0x0001
#define CLASS_TYPE_INIT       0x0002
#define CLASS_TYPE_DYNAMIC    0x0004
#define CLASS_TYPE_REMOTE     0x0008   // not used anymore

#define CLASS_FLAG_CREATEINST 0x1000
#define CLASS_FLAG_VIEWS_DEF  0x2000


typedef class CLASS_DEF {
   union {
      char *         RoleName;
      PMACHINE_ROLE  pRole;
   } Role;
protected:
   int            Entries;
   int            NextIndex;

public:
   char       *Key;
   char       *Name;
   FLAGS       flStatus;
   int         nFields;
   PFIELD_DEF  pFieldArray;

   CLASS_DEF (char *k, char *n, FLAGS fl, int nF, PFIELD_DEF pFA, char
*ServerRole);
           int        matchRole (char *Role);
           FIELD_DEF &getFieldDef (int Index) { return
pFieldArray[Index]; }
   virtual PRETRIEVER BeginEnumObjects (PMACHINE pServer)     { return
NULL; }
   virtual PRETRIEVER GetNextObject (PMACHINE pServer)        { return
NULL; }
   virtual void       EndEnumObjects (PMACHINE pServer)       { }
           int        getFieldIndex (char *Name) const;
   virtual int        getClassVersion (void) const            { return 0;
}
   virtual PMACHINE_ROLE getRole (void) const { return Role.pRole; }
   virtual char *     getRoleName (void) const { return Role.pRole ->
Name; }
   virtual int        getDefaultViewDefinition (int Index,
DEFAULT_VIEW_DEF &DefViewData) { return FALSE; }

           void       ResolveRoleName (void);
} *PCLASS_DEF;
```

MODULE_PRESENTATION

The class MODULE_PRESENTATION is the true OO interface between the monitor agent and the monitor module (in contrast to the technical interface of the C function, which is exported by the DLL, because this is the only way OS/2 allows a process to dynamically access code in a module). Each monitor module has to provide implementations of the virtual methods of the class.

```
typedef class MODULE_PRESENTATION {
protected:
   HMODULE hMod;
public:
           void setModuleHandle (HMODULE _hMod)
                { hMod = _hMod; }

   virtual void getClassDefinitions
                   (PMODULE_CLASS_DEF pModClassDef) = 0;
   virtual int  getModuleCopyright
                   (char *TextBuffer,
                    unsigned long BufferSize) const
                   { return FALSE; }
   virtual int  getRoleDefinitions
                   (PMACHINE_ROLE pMachineRole[])
                   { return 0; }
   virtual int  initMonitoring      (void) { return TRUE; }
   virtual int  getServerNames      (ServerDef Names[])
                   { return 0; }
} *PMODULE_PRESENTATION;
```

Method	Meaning
setModuleHandle	reserved; The method is used to set the module handle of the OS/2 DLL. The handle is not known by the module but by the code that loads the DLL (the monitor agent). After the monitor agent gets the instance of the class it sets the module handle which is available for all further processing.
getClassDefinitons	When this method is called it is supposed to return references to instances of the monitor factory objects that it provides. The monitor agent passes the pointer to an object of type MODULE_CLASS_DEF to the method. It inserts object pointers and the number of provided classes into that object.
getModuleCopyright	The method is called by agent and it provides a textbuffer to the method. It may put a copyright string into the buffer and this string is displayed on the console. It is recommended that version/release and publication year (date) are mentioned.
getRoleDefinitions	The method may return new role definitions which consists of a name and icons for the user interface. Classes of this and other modules may reference these roles by their name.
initMonitoring	The method is called by the monitor agent and it should be used to initialize monitoring if necessary. It returns TRUE on successful initialization or FALSE otherwise. In the later case the module is ignored and no class from it will be used.
getServerNames	The method is called by the monitor agent and it is used to register target servers of the monitor module. This method is only used when the monitor agent is in "remote monitoring" mode. Prerequisite is that the monitoring code is able to detect remote servers automatically.

Appendix E. Currently Supported Resources

The following tables contain a detailed summary of all extracted data items and the classes within they are organized. If the meaning of a data item is obvious from its name the column „**Meaning**“ does not contain an explanation.

Part III describes how the data can be used in the tasks of an system architect or an operator and what can be derived from a real networks.

IBM LAN Server

The following values are derived from the **IBM LAN Server** network operation system starting with version 3.0. Most of the information is available via published APIs. If another access method is used it is mentioned in the data description.

Class	Class Description	
	Data Item	**Meaning**
Connection	measures for each shared resource information about established connections. Each connection is represented by its *share name*; this is the name which is used to access a resource at the server. An exception to this rule are (private) connections to home directories[45]. Due to the possible great number of active users connections to home directories are summarized under one monitor object which is called *Home Dir.* and which is the representation of all active users with a home directory.	
	Connections	number of connection for the referenced resource
	Open Files	number of open files (on behalf of the connection) at the time a data sample is retrieved
Printer queue	provides information about shared printer queues which spool print jobs to network printers.	
	Jobs	total number of print jobs which are currently waiting in the queue (which includes the print job which is currently spooled to the printer)
	Total size of print jobs (kB)	size of all waiting print jobs in kB
	Print time of first job	the time the server needs to spool the first print job to the printer (this value is not provided by the server but is calculated by the module
User	information about currently logged on user	
	User	number of logged on users
Statistics	several statistic information which is collected by the LAN server itself.	
	new print jobs queued	the number of print jobs which were entered into a printer queue of the server since the last update

[45] A home directory is a shared file resource which (under normal circumstances) can only be accessed by a certain user who is the owner of the home directory. Home directories are assigned during logon automatically and cannot be accessed like a normal shared resource.

Class	Class Description	
	Data Item	**Meaning**
	new server sessions	number of sessions which were established since the last update
	autodisconnected sessions	number of sessions which were disconnected by the server (because they were not used for some amount of time) since the last update
	sessions errored out	number of sessions which disappeared due to an error
	password errors	number of wrong passwords used in explicit or implicit logons
	access permission errors	number of invalid access attempts to resources which are protected by LAN server security
	average response time	average response time to service requests in milliseconds
	file open requests	number of requests to open a file for all file resources since the last update
Data	information about the volume of data handled by the server; all data items are deltas (they are calculated as difference between the current value and the value of the most recent data update).	
	kB server sent	number of kB sent to the server[46]
	kB server received	number of kB received by a client from the server[46]
	kB redirector sent	number of kB sent to the redirector service[46]
	kB redirector received	number of kB received from the redirector service[46]
	kB application sent	number of kB sent to an application on the server
	kB application received	number of kB received from an application on the server
	buffers required	number of buffers required by the server but which were not available at that time[47]; these indicates a configuration problem and degrades performance.
	big buffers required	same as above but for a class of larger buffers ("big buffers")
	NCBs issued server	number of network control blocks (NCB) which were issued by the server service

[46] In the beginning it was unclear what the value of this item meant. From the descriptions it seemed to mean "sent **by** the server". By observations in comparison to known actions (e.g. file copy) it became obvious that the contrary is true. That is why the name of these data items are a bit confusing. For the sake of continuity of an already distributed program the names were not changed.
[47] The IBM LAN server software does not provide the number of buffers in use.

Class	Class Description	
	Data Item	**Meaning**
	NCBs failed server	number of network control blocks (NCB) which were issued by the server service but which returned an error code
	NCBs issued redirector	number of network control blocks (NCB) which were issued by the redirector service
	NCBs failed redirector	number of network control blocks (NCB) which were issued by the redirector service but which returned an error code
	NCBs issued application	number of network control blocks (NCB) which were issued by any application, which uses the services of the LAN server
	NCBs failed application	number of network control blocks (NCB) which were issued by any application, which uses the services of the LAN server, but which returned an error
Sessions	information about server sessions	
	Sessions	number of established sessions
	active time	sum of the time all sessions are active in minutes; a session is active from the moment it is established; its active time is increased as long the session does exist
	idle time	sum of the time all sessions are idle in minutes; a session is idle if no requests (or data traffic) are performed via the session; the idle time starts when the last action has finished and is increased until the action is performed. Increasing idle time points to a number of idle sessions. As any sessions - used or unused - consumes system resources it is desirable to remove unused (idle) sessions.

HPFS386

If the network file system HPFS386 (high performance file system 80386 optimization[48]) is used on the server, the monitor agent is able to query information about the effectiveness of the file system cache by use of the program CACHE386. This program does not provide an API but the monitor agent uses the information written to *stdout*.

Class	Class Description

[48] 80386 optimization means that it is the 32-bit version of the standard OS/2 file system HPFS, which is 16-bit code (for the 80286).

	Data Item	Meaning
CACHE386	information about the cache of the HPFS386 file system; with the exception of hit rates all values are deltas between the current and the most recent data update.	
	Read Requests (x 100)	number of read requests (the real value is divided by 100 in order to fit the result into the two byte data range)
	Disk Reads (x 100)	number of read requests which actually cause a disk read operation
	Cache Reads (x 100)	number of read requests which could be satisfied by the cache
	Cache Hit Rate (Reads)	cache efficiency for read requests in percent
	Write Requests (x 100)	number of write requests
	Disk Writes (x 100)	number of write requests which actually cause a disk write operation
	Cache Writes (x 100)	number of write requests which could be satisfied by the cache
	Cache Hit Rate (Writes)	cache efficiency for write requests in percent
	Hot Fixes	number of automatic disk error corrections and recovery actions

DB2/2 Version 1

In contrast to the later versions of DB2 the IBM DB2/2 Version 1.x family of DBMS did not support load measuring. Nevertheless, with the help of a set of configuration APIs it is possible to retrieve some information. But due to the following problems the support for DB2/2 V1.x will be dropped in the near future and it is not recommended to use this module:

1. There is not much relevant information available via these APIs

2. On the test system the monitor agent produces a memory access violation if a certain database application is active. The error occurs within a DB2 code DLL and it seems to be a problem of the old DB2 code.

3. On a production system unpredictable system halts occurred if the module is in use. I did not found a direct connection and the halts may have a different reason but if the module is not used less production interruptions were detected.

4. Code developed with the current developer's toolkit is incompatible with the older DB2 version; exactly the three APIs which are used in the module, are even documented to be incompatible:

   a. Code developed with the older V1 libraries trap under a V2; a version check was added and the module is deactivated under V2.

   b. Code developed with the newer V2. libraries cannot be loaded on a V1 system - some DLLs and function entries are missing.

   That makes development and code maintenance very cumbersome and errorprone.

At least for historic reasons I document what could be done under DB2/2 Version 1. Note that - though they are members of the same product families - the two versions of DB2/2 are totally different code and that two totally different agent modules had to be written to access load information.

Class	Class Description	
	Data Item	**Meaning**
DB2 Database	information about DB2/2 V 1.x	
	connected user	total number of users which have an active connection to any database on the system
	number of transactions	number of database transactions since the last update
	number of requests	number of SQL requests since the last update
	number of current requests	number of currently active SQL requests
	connection time (min)	total number of minutes for which active connections do exist
	transaction time (min)	total number of minutes for which current transactions are pending[49]

[49] This number turned out to be not very meaningful. The systems counts the time starting from the last COMMIT statement. Therefore, if an application commits a transaction and then pauses while keeping the connection open DB2 accounts this time for the next transaction.

DB2/2 Version 2

As mentioned above starting with version 2.0 DB2/2 offers an API which provides a great wealth of information about performance and load in a DB2 system. Below you find the list of data items which are used in the load monitor. Most item names make clear what they mean. If you are interested in more detail refer to the technical reference material.

Later you will read about the usefulness of this information and about connections and relations between these items and other information retrieved from an OS/2 system with DB2/2.

Class	Class Description	
	Data Item	**Meaning**
DB2 2.x Engine	information about the DBMS (the database engine)	
	Sort Heap Allocated	For detail information see the DB2/2 technical reference material.
	Post Threshold Sorts	-""-
	# of Piped Sort Requests	-""-
	# of Sort Reqs Accepted by SLS	-""-
	Remote Connects	-""-
	Remote Connects To Target Exec	-""-
	Current Local Connections	-""-
	Local Connects To Target Exec	-""-
	Local DBs with Connects	-""-
	# of Registered Agents	-""-
	# of Agents Waiting on Token	-""-
	# of Unassigned Agents	-""-
	KB Committed Private Memory	-""-
DB2 2.x Database	information's about a particular database	
	# of locks held	-""-
	# of lock waits	-""-
	wait time on locks	-""-
	total lock list memory in use (KB)	-""-
	# of deadlocks	-""-
	# of lock escales	-""-
	# of X-lock escales	-""-
	# of lock time-outs	-""-
	# of applications waiting on locks	-""-

Class	Class Description	
	Data Item	**Meaning**
	sort heap allocated (KB)	-""-
	# of sorts	-""-
	time spent in sorts	-""-
	# of sort overflows	-""-
	# of active sorts	-""-
	# of pool data logical reads	-""-
	# of pool data reads	-""-
	# of pool data writes	-""-
	# of pool logical index reads	-""-
	# of pool index reads	-""-
	# of pool index writes	-""-
	buffer pool read time	-""-
	buffer pool write time	-""-
	# of files closed	-""-
	# of asynchronous pool data reads	-""-
	# of asynchronous pool data read requests	-""-
	# of asynchronous pool data writes	-""-
	# of asynchronous pool index writes	-""-
	async read time	-""-
	async write time	-""-
	LSN Gap cleaner triggers	-""-
	dirty page steal cleaner triggers	-""-
	dirty list threshold cleaner triggers	-""-
	direct reads	-""-
	directs writes	-""-
	direct read requests	-""-
	direct write requests	-""-
	direct read time	-""-
	direct write time	-""-
	# of commit SQL statements	-""-
	# of rollback SQL statements	-""-
	# of dynamic SQL	-""-

Class	Class Description	
	Data Item	**Meaning**
	statements	
	# of static SQL statements	-""-
	# of failed SQL statements	-""-
	# of SQL select statements	-""-
	# of DDL statements	-""-
	# of update/insert/delete statements	-""-
	# of auto rebinds	-""-
	# of internal row deletes	-""-
	# of internal row updates	-""-
	# of internal row inserts	-""-
	# of internal commits	-""-
	# of internal rollbacks	-""-
	# of rollbacks due to deadlocks	-""-
	# of row deletions	-""-
	# of row inserts	-""-
	# of row updates	-""-
	# of row selections	-""-
	# of binds/recompiles	-""-
	# of new connections	-""-
	# of connected applications	-""-
	# of applications executing in DB2	-""-
	# of secondary logs allocated	-""-
	# of log reads	-""-
	# of log writes	-""-
	package cache lookups	-""-
	package cache inserts	-""-
	catalog cache lookups	-""-
	catalog cache inserts	-""-
	catalog cache overflows	-""-
	catalog cache heap full	-""-

Local Area Network

This module contains a class for accessing NETBIOS information and another for direct query of adapter data. However, the second class turned out to fail to deliver useful (dynamic) information. Therefore it was dropped and is not mentioned in this context.

For more information about the NETBIOS stack and the meaning of the items see [2].

Class	Class Description	
	Data Item	**Meaning**
NETBIOS	Information about the NETBIOS stack on each logical network adapter[50]. If not stated otherwise all items below are deltas between data updates.	
	Frames Received	number of NETBIOS frames received
	Frames Sent	number of NETBIOS frames sent by the station
	Bad IFrames Received	
	Aborted Transmissions	
	Packets Transmitted	number of packets which were generated by the local computer; the item is an indicator for network activities on the machine
	Packets Received	number of packets received by the machine; that does not mean that all the packets were addressed to it and processed on the local machine; the item is an indicator for network traffic in general within the physical LAN segment
	Bad IFrames Transmitted	
	Lost Data	
	T1 Expiration	number of times the T1 timer expired[51]
	Ti Expiration	number of times the Ti timer expired[52]
	Free NCBs	absolute number of network control blocks
	Busy Condition	
	pending sessions	absolute number of pending sessions
	Names Present	absolute number of known (= registered by a local application) NETBIOS names on the adapter

[50] For each physical network adapter one or more logical adapter can be defined.
[51] The T1 timer is set when a LPDU frame is sent and measures the time until an answer to that frame is received. If the timer expires another frame is sent in order to retry to establish the required connection.
[52] The Ti timer (inactivity timer) is set after an activity on an established connection. If the timer expires before any other activity the network software checks whether the connection is still up.

OS/2

The following classes are based on standard OS/2 APIs and provide some information about the base system itself. The class *directory tree* compiles information about the last time an data file was accessed. My assumption is that - if collected over several months - I can gain valuable observations about the aging of information on a server and about the amount of "dead" information which may be archived and removed from the disks.

Class	Class Description	
	Data Item	**Meaning**
Logical Disk	Information about logical disks (partitions of local fixed disks)	
	MB available	available disk space in Mbytes
	% full	percentage of total disk space which is used
Directory Tree	Information about the directory tree of a logical disk; contrary to logical disks this class can be configured by the operator: if (s)he provides the file SRVMDRTR.DEF the class reads from that file what will be monitored. It is a text file and it must contain a number of lines that consist of target path, update interval and monitor name (for display and logging). By default the entire directory tree of each local disk partition is considered and data are taken once an hour[53].	
	# of files (x 10)	total number of files in whole directory tree (divided by ten)
	# of directories	total number of directories
	MB allocated	total disk space allocated by files and directories in Mbytes
	KB wasted	delta between allocated and used disk space as reported by the file system in Kbytes; this number depends on disk segmentation and file sizes
	# of files accessed within 1 day	total number of files which were accessed during the current day; calculations are based on dates reported by the file system; not every file system can handle access dates (the DOS file system FAT, which is supported by OS/2, does not distinguish between access and creation date - but nobody uses FAT on a file server);
	# of files accessed within 1-3 days	
	# of files accessed within 4-10 days	
	# of files accessed within 11-30 days	
	# of files accessed within 31-100 days	
	# of files accessed not within 100 days	total number of files which were not accessed during the last 100 days

[53] The update interval for the class is independent from the sampling interval of the monitor agent.

Class	Class Description	
	Data Item	**Meaning**
	MB allocated accessed within 1 day	total amount of disk space in Mbytes which is allocated in files which were accessed during the current day
	MB allocated accessed within 1-3 days	
	MB allocated accessed within 4-10 days	
	MB allocated accessed within 11-30 days	
	MB allocated accessed within 31-100 days	
	MB allocated accessed not within 100 days	
System	General information about the OS/2 system	
	MB memory available	total amount of virtual memory available to all processes of the system; this value depends on available disk space on the partition where the swapper resides
	% CPU used	percentage of the processor time interval - defined by the system variable MAX_WAIT[54] - which is not available to an "idle class" (background) thread/process; the rest of the processor time is spent for other activities on the machine; the default for MAX_WAIT is 3000 ms
	MB swapper	size of the swapper file in Mbytes

[54] The MAX_WAIT value defines the amount of time a thread in the normal priority class can be ready for execution and suspended, until it receives a *starvation boost*. That means that its priority is raised and it becomes more likely that it can be scheduled.

SRVMSRVR

The monitor application provides the ability of monitoring its own workload. This is an example of how a "normal" business application" can be enabled for monitoring.

Class	Class Description	
	Data Item	**Meaning**
SRVMSRVR	Information about the collection server component of the monitoring tool	
	Available Threads	number of service threads which are currently active to handle requests to the server (threads are dynamically started and ended depending on the server usage
	Used Threads	number of threads which were active during the last sample interval
	Managed Machines	number of machine (= agent) definitions which are known to the server
	Managed Objects	number of monitor object definitions, which are known and handled by the server
	Messages received	number of messages received and processed during the last sample interval
	Messages sent	number of messages sent to a partner during the last sample interval
	Connections established	number of connections which where established by agents (or other servers) during the last sample interval
	number of kB received	number of kB of data received by the server during the last sample interval
	number of kB sent	number of kB of data sent by the server during the last sample interval

The technique used by the server is the use of a DLL with a global, shared data segment. The only difference to a normal DLL is that you have to provide two different keywords in the module definition file for linkage. The effect is that all processes that link to this DLL share the data segment of the DLL. In other words, there will be only one instance of all the global variables of the source module and they are shared and accessible for all processes. That way it is very simple to provide load information by an application:

Create a DLL with, at least, two entry points: one to set or increment load data counters and one to query or reset them. The use of *running counter variables* is very simple and appropriate in most cases. The counter have to be global within the source module of the DLL. A new monitor module[55] can link to the DLL and use the query function to access the load data of the business application.

TCP/IP

The monitoring agent uses the output of the TCP/IP utility program ***netstat*** and converts the information into the format for SRVMONPM. As netstat reports more than 200 items I do not present them as a table like the other classes.

[55] It would be possible to merge the interface DLL and monitor module into one physical DLL but that is not recommended. Monitor modules which have to confirm to certain standards should not be mixed up with parts of other applications.

A new problem was introduced with *netstat* which needed a different design for data access by the monitor agent:

1. *netstat* pauses while and after displaying information at the console and expects console input. It does not use *stdin* but obviously uses console i/o functions to handle this. To be able to use the program in the background it must be started as an independent and detached process.

2. *netstat* may or may not provide output. Therefore waiting for output may never yield any result. As *netstat* has to be an independent and detached process the calling process has no control over it and cannot tell whether netstat is still active or not. There is no direct mean to detect whether some output was generated or not.

This lead to a new, more robust design of the monitor agent.

Appendix F. Message Structure of the Transport Layer

First, we will look at the definition of the messages which are sent across the network. All definitions contained in this chapter are defined in the source file **SRVMPROT.HPP**. The messages are organized in a class hierarchy. The receiver of a message identifies the message by the message id. The first two bytes of the message header contain this id. The id consist of a number and a version. This is necessary to support different (newer) versions of infrastructure components during upgrades. Following messages are available:

```
#define MSG_TYPE_OBJ_DEF            0x1001
#define MSG_TYPE_DATA               0x0002
#define MSG_TYPE_ERROR              0x0003
#define MSG_TYPE_CLIENTDATA         0x0004
#define MSG_TYPE_REGISTER_OBJ       0x0005
#define MSG_TYPE_REGISTER_MACHINE   0x1006
#define MSG_TYPE_ACKNOWLEDGE        0x0007
#define MSG_TYPE_QUERYINFO          0x0008
#define MSG_TYPE_QUERYMANAGER       0x0009
#define MSG_TYPE_QUERYREMARK        0x100A
#define MSG_TYPE_QUERYUPDATEINT     0x000B
```

Not every type has a corresponding message class. Some types are used with the same structure.

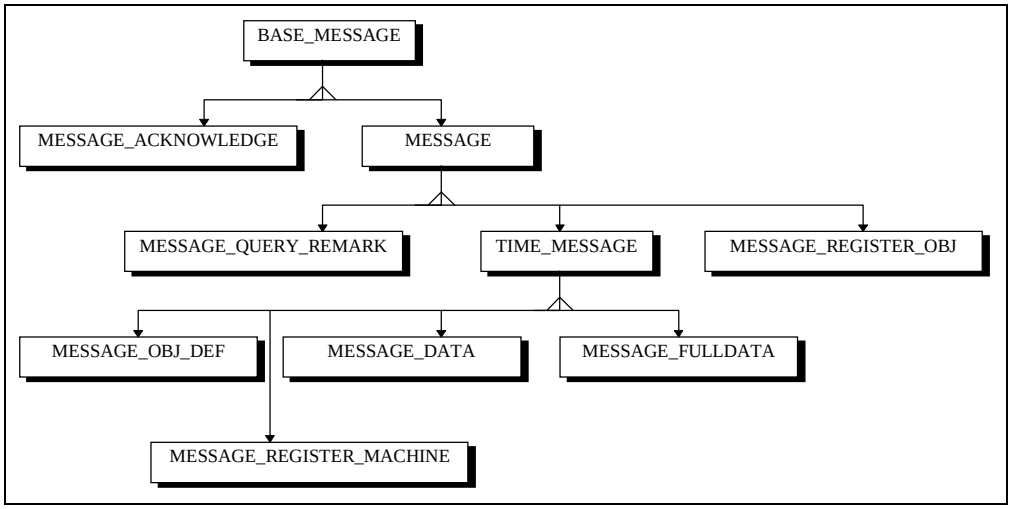


Figure 50. Message class hierarchy

BASE_MESSAGE

The root class in the hierarchy of messages is the class **BASE_MESSAGE**:

```
typedef struct BASE_MESSAGE
{
   unsigned short Type;

   BASE_MESSAGE (unsigned long _Type):  Type (_Type) { }
} *PBASE_MESSAGE;
```

The message contains the message type only. Each message can be accessed as BASE_MESSAGE and the receiver of the message has to cast the reference to the message according to the message id.

MESSAGE_ACKNOWLEDGE

Directly derived from BASE_MESSAGE is the class **MESSAGE_ACKNOWLEDGE**. It is used for all acknowledgments (responses to messages). This class contains two unsigned shorts; one for an result code for the requested operation and one for an extra argument in addition to the code.

```
typedef struct MESSAGE_ACKNOWLEDGE: public BASE_MESSAGE
{
   unsigned short ResultCode;
   unsigned short Argument;

   MESSAGE_ACKNOWLEDGE (unsigned short _ResultCode):
      BASE_MESSAGE (MSG_TYPE_ACKNOWLEDGE),
      ResultCode (_ResultCode) { }
} *PMESSAGE_ACKNOWLEDGE;
```

MESSAGE

The second child of BASE_MESSAGE is the class **MESSAGE**. This class is the base for most other message classes. It contains either the name of a machine definition or the *machine request*. Messages that are based on this class, reference a machine definition.

```
typedef struct MESSAGE: public BASE_MESSAGE
{
   union {
      char           Machine[MACHINE_NAME_SIZE];
      int            IBD_Index;
   } MachineRef;

   MESSAGE (unsigned long _Type):
         BASE_MESSAGE (_Type) { }
} *PMESSAGE;
```

MESSAGE_QUERY_REMARK

Based on MESSAGE, the class **MESSAGE_QUERY_REMARK** is used to get detail information about a machine. At the moment this information consists of a remark and the name of the default domain.

```
typedef struct MESSAGE_QUERY_REMARK: public MESSAGE
{
   char           Domain[MACHINE_NAME_SIZE];
   char           Remark[REMARK_SIZE];

   MESSAGE_QUERY_REMARK (void):
      MESSAGE (MSG_TYPE_QUERYREMARK) {}
} *PMESSAGE_QUERY_REMARK;
```

MESSAGE_REGISTER_OBJ

The class **MESSAGE_REGISTER_OBJ** is used for the registration of monitor object definitions at a server and for sending the information from a server to a manager. The message contains the name of the sending station, the class key, name and version of the

object and the offset into the block of data items, where the load data of the object can be found (= OBJECT_DEF).

```
typedef struct MESSAGE_REGISTER_OBJ: public MESSAGE
{
   char     Sender[MACHINE_NAME_SIZE];
   char     Class[CLASS_NAME_SIZE];
   char     Name[OBJ_NAME_SIZE];
   int      Version;
   int      Index;

   MESSAGE_REGISTER_OBJ (void):
      MESSAGE (MSG_TYPE_REGISTER_OBJ) { }
} *PMESSAGE_REGISTER_OBJ;
```

TIME_MESSAGE

The class **TIME_MESSAGE** is based on MESSAGE and is used for all messages where a time stamp is moved over the network.

```
typedef struct TIME_MESSAGE: public MESSAGE
{
   unsigned long  TimeStamp;

   TIME_MESSAGE (unsigned long _Type, unsigned long _TimeStamp):
      MESSAGE (_Type), TimeStamp (_TimeStamp) { }
} *PTIME_MESSAGE;
```

MESSAGE_OBJ_DEF

The class **MESSAGE_OBJ_DEF** is based on TIME_MESSAGE and is used to ask for the next monitor object definition (referenced by a running counter - the object index). The time stamp is the time of the last update of the object definition and it is used to check at the manager whether the object definition has changed.

```
typedef struct MESSAGE_OBJ_DEF: public TIME_MESSAGE
{
   unsigned short ObjIndex;

   MESSAGE_OBJ_DEF (void):
      TIME_MESSAGE (MSG_TYPE_OBJ_DEF, 0) { }
} *PMESSAGE_OBJ_DEF;
```

MESSAGE_DATA

The class **MESSAGE_DATA** is used to send load data

- from an agent or intermediate managing server to another server or
- from the server to a manager.

The time stamp is used to check whether the data are valid. The message contains the name of the sending station, the number of data items in the message, the update interval, on which delta information is based, and the actual data. The class definition contains one data item but during runtime the necessary amount of data is allocated and attached to the message.

```
typedef struct MESSAGE_DATA: public TIME_MESSAGE
{
   char     Sender[MACHINE_NAME_SIZE];
   unsigned short DataNum;
   unsigned short UpdateInterval;
   DATAITEM Data[1];

   MESSAGE_DATA (void): TIME_MESSAGE (MSG_TYPE_DATA, 0) { }
} *PMESSAGE_DATA;
```

MESSAGE_FULLDATA

MESSAGE_FULLDATA is the same as MESSAGE_DATA but the class definition contains the maximal possible number of data items in the message (32.000). This definition is used during preparation of a message of type MESSAGE_DATA. Because of the size of this message it must be allocated dynamically and is not placed on the stack of the thread as it is done with the other (much smaller) messages. There is no extra type for this message; instead MSG_TYPE_DATA is used as the message identifier.

```
typedef struct MESSAGE_FULLDATA: public TIME_MESSAGE
{
   char     Sender[MACHINE_NAME_SIZE];
   unsigned short DataNum;
   unsigned short UpdateInterval;
   DATAITEM Data[MAX_MESSAGE_BUFFER_SIZE];

   MESSAGE_FULLDATA (void): TIME_MESSAGE (MSG_TYPE_DATA, 0) { }
} *PMESSAGE_FULLDATA;
```

MESSAGE_REGISTER_MACHINE

The class **MESSAGE_REGISTER_MACHINE** is used by the monitor agent or by the server to register a machine definition at a server. The message contains

- the name of the sending station,
- the default domain of the machine,
- the machine remark,
- the number of data items which will be send for that machine,
- the number of objects which will be registered for the machine,
- the update interval.

```
typedef struct MESSAGE_REGISTER_MACHINE: public TIME_MESSAGE
{
   char           Sender[MACHINE_NAME_SIZE];
   char           Domain[MACHINE_NAME_SIZE];
   char           Remark[REMARK_SIZE];
   short          DataIndex;
   short          SupportedObjects;
   unsigned short UpdateInterval;

   MESSAGE_REGISTER_MACHINE (void):
      TIME_MESSAGE (MSG_TYPE_REGISTER_MACHINE, 0) { }
} *PMESSAGE_REGISTER_MACHINE;
```

Appendix G. GUI Components of the Workload Manager

The following paragraphs describe the parts of the application the user interacts with. These windows make up the visible part of the application.

Single View

In the context of the graphical user interface the view of a monitor is a window which displays the contents of a view (as described in "**Internal Data Management**"). It consists of the following subparts:

- in the center lies the graphing area where the load data are plotted and a grid according to the legend is drawn
- below the graphing area a scrollbar shows, which section of the available (span of time of) load data is displayed (see below); it can be used to change the location of the visible area
- at left the vertical legend displays the scale of the values in the graph
- at the top the horizontal legend is drawn where the time and date is displayed when the data where captured
- at the bottom a short description for each plotted line is displayed; the description contains the name of the lens which is used to display the data
- at right an indicating instrument shows the most current values both graphically and in textual form - independent from the current scrolling

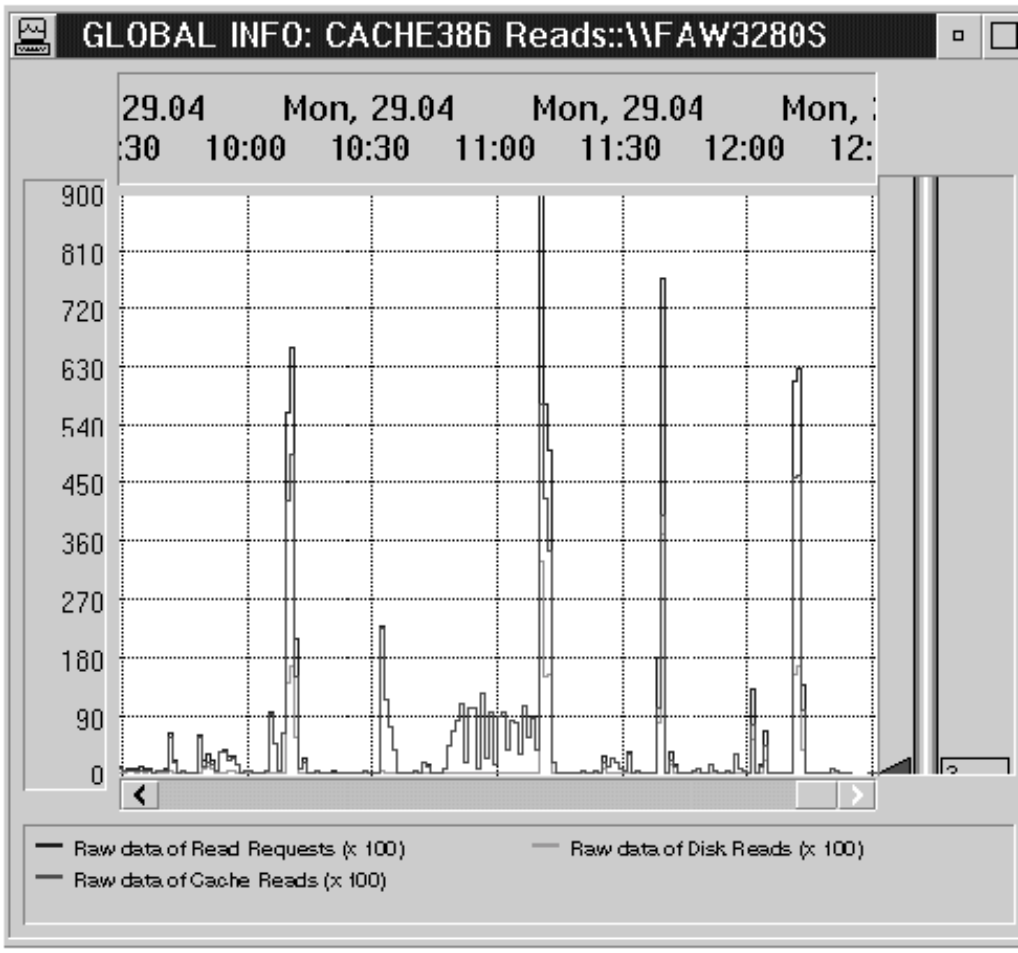


Figure 51. A single monitor view

Normally, the vertical scale is adjusted by the application automatically. Whenever a new value exceeds the current upper or lower margin the value is rounded up (or down) to the next multiple of ten and is used as the new upper (or lower) boundary. The graph

is adjusted accordingly. In addition the user can set the lower and the upper boundaries of each view. The user may disable automatic boundary adjustment to keep his selection in place.

The window displays only a section of all available data, because the computer screen is not large enough to draw all data. While the scrollbar represents the set of all available data the *bubble* of the scrollbar represents the actual visible part[56].

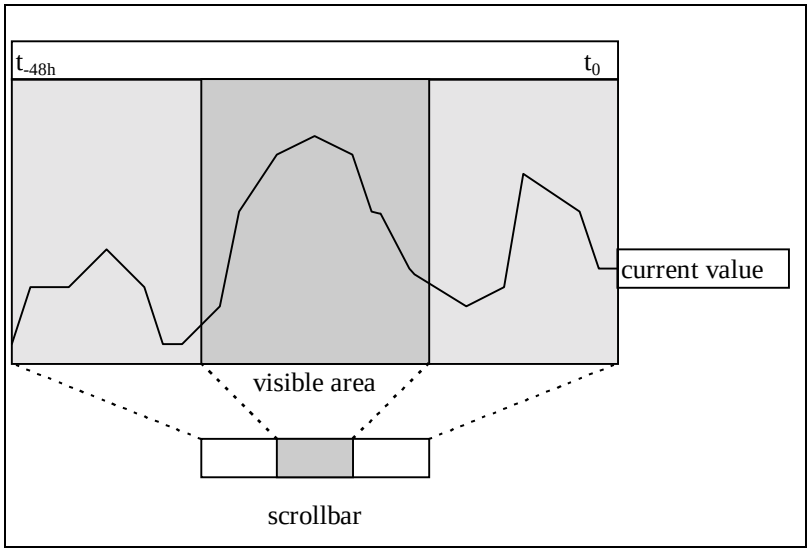


Figure 52. Relation between data area and view

If the single view is framed by its own window (like in the figure above) the titlebar of that window contains a combination from the name of the underlying view definition and the machine for which the view is drawn.

A view definition defines which data items are displayed within a view and how the data are represented (see below).

View Definition

The application manages a number of view definitions. They define which deducted monitors (= views) are created, what is part of a view and how the menu structure is build up. Each monitor module should provide default view definitions for the monitor factory objects that it exports to the application. The user may add, remove or alter the definitions.

[56] In contrast to applications in the Windows environment this is the recommended behavior or OS/2 applications.

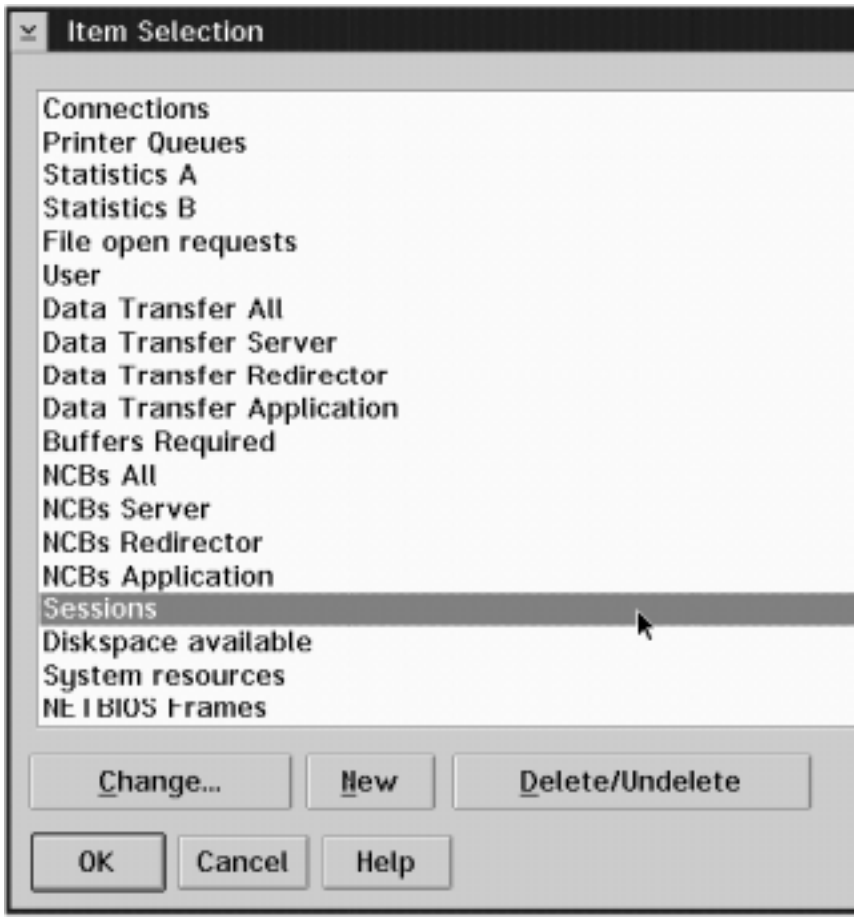


Figure 53. The view definition list

If the user wants to work with view definitions, first, he is presented with the list of available view definitions. He may choose one and select one of the provided push buttons. Note that, if one definition is removed, it is still in use by the application. Therefore it is not removed from the list immediately. Instead it is marked as "to be deleted". It will be removed from the configuration file. While it is in the list the user may undo his deletion and unmark the entry.

Any changes to the view definition list or any individual definition become effective with the next invocation of the application. Until then the old definitions stay active. A message box notices the user that he must restart the application to make use of his changes.

The **view definition** contains the following information:

- the name of the definition as it is used in the window title of the view
- the number of graphs (lines) which are drawn in the view
- the associated monitor factory object; this data item is chosen when the definition is initially created; it cannot be changed at a later time
- the menu entry under which views can be activated and displayed on the screen; by using slash ("/") or backslash ("\") characters a structure of cascading menus can be defined.
- up to six graph definitions; for each graph definition the user defines
  - ✓ the class item to be displayed
  - ✓ the lens which should be used to display the data
  - ✓ the color of the graph; the user may choose one of the listed colors or the entry *default* which lets the system select a color.

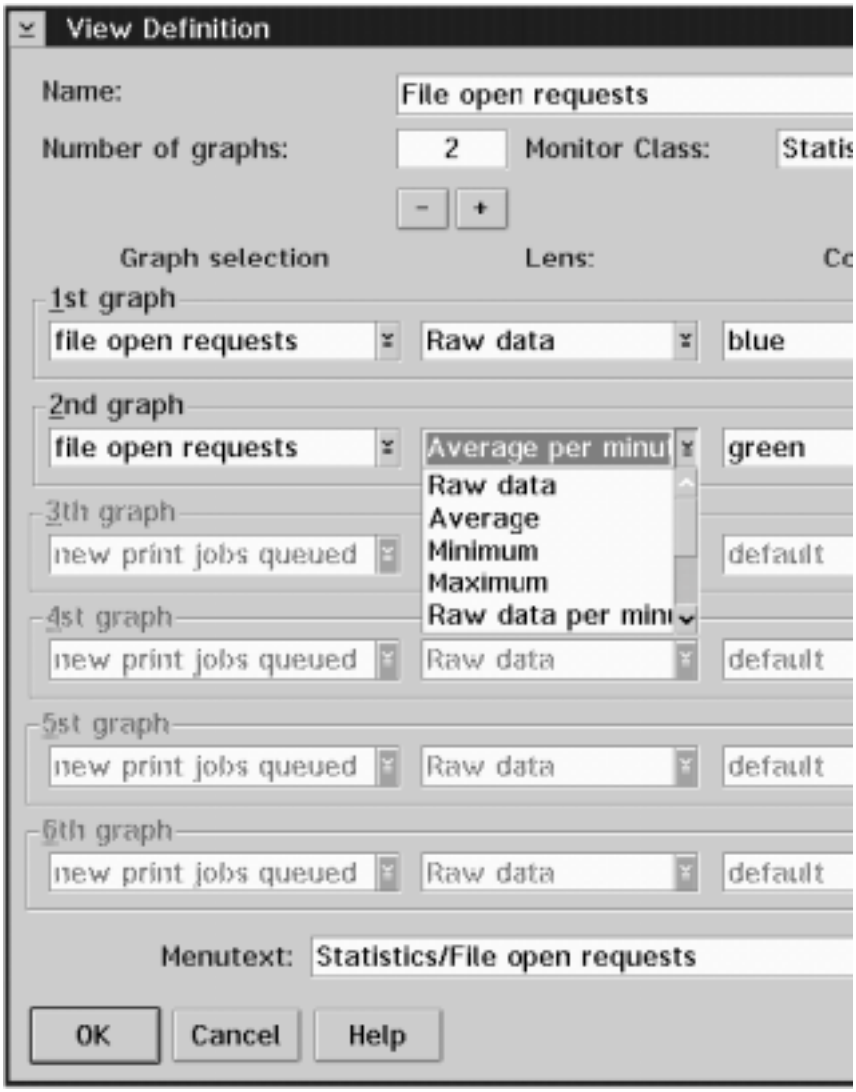


Figure 54. A monitor view definition

All definitions for graphs which are not used in the view are "disabled". The items in the graph definition are pull-down lists. Therefore the user can easily see and select available choices.

View Window

A view window represents a collection of monitors. The window of a view consists of a notebook control. Each page contains a single view of an individual monitor. The first page, which is labeled "Total", contains the sum (or an average) of the values of all notebook pages.

The average is calculated for *absolute* or *normalized* values like e.g. "response time in ms". It would not make sense to build the sum over all response times. All other (normal) values are added up. The class definition defines whether an item is *absolute* or not.

The *totals page* gives a very good overview over the activities on the monitored systems. The operator quickly gets an idea of what is going on in the group of monitored resources.

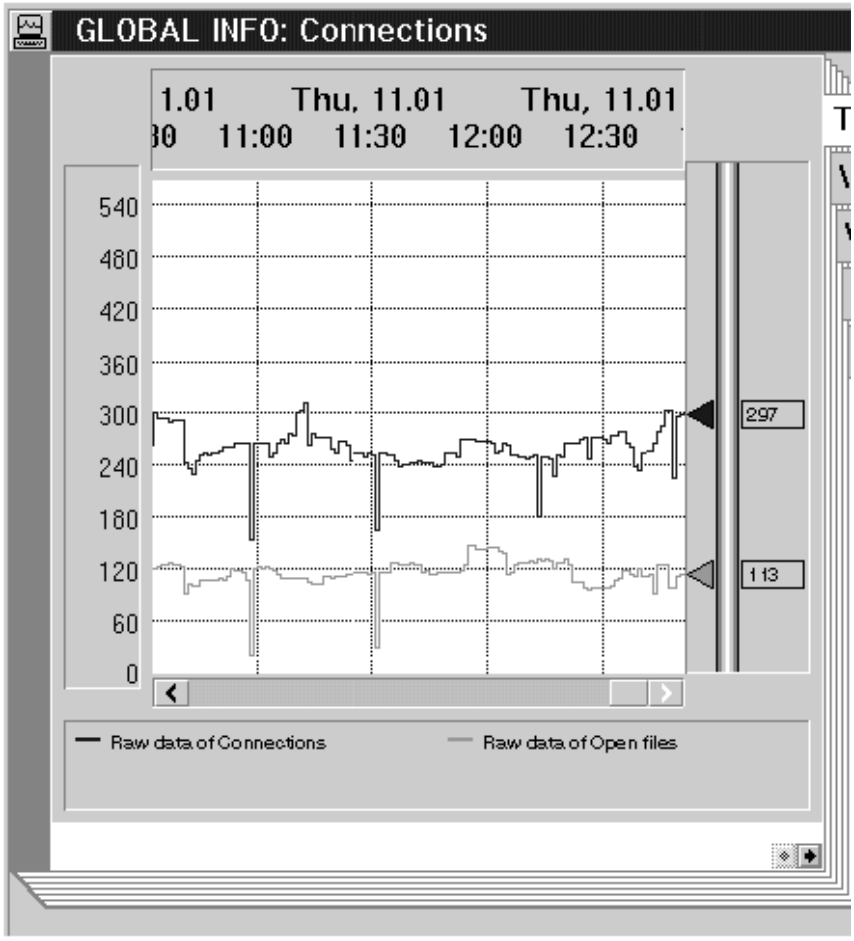


Figure 55. A monitor view window

Monitor Selection

As described above all views are ordered by machines and roles. This becomes visible in the monitor selection window. This window is the **main window** of the application which is displayed first after the application is started. It is a container window (often called a folder) which contains a number of icons.

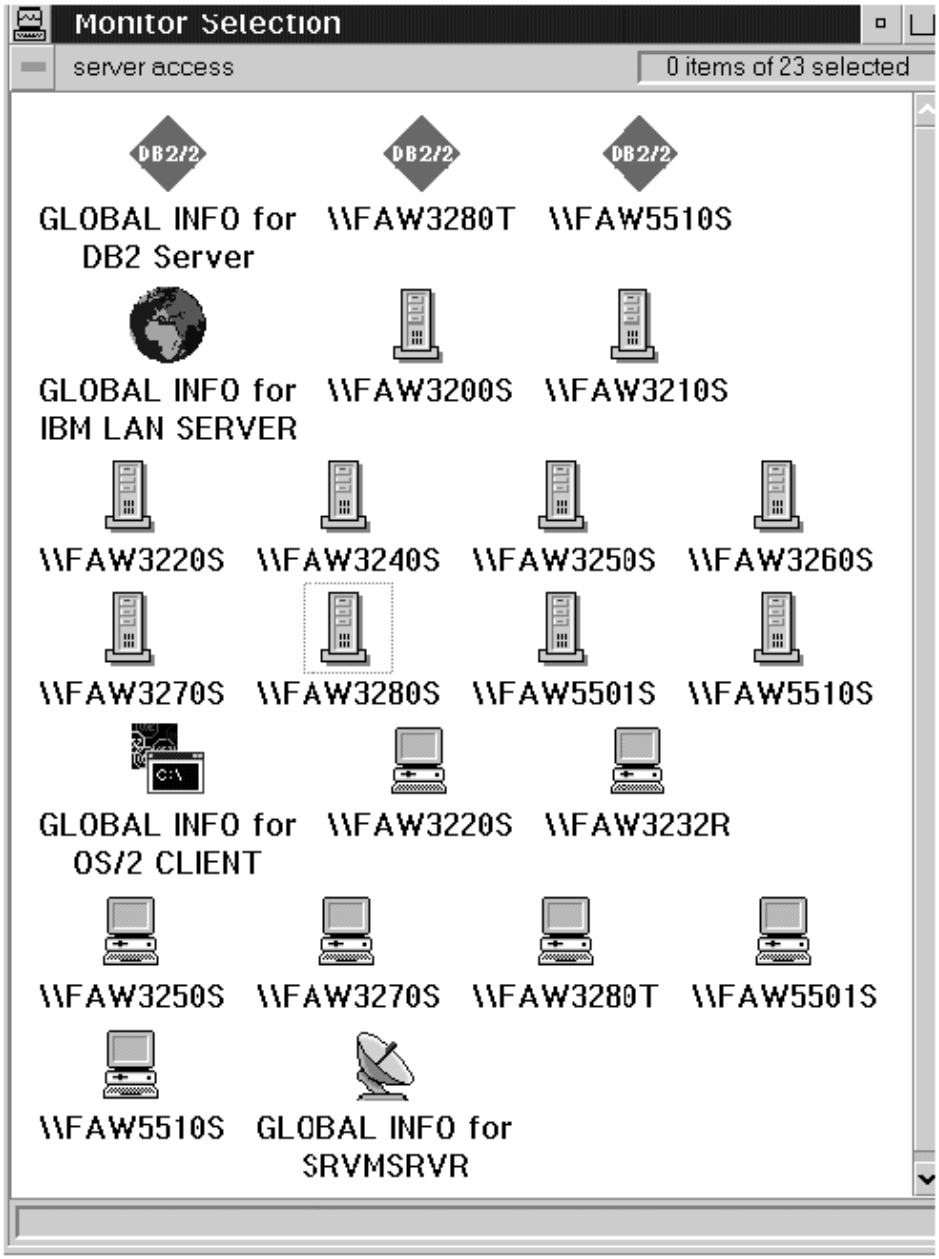


Figure 56. Monitor selection window (application main window)

Each icon represents a number of views which are accessible via the menu that is associated with the icon. Each role defines two icons: one for global views and the other for *multiple* views - these are views on resources which may exist in several instances on a single machine. Global views are represented by one instance of the global icon (e.g. the "globe" for global LAN server information). Because any number of (including no) instances of multiple views may exists for each machine, zero or more instances of the second icon may be visible - for each detected machine one instance is displayed (e.g. the "server icon" in Figure 56).

Each machine that owns monitors for different roles, may be represented by several icons (e.g. machine \\FAW5510S is a database server and a LAN server and an OS/2 client. Therefore it appears three times (with different icons) in the window above.)

In addition to global views the global icon contains references to the total pages of the multiple views. That offers the operator several levels of detail about resources:

1. the sum over the whole system

2. the sum over the resources on machine level

1. details about each resource

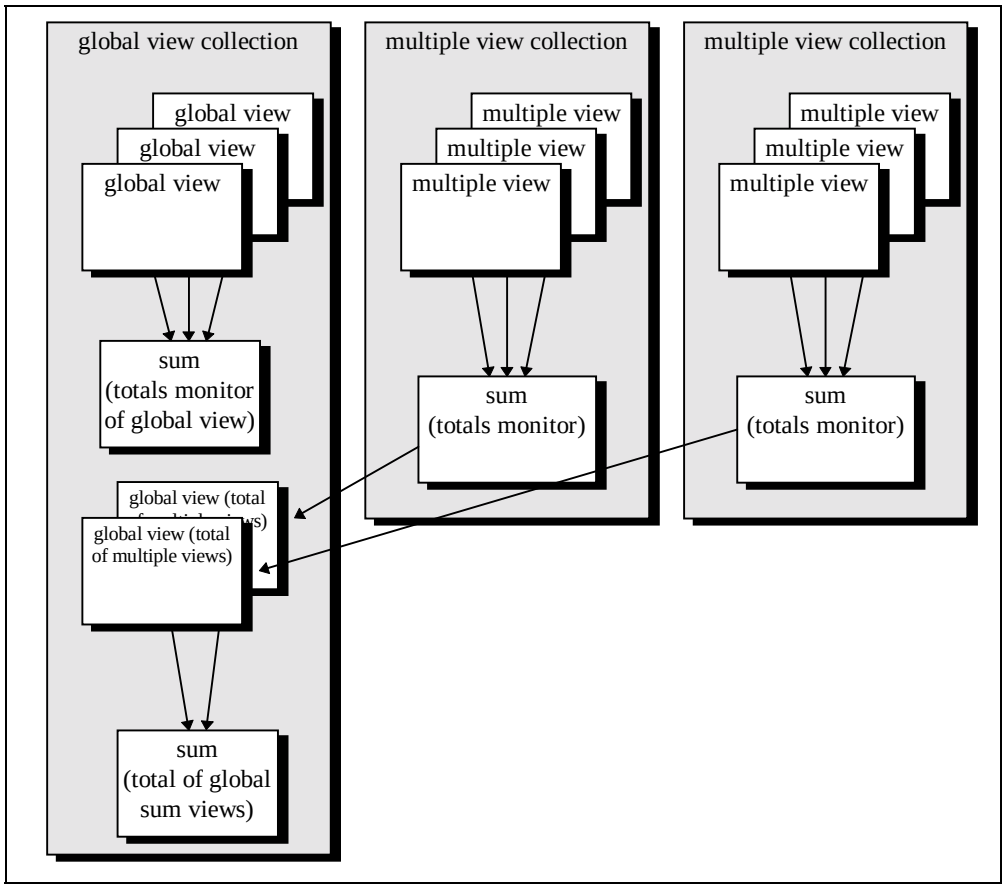


Figure 57. Multiple views, sum views and global views

Menus

Menus are the means to access views. Each role icon has its own menu which is generated depending on the active view definitions. As mentioned above the user may change the contents and structure of each menu. By clicking the right[57] mouse button over an icon the menu appears and the user may select a view window. A menu-item may be disabled by the monitoring application if no views do exist for a given definition.

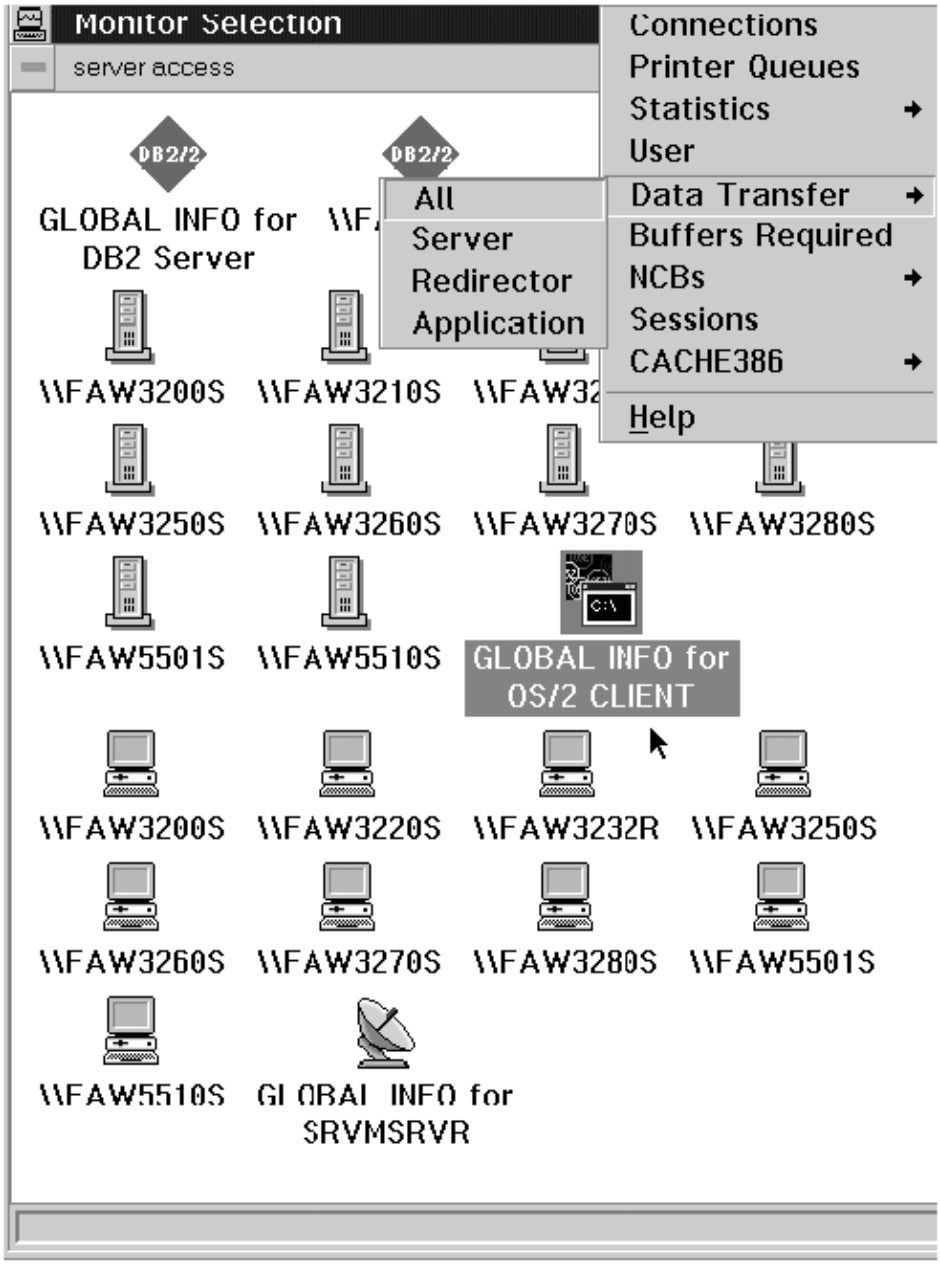


Figure 58. Main window with selected popup menu

The dynamic and open definition of menus and ways of accessing the data is very important for the seamless integration of new monitor modules into a single consistent user interface.

[57] This is true for OS/2's default settings. Left handed people may change the settings which would make the left button the one which brings up the menus.

Alert Definition

Similar to view definitions the user can work with alert definitions (see chapter 6.4. "**Alerts and Threshold Values**" for more information on alerts). The first window, when working with alert definitions, is the "Item Selection" Window. It contains the list of available alert definitions. From here the user can select alert definitions for editing.

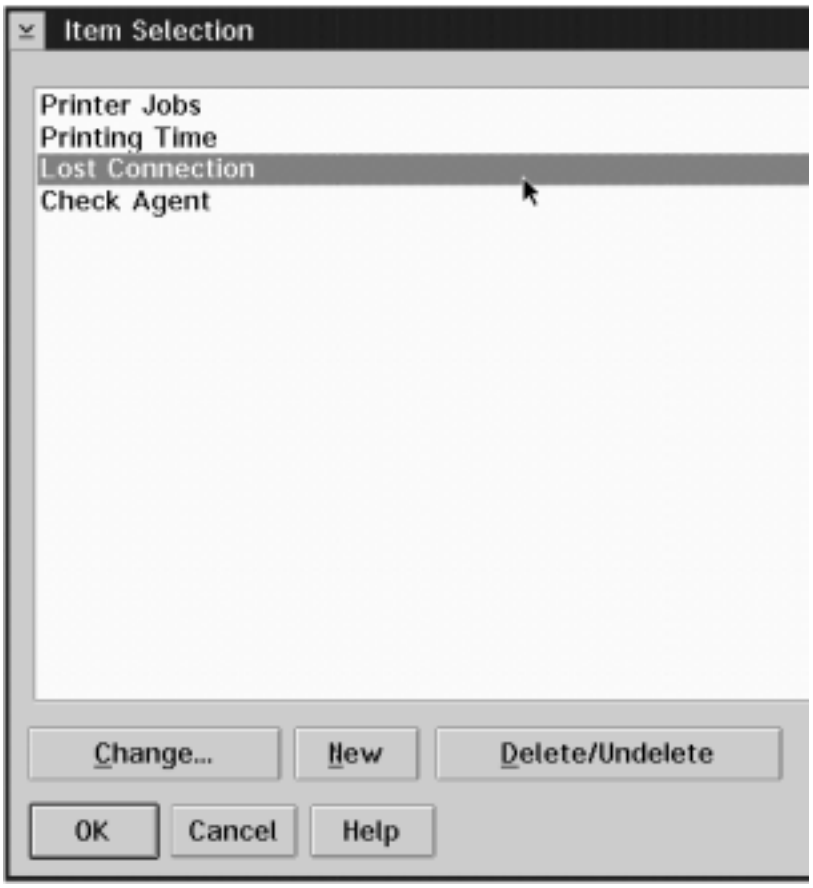


Figure 59. Alert selection list

Alert definitions are edited within a notebook which contains a number of alert specific pages. Starting from the standard pages of ALERT_DEF each subclass can add its own pages to the notebook. That way new alert classes can easily be integrated into the user interface.

Settings for ALERT_DEF

The base class of all alerts provides two pages:

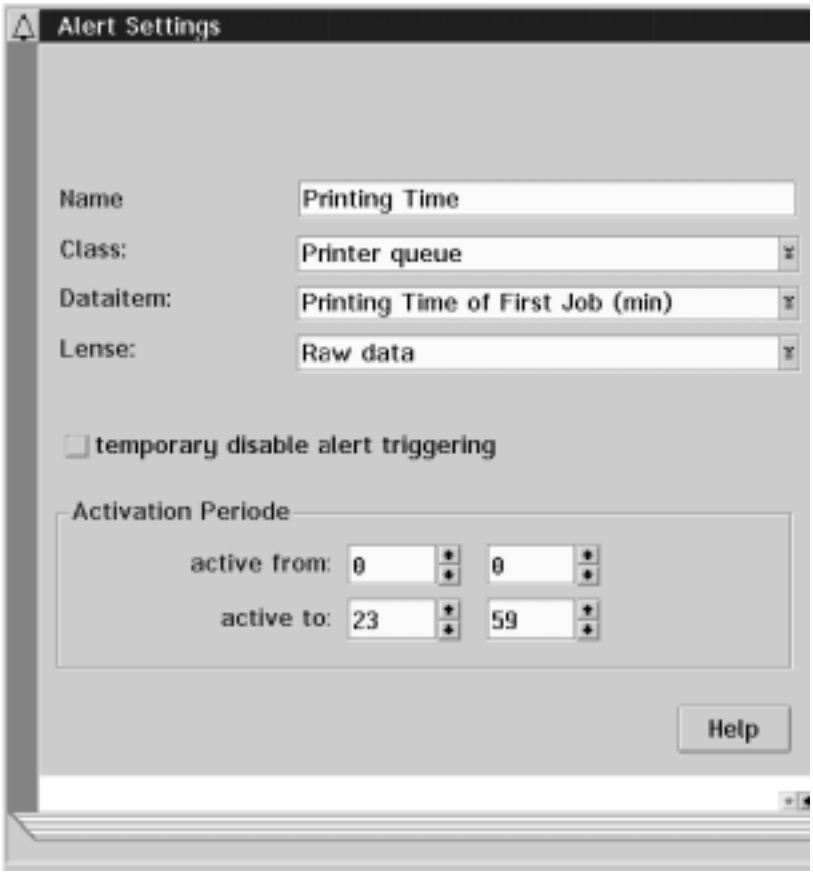


Figure 60. Alert definition page 1

The first page is used to define:

1. the name of the alert definition
2. the monitor factory object for which the alert definition will be used
3. within the class the data item which is to be checked
4. the lens that is used to preprocess the data before they are checked (this is done in analogy to the view windows)
5. whether the definition should be already used or not
6. the time of day during which the alert definition is to be used

These settings will be necessary for any kind of current or future alert classes.

Figure 61. Alert definition page 2

The second page is used to define:

1. The server for which the definition will be used; normally alert objects for all servers will be generated. The user may provide a part of a server name. All server will match which start with the given character. The wildcard characters '*' and '?' may be used for matching certain server names.

2. The resource for which the definition will be used; normally alert objects for all resources will be generated. The user may provide a part of a resource name. All resources will match that start with the given character. The wildcard characters '*' and '?' may be used for matching certain server names.

3. The condition on which the alert object will react.

4. In the case of threshold value checking, the definition of upper and lower limits. For each limit the user must provide a second value which marks the end of the alert condition. This is necessary to reflect the fact of possible hysteresis.

   If no other value is provided the same limit is used for begin and end of an alert condition.

This page is part of the general settings of the root alert class. In the case that new alert classes are introduced which implement new *conditions* the layout of the page and the underlying class implementation will be changed slightly.

The general class ALERT_DEF implements the conditions of an alert (**trigger condition**) but no action which should be taken in the case of the alert. This is delegated to specializations of the class (or subclasses of ALERT_DEF).

Additional Settings For ALERT_DEF_EXEC

One (and at the moment the only) specialization of the class ALERT_DEF is ALERT_DEF_EXEC. Its primary target is the definition and execution of user definable commands which will be fired when an alert condition is triggered.

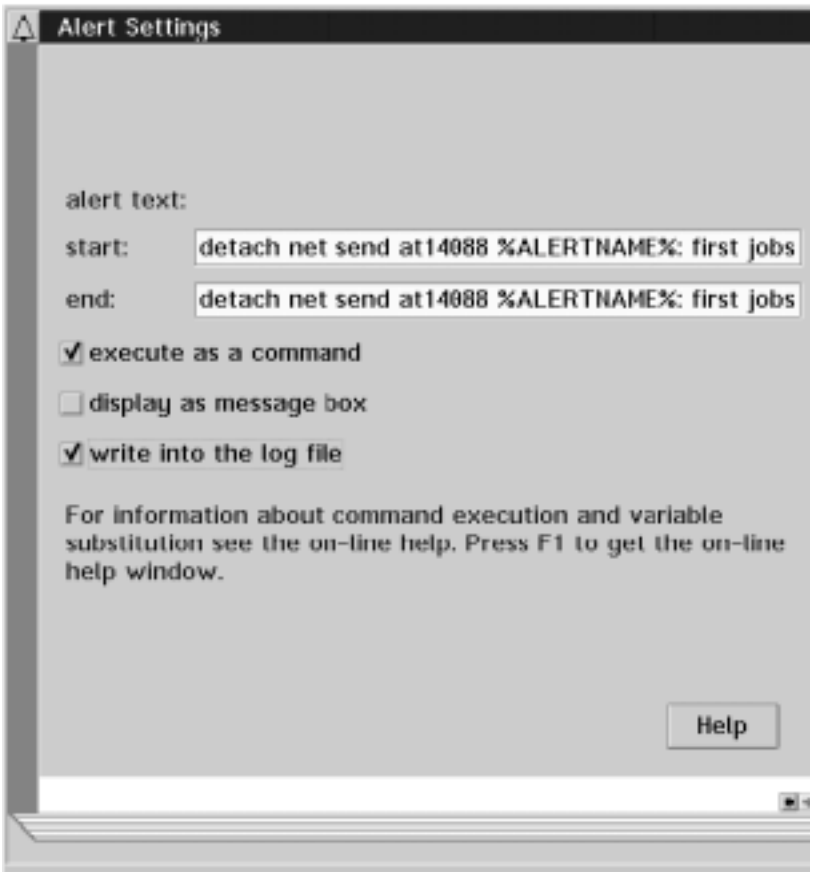


Figure 62. Alert definition page 3

More general, the user can define some text which may be

- executed as a command,
- displayed in a message box,
- written into a log-file.

A combination of all three options is possible. As mentioned above, separate commands are provided for the beginning of an alert and for the end of the alert condition.