TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Quality of Service IP Networking with Multiservice Admission Control

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieur der technischen Wissenschaften unter der Leitung von

o. Univ. Prof. Dr.-Ing. Harmen R. van As
und
Dipl.-Ing. Brikena Statovci-Halimi
E 388 Institut für Breitbandkommunikation

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht an der Fakultät für Informatik der Technischen Universität Wien

von

Vincent Chimaobi Emeakaroha
Matr.-Nr. 0027525. Kennzhl. 066 937

Wien, October 16, 2008 .... (Unterschrift Verfasser/in) .... (Unterschrift Betreuer/in)

Kurzfassung

Das Internet protocol (IP) unterstützte ursprünglich nur "Best Effort" Zustellungsstrategien für IP Verkehrsflüsse. Mit der schnellen Evolution von Kommunikationsnetzen und der Transformation des Internets in eine kommerzielle Infrastruktur, ergab sich das Verlangen nach Dienstgüte(Quality of Service). Durch die Verbindung von ursprünglich isolierten Netzen entstand die Notwendigkeit von Mehrdienst-Netzen, wodurch die Differenzierung von Dienstgüte, bedingt durch die unterschiedlichen Dienste, zur zentralen Herausforderung wurde.

Das Streben um Service-Qualität in IP Netzen zu garantieren führt zu Definitionen von Technologien wie Integrated Service, Differentiated Service und Multiprotocol Label Switching. Unterschiedliche technologische Ansätze verwenden Admission Control (AC) Mechanismen um Ressourcen-Verfügbarkeit sicherzustellen, und dadurch ausreichende Qualität, in Applikations-Verkehrsflüssen, zu gewährleisten. Es gibt unterschiedliche Ansätze für Admission Control - der Measurement-Based Admission Control (MBAC) Ansatz trifft seine Entscheidung, ob Verbindungsanfragen angenommen oder abgelehnt werden, anhand von aktuellen Netzlastmessungen.

Diese Diplomarbeit erarbeitet einen Ansatz und eine Implementierung eines Mehrdienste Frameworks, um Dienstgüte für mehrere priorisierte, unterschiedliche, Verkehrsflüsse, welche gleichzeitig ein Netz passieren, zu garantieren. Das Framework implementiert vier MBAC Algorithmen um Verkehrsflüsse zu steuern, sowie einen statischen und einen dynamischen Bandbreitenallokations-Mechanismus um die Zuteilung der Netzressourcen, zu den Verkehrsklassen, abhängig von ihrer Priorität, durchzuführen. Die Funktionalität des Frameworks wird durch sorgfältige Netzsimulationen mit VoIP, Video und "Best Effort" Applikations-Verkehrsflüssen, unter Verwendung des ns-2 Simulators, untersucht. Das Verhalten der Algorithmen wird durch Evaluierung von Netzauslastung, Paketverlust und Verzögerung, erforscht.

Abstract

Internet protocol (IP) networks are originally designed to provide best effort delivery services to IP traffic flows. With the rapid evolution of communication networks and the transformation of the Internet into a commercial infrastructure, demand for quality of service arised. Nevertheless, the interconnection of earlier isolated networks created the need for multiservice networks where the differentiation of service quality provided by the network became the central issue.

The quest to ensure service quality in IP networks leads to the definition of technologies like the integrated service, differentiated service, and multiprotocol label switching. Several technology designs use admission control (AC) mechanism to provide quality communication by ensuring resources availability for customer traffic flows. There are different approaches to admission control. The measurement-based admission control (MBAC) bases its decision of accepting or rejecting a new flow request, on the measurement of the current network load.

This thesis provides a proposal and an implementation of a multiservice framework for guaranteeing service quality to prioritized multi-class traffic flows, simultanously traversing a packet network. The framework implements four MBAC algorithms for controlling the flow admission process, as well as a static and a dynamic bandwidth allocation mechanism to manage the allocation of the network resources to the traffic classes according to their priority. The performance of the framework is proven by thorough network simulations for VoIP, video, and best effort applications using the ns-2 simulator. The behaviours of the algorithms are studied by evaluating network utilization, drop rate, packet loss, and the delay experienced by the traffic flows.

Acknowledgment

This work is the result of my master's thesis at the Institute of Broadband Communication (*Institut für Breitbandkommunikation*) at the Vienna University of Technology (*Technische Universität Wien*).

I cannot adequately express my deep sense of gratitude and thanksgiving to God who crowns all human efforts with success and whose love has been carrying me through. His blessings and protection have always been upon me.

I would like to express my deepest and sincere gratitude to my supervisor, Dipl.-Ing. Brikena Statovci-Halimi and o. Univ. Prof. Dr.-Ing. Harmen R. van As for giving me the opportunity to write this thesis and for their constructive support and guardiance throughout the work.

With overwhelming joy, i sincerely thank the family of *Gisela & Oliver Prisching* for making it possible for me to study here in Austria and for their hospitality, love and support all these years. May the good lord reward you abundantly.

I remain sincerely grateful to my close friends *Lucia & Michael Zehndorfer, Marianne & Kurt Zehndorfer, Opa Johann Radlherr & Barbara, Oma & Opa Prisching, Brigitta & Gehard Kuntner* without your friendship, love and support, i couldn't have achieved this success. I pray that the good Lord shall reward, guide and protect you all.

I owe lots of appreciations to my best friend and college *Bakk.techn Oliver Gündonner* for his time, technical advises and help throughout my studies. I will ever remain grateful to you.

My profound gratitude goes to *em. o. Univ. Prof. Dr Norbert Leser* for his friendship, help and making life enjoyable in Vienna. May the good lord bless and reward you.

Finally, i dedicate this work to my whole family especially my parents *Chief Prince & Lady Eugene Emeakaroha* and my big brother *Rev. Fr. Mag. Dr. Emeka Emeakaroha* thank you for steady encouragement, support and love. May God bless and protect you all.

Vienna, October 2008 Vincent Chimaobi Emeakaroha

Contents

List of Figures

List of Tables

1 Introduction

Traditionally, the Internet Protocol (IP) networks have solely provided a “best effort” delivery service for IP traffic. In these best-effort IP networks, all traffic is treated equally, it is processed as fast as possible, but there is no guarantee of actual delivery or how much could be delivered (throughput). This best effort mechanism has proven to be scalable, but with the increase in the number of connected host, it makes demands exceed network capacity. This condition noticeably degrades the delivered service and causes situation like excess packet loss, long delay and jitter.

In the last years, a big growth of Internet applications and a high increase of Internet users are experienced. These applications which include video conferencing, voice over IP, and video/audio streaming brought a high growth in demand for bandwidth and quality requirement. Nevertheless, the ongoing transformation of the Internet into a universal commercial infrastructure forces new changes to IP-Service infrastructure, both in bandwidth demand and in service requirements. Electronic sales, banking, finance, and collaborative work are few examples of this trend. They require real–time transfer of information, this consequently means that the IP Network must be upgraded in a fashion to fulfill these requirements. Hence the integration of multiservice over the Internet, is one of the main reasons behind the essence of provisioning quality of service (QoS) guarantee to end users. For an Internet-based web service the quality of service percieved by its users is the dominant factor for success [14].

The telecommunication environment in the recent years experienced the deregulation process, which brought many meaningful changes in the telecommunication laws that made increase in competition among service/network providers possible. The high competition is mainly provoked by the high performance requirement of the customers. Consequently, providers have to differentiate their products from those of their competitors. This requires the description of the roles of all entities that take part in the service provision and their requirements. The tool used to formalize the requirement of these entities is the service level agreement (SLA). An SLA is the fully specified and documented result of a negotiation between two parties; a customer and a service/network provider, that defines service characteristics, responsibilites and priorities of every party. An SLA may include statements about tariffing and billing, service delivery and compensations [77].

According to [19], QoS (either pronounced as "Q-O-S" or "kwos") is the term used to describe the science of engineering a network to make it work well for applications by treating traffic from applications differently depending upon their SLA requirement.

Several techniques and technologies have been developed to meet the demand of provisioning guaranteed QoS to end users. They include integrated service, resource reservation protocol, differentiated service, multiprotocol label switching, etc. These technologies work toward guaranteeing specific aspects of services, for example guaranteeing no packet loss and bounded delay for Voice over IP (VoIP) traffic flows. Some aspects of services like reliability, availability and dependability could be grouped as quality of communication services.

The QoS mechanism pointed out so far requires that some of the network components have to make decision at some point in time about admitting new flows into the network. Admission control mechanism provides decision algorithms that can be implemented in a router or gateway host to decide whether a new request could be admitted into the network without compromising the agreed QoS commitment of the existing flows in the network (RFC 1633) [7].

There are many different approaches to admission control which are discussed in detail in this thesis. Traditional approaches to admission control require an apriori specification of traffic characteristics in form of particular parameters. The admission decision is then based on comparing the specified parameters with the network load. This is the simplest and widely implemented model, and it guarantees absolute bound to QoS parameters with the drawbacks of poor network ultilization. This model is called parameter-based admission control algorithms (PBAC).

Another approach is the measurement-based admission control mechanism (MBAC). This approach does not rely on static specified parameters, rather its admission decision is based on the measured traffic characteristic in the network. This approach doesn't guarantee absolute bound on QoS parameters but promises a reliable bound. Thus this approach achieves high network utilization at the expense of weakened QoS commitments.

IP is becoming the convergence technology for multimedia services and consequently QoS is one of the hottest topics in IP networking. Although researchers have addressed many isolated areas of QoS provisioning in IP networks including admission control algorithms, little attension has so far been paid to the measurement-based admission control algorithms for multiservice traffic flows. In this master thesis, this deficiency is addressed by extending the existing measurement-based admission control modules in ns-2 to accomodate multiservice traffic flows. The algorithms are further extended with two resource allocation mechanism to share the network resources among the traffic classes.

Measurement-based admission control in computer networks uses three methods of verification: formal methods, experiments in real networks, or network simulation [56]. Formal methods are not suitable for measurement-based admission control algorithms, because they are based on measurement of the current network traffic flows. Experiments in laboratory for MBAC is expensive and not flexible. So network simulation is a well suited and widely used method of verifying MBAC algorithms.

1.1 The Organisation of the Work

This master thesis is organised in seven chapters. A short description of their contents is given in the following:

- *Chapter 2 QoS in the Internet:* This chapter provides an introduction and overview to the research area quality of service. QoS practically involves a range of functions and features like classification, scheduling, policing, and shaping within the context of QoS technologies (e.g Integrated service, Differentiated service) in order to ensure that a network delivers the SLA characteristics required by applications. Service level agreements (SLA) provide the context for IP quality of service. Application and service SLA requirements are inputs and also the qualification criteria for measuring success in a QoS design. Chapter 2 further compares the differentiated service with the integrated service and then discusses the multiprotocol label switching (MPLS) which is a traffic routing and forwarding mechanism introduced by IETF.

- *Chapter 3 Admission Control Description:* This chapter discusses the admission control mechanism, as a process used to determine if a new flow can be granted its requested QoS without affecting those flow already granted admission. The admission control process consist of different approaches and algorithms. Some of the approaches are still evolving. Chapter 3 presents different approaches, algorithms and mechanisms for measurement-based admission control. It further discusses the new proposed multiservice framework implemented to accomodate multiservice admission control in ns-2.

- *Chapter 4 Studying Network Performance with the Network Simulator Tool (ns-2):* The network simulator tool version 2 (ns-2) is an open source software developed and maintained at the information science institute (ISI) of the unversity of southern California. This chapter describes the usage of this tool, starting from downloading, installing, simulating, gathering and processing the achieved simulation results. The chapter further presents lectures with this tool that are beneficiary to researchers unfamiliar with the tool and for students interested in computer networks, to give them some practical feeling of how data are transported over networks.

- *Chapter 5 Code Description of the Multiservice Framework:* The code used to extend the existing MBAC algorithms in ns-2 to accomodate the simulation of multiservice traffic flows is described in this chapter. The chapter describes also the implementation of the bottleneck link for building the topology through which multiservice traffic transverses.

- *Chapter 6 Simulation Scenarios and Results:* This chapter describes two different simulation scenarios and two network topologies used for the simulation to verify the performance of the extended existing MBAC algorithms in ns-2 (multiservice framework). There are two bandwidth allocation mechanism integrated in the framework. One of the mechanism statically share the total bandwidth among the traffic classes and the other also does bandwidth sharing but with the extra capability of dynamically borrowing best effort class bandwidth to the higher priority classes. These two mechanisms are simulated and their achieved results compared.

- *Chapter 7 Conclusion:* This chapter concludes the master thesis and summarizes once more the essential points of the research work.

2 QoS in the Internet

The Internet protocol (IP) was created as a connectionless network layer protocol that makes no attempt to distinguish between different application types. Hence with the introduction of new delay or loss sensitive applications the IP networks are moving in current time from best effort level to networks that can provide different service levels. The resource requirements of these applications ranges from real–time delay sensitive interactive applications like voice over IP that requires low packet loss, bounded delay and jitter, to delay tolerant applications like e-mail and file transfer protocol (FTP). The essential difference in handling real–time delay sensitive and delay tolerance applications are that real–time applications must receive data within some short specified period of time otherwise the packets become worthless. In respect of this, Internet providers are facing the challenge of designing their networks to accommodate the customer's requirement of fast, reliable and differentiated services.

The term service in the telecommunications context is very popular. It boils down to the capability to exchange information through a telecommunication medium, provided to a customer by a service provider. Services have well specified features and parameters. The international telecommunication union (ITU) defines service in an IP environment (IP-based service) as *a service provided by the service plane to an end user and which utilises the IP transfer capabilities and associated control and management functions, for delivery of the user information specified by the service level agreements (ITU–T Y.1241) [42]* . ITU describes parameters, attributes and classes of IP–based services. The term quality can be defined as the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs. This definition is not so exact. In fact, the meaning of this term is very broad. For example, in telecommunications the term quality is commonly used in determining whether the service satisfies the user's expectations. The judgement, however, depends on various criteria related to the party rating the service. Customers assess it on the basis of a personal impression and in comparison to their expectations, while an engineer expresses quality in terms of technical parameters. This discrepancy may sometimes lead to misunderstandings. Hence, the term QoS is used in many meanings ranging from the user's perception of the service to a set of connection parameters necessary to achieve particular quality of service. Before diving into the definitions and details of quality of service, it is considered wise to first of all take a closer look at the service level agreement, which provides qualification criteria for measuring success in a QoS design [19].

2.1 Customer/Provider Service Level Agreement

Due to the deregulation process experienced in the telecommunication environment in the last decade, there arises high competition among service/network providers. This situation is further provoked by high performance requirements from customers arising due to the introduction of new applications. The service/network providers moved to upgrade their services so as to remain competitive and maintain their customers. Consequently this caused the definition of roles of all the entities that take part in service provisioning and their relationship. i.e the responsibility of the provider to assure quality of service required by the customer and the responsibility of the customer to compensate the provider. A useful tool to formalise the above mentioned inter-relationship between customer and provider is the service level agreement (SLA) .

2.1.1 Service Level Agreement Concept

SLA provides the context for IP quality of service. Application and service SLA requirements are the inputs and also the qualification criteria for measuring success in a quality of service design.

The ITU recommendation E.860 [39] defined service level agreement as *a formal agreement between two or more entities that is reached after a negotiating activity with the scope to assess service characteristics, responsibilities and priorities of every party.* An SLA may include statements about performance, tariffing and billing, service delivery and compensations. The content of the SLA varies depending on the service to be provisioned and includes the QoS parameters needed for the negotiated agreement [56]. Figure 2.1 shows graphically an example of SLA between a user and a service provider.

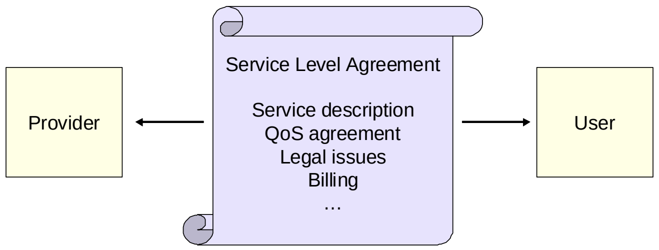


Figure 2.1: Example of service level agreement

The Internet engineering task force (IETF) introduced another definition of SLA in context of differentiated services. Based on [6], an SLA is a service contract between a customer and a service/network provider that specifies the forwarding service a customer should receive. A customer may be a user, an organisation, or other DiffServ domain. An SLA may include service conditioning rules which comprises a traffic conditioning agreement in whole or in parts. In order to solve the problem of assurance of QoS in a multi-provider environment, the concept of *one stop responsibility* was introduced [56]. This concept allows a user to retain a primary service provider with whom he agreed on SLA, as the only one responsible for the overall QoS the user receives. According to [55], service provider maybe an operator, a carrier, an Internet service provider (ISP), or an application service provider (ASP). Also the term customer could refer to companies, organisations or individuals that make use of telecommunication services provided by a service provider.

The service level agreement should be made up of service level objectives, service monitoring component, financial and compensation component, an expiry date, and authentication component. Because of this composition, the SLA should be formulated and expressed in a way the customer understands. Furthermore, it should contain rules governing the consequences of breaking the contract by any party. By means of an SLA, the provider can proof to the customer that the agreed services are being delivered and the customer can complain or demand compensation from the provider if the agreed service is not being delivered.

Normally, an SLA is seen as a contract between a customer and a service provider. This is referred as intra-domain SLA. SLA is not limited to this, but could also exist among different service providers which is then known as inter-ISP SLA. Figure 2.2 shows how the inter-provider SLA could be achieved. An SLA is created when a customer subscribes to a service offered by a service provider organisation. First of all the customer and the provider outlines their contract and agree to it. After this, the provider makes the necessary configuration like granting access, setting up the billing mechanism, to accommodate the new subscription. Lastly, the quality of service requirement from the customer are mapped into SLA parameters.

2.1.2 Service Level Specification

The service level specification (SLS) represents the technical part of an SLA. It is a set of technical parameters and their associated semantics that describes the service to be provisioned (network availability, throughput, latency). An SLS can be defined in template, then for a specific customer this template is instantiated, for example by setting the actual values per threshold.

The focus on service level rather than on network level enables the definition of service/SLA/QoS independently from the underlying network technology [55]. Service should be exactly defined by using service level specifications. The fol-

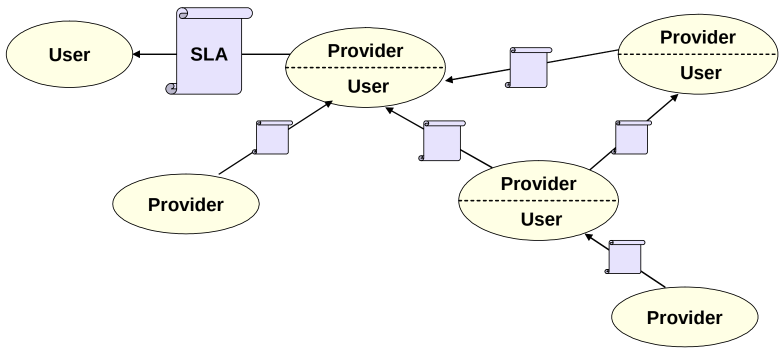


Figure 2.2: Inter–provider SLA

lowing type of information should be described:

- QoS metrics and corresponding thresholds that must be guaranteed by the service provider.
- Service performance measurement methods, measurement periods, provided reports (contents, format, frequency).
- Service schedule (activation time period).

The SLS should also define commitment over aggregated parameters. For example maximal unavailability time for all the service access points. Nevertheless, it should support various network interconnection models (e.g cascade, star, hub) and various traffic models. Currently, there exist different types of SLS but they all are based on the following criteria:

1. *Specified service:* The scope of services that can be defined with the SLS template.
2. *Information model:* The model of the SLS including categories used to classify the data.
3. *Data presentation:* The formalism used to describe the SLS data (e.g., XML, DTD, UML).

To guarantee that customers respect the agreements, traffic conditioning components (classifier, meter, shaper, dropper) are configured at the boundary between provider and customer.

2.1.3 Service Classification

Service can be consolidated in different categories. There are many ways to consolidate distinct services, their classification depends on the SLS parameters.

Service classification in IP networks is known as class of service (CoS). It is a broad term describing a set of characteristics available with a specific service. Both ITU and European telecommunication standard institute (ETSI) have their own definition for CoS, but IETF defined CoS as *the definition of the semantics and parameters of a specific type of QoS [15].* Services belonging to the same class are described by the same set of parameters, which can have qualitative or quantitative values. Usually, the set of parameters within the class is defined without assignment of concrete values, but these values can be bounded.

The idea of service classification is relatively mature. For example the original IP was intended to provide a simple way of classifying packets, but this capability of IP is rarely used. Traffic in asynchronous transfer mode (ATM) network is divided into classes as well. Currently, concrete service classes have been defined within the IP QoS architecture proposed by IETF, such as IntServ and DiffServ.

The following three classes of services are defined within the IntServ architecture: guaranteed service, controlled load service, and best effort service. Also in DiffServ architecture, the classification is based on the differentiated service code point (DSCP) coding. This coding allows the definition of 64 classes of service. Details about DiffServ are given in subsection 2.4.2 and for IntServ in subsection 2.4.1.

2.2 QoS Definitions

The term service is defined in telecommunications as the capability to exchange information through a telecommunication medium, provided to a customer by a service provider.

The ITU and ETSI used the quality of service definition first stated in the ITU document E.800 [38] as *the collective effect of service performance which determines the degree of satisfaction of a user of the service.* Further, the definition of quality of service was refined in 2002 and documented in E.860 [39] as *the degree of conformance of the service delivered to a user by a provider in accordance with an agreement between them.*

There exist three notions of QoS as defined in [34], named intrinsic, perceived, and assessed QoS. The intrinsic QoS relates to service features originating from technical aspects. Thus the intrinsic quality is decided by the network design and the provisioning of network access. Intrinsic quality is evaluated by comparing measured characteristic against expected performance characteristic. How the user perceived the service does not influence the grading of intrinsic QoS. The assessed QoS manifests in the decision of the customer whether to continue using the service or not. Such decisions depends mostly on the perceived quality, service price, and response of the provider to customer complaints and problems. It follows that even a customer service representative's attitude to a client maybe an important factor in rating the assessed QoS [26]. Neither ITU nor ETSI deals with the assessed QoS.

As the above definition suggests, QoS in the ITU/ETSI approach adheres mainly to perceived QoS rather than to intrinsic QoS. Nevertheless, they introduce the idea of network performance to cover technical facets. They make a clear distinction between QoS, understood as something focused on user–perceivable effects, and network performance, encompassing all network functions essential to provide a service. QoS parameters are user–oriented and do not directly translate into network parameters. On the other hand, the network performance parameters determine the quality observed by customers but are not necessarily meaningful to them [33]. But there must exist a consistent mapping between the QoS and network performance parameters.

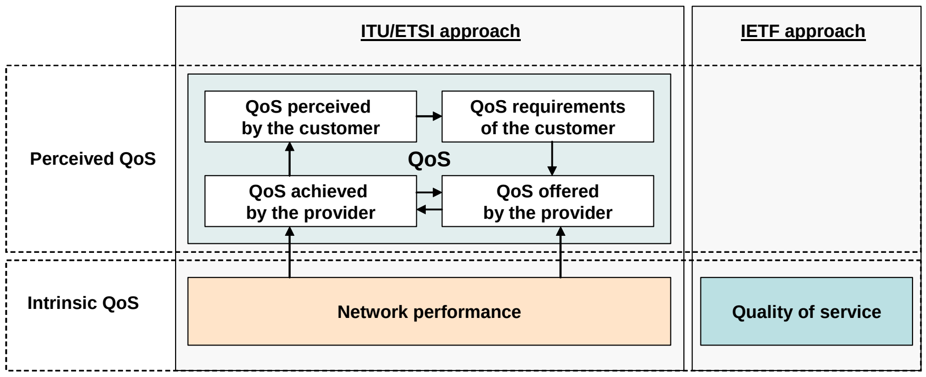

ITU/ETSI approach
IETF approach
Perceived QoS
Intrinsic QoS
QoS perceived by the customer
QoS requirements of the customer
QoS
QoS achieved by the provider
QoS offered by the provider
Network performance
Quality of service

Figure 2.3: A general QoS model

Figure 2.3 shows the relationship between the network performance and QoS. Network performance, as mentioned above, corresponds to intrinsic QoS. It is defined in E.800 [38] as *the ability of a network or network portion to provide the functions related to communications between users.* Network performance is defined and measured in terms of parameters of particular network components involved in provisioning a service. These parameters are the key to network efficiency and effectiveness in service provisioning. A high level of network performance is achieved by appropriate system design, configuration, operation, and maintenance. Some network performance parameters are defined by ITU in E.800, Y.1541, G.1000, and I.350 recommendations [38, 44, 40, 41]. To cover important aspects of QoS, ITU and ETSI distinguish four particular definitions (Figure 2.3):

- QoS achieved by the provider.
- QoS offered by the provider.

- QoS perceived by the customer.
- QoS requirements of the customer.

The customer requirements state their preferences for a particular service quality. They may be expressed in technical or nontechnical language understandable to both the customer and the service provider. The provider designs the service offered to the customer on the basis of the customer's requirements, even though the service provider may not always be in a position to meet the customer's expectations. The QoS offered may be influenced by the considerations of a service provider's strategy, benchmarking, service deployment cost, and other factors [40]. The quality requirements are mostly expressed in values assigned to parameters understandable to the customer e.g., *a basic telephony service availability is planned to be 99.95% in a year with not more than 15–minute break at any one occasion, and not more than 3 breaks over the year.* The QoS achieved is usually expressed by the same set of parameters. Comparison of the quality offered and achieved gives the service provider a preliminary grading of perceived service performance. However, the most important feedback, from the service provider's perspective, is QoS perceived by the customer, who finally rates the service quality comparing the experienced quality to his/her requirements. The ITU defines a set of QoS parameters in the E.800 recommendation [38].

QoS and network performance are closely interrelated. Ensuring high network performance is crucial to a successful service provision. The offered QoS parameters can be grouped into network– and non–network–related parameters. The former, in turn, can be translated into network performance parameters. These parameters are assigned target values. The achieved network performance is obtained on the basis of a parameter measurement. This serves as feedback to the network provider. The combination of the network performance achieved and non-network-related QoS constitutes the QoS achieved.

2.3 QoS Specification and Parameters

QoS specification is concerned with capturing application level QoS requirements and management policies. QoS specification is generally different at each system layer and is used to configure and maintain QoS mechanisms residing in the end–system and network. For example, at the distributed system platform level QoS specification is primarily application–oriented rather than system–oriented. Consideration in lower levels such as tightness of synchronization of multiple related audio and video flows, the rate and burst size of flows, or the details of thread scheduling in the end–system should all be hidden at this level [84]. QoS specification is therefore declarative in nature: applications specify what is required rather than how this is to be achieved by underlying QoS mechanisms. Quality of service specification encompasses the following:

- *Flow synchronization specification:* This characterizes the degree of synchronization (i.e.tightness) between multiple related flows. For example, simultaneously recorded video perspectives must be played in precise frame by frame synchrony so that relevant features may be simultaneously observed.

- *Flow performance specification:* This expresses the user's flow performance requirements. The ability to guarantee traffic throughput rates, delay, jitter and loss rates, is especially relevant for multimedia communications. These performance–based metrics are likely to vary from one application to another. To be able to commit necessary end–system and network resources QoS frameworks must have prior knowledge of the expected traffic characteristics associated with each flow before resource guarantees can be met.

- *Level of service:* This specifies the degree of end–to–end resource commitment required (e.g, deterministic, predictive and best effort). While the flow performance specification permits the user to express the required performance metrics in a quantitative manner, the level of service allows these requirements to be refined in a qualitative way to allow a distinction to be made between hard and soft performance guarantees. Level of service expresses a degree of certainty that the QoS levels requested at the time of flow establishment or renegotiation will be honoured.

- *Cost of service:* This specifies the price the user is willing to pay for the level of service. Cost of service is a very important factor when considering QoS specification. If there is no idea of cost of service involved in QoS specification, there is no reason for the user to select anything other than maximum level of service, (e.g., guaranteed service).

2.3.1 QoS Parameters

This subsection defines several IP QoS parameters supported by systems, including intrinsic and operational parameters. A quantification of these parameters constitutes an entire line of work, allowing to touch on the most important requirements based on IP. The ITU telecommunication standardization sector (ITU–T) , as well as the Internet engineering task force (IETF), and IP performance metrics (IPPM) working group, have made efforts to define a standard framework and provide definitions for IP QoS parameters. The ITU–T documented the results of its efforts as regards to standardizing IP QoS parameters in the Y.1540 recommendation [43]. The IPPM standard parameter framework is described in (RFC 2330) [68], where additional requests for comments (RFCs) exist for each IP QoS parameter. As in [33], IP QoS parameters are divided into two separate groups, namely the service–intrinsic group of parameters and the operational parameters, as represented in Figure 2.4.

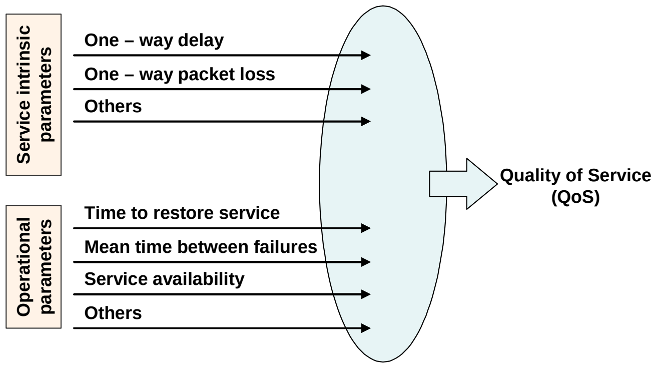


Figure 2.4: The service intrinsic and operational parameters

2.3.1.1 Intrinsic Parameters

The intrinsic QoS parameters expose the exact requisites that must be met for the service to conform to its SLA commitment. In packet networks, the intrinsic QoS parameters are expressed by the following [33]:

- **Throughput:** This represents the amount of data moved successfully from one place to another in a given period of time. Capacity and available bandwidth are the two bandwidth–related parameters.
- **Delay:** Defined as the notion of time experienced by packets while passing through the network. It may be considered either in an end–to–end relation or with regard to a particular network element. Packet delay has following components: network access delay, propagation delay, transmission delay, and queueing delay.
- **Jitter:** Expressed as one–way IP packet delay variation (IPDV) in RFC 3393 [18]. RFC 2679 [1] defined it as the difference between the one–way delay of a selected pair of packets within the same stream, going from one measurement point to the other measurement point.
- **Packet loss rate:** This is usually defined as the ratio of the number of undelivered packets to number of sent packets.
- **Network availability:** Defined as the probability that the network can perform its required functions.

These parameters describe the treatment experienced by packets while passing through the network. They can be translated into particular parameters of the network architecture components used to ensure QoS. They are finally mapped into the configuration of network elements. They are also closely connected with protocols used in the network and equipment abilities.

QoS is usually an end–to–end characteristic of communication between end hosts. It should be ensured along the whole path between peers, but the path may cross several autonomous systems belonging to various network providers, thus performance of all autonomous systems contributes to the final service quality. It is to a large extent difficult to define the parameters of perceived QoS, because they depend not only on the network architecture, nor technique, nor mechanisms used to ensure service quality. They are usually expressed in different terms but should be always somehow translatable into specific network parameters regardless of the network architecture. An example of an extensive set of parameters of the perceived QoS is provided by ITU in the E.800 recommendation [38]. These parameters are grouped into four subsets namely: service support, service operability, service servability and service security.

2.3.1.2 Operational Parameters

The operational parameters are related to the performance of an organization and reflect the overall quality of the organization's operational processes. Operational parameters are always service/technology–independent, they have therefore quite general characters. This implies that the definition of these operational parameters can be reused in service level agreements (SLAs) for different services more often than parameters directly depending on the provisioned service. The main operational parameters are described as follows:

- **Service availability percentage (SA):** This is defined as a percentage that indicates the time during which a contracted service specified in an SLA is operational at its respective service access points (SAPs) with respect to the scheduled service time. The service availability percentage thus indicates how well the service provider is doing in providing the customer with the service that it requested. The service availability parameter is the most important operational parameter to customers.

- **Reliability performance:** According to ITU–T recommendation E.800, it is defined as the ability of an item to perform a required function under stated conditions for a given time interval.

- **Mean time to failure (MTTF):** This is expressed as the expectation of the time to failure [38].

- **Mean time to restoration (MTTR):** Defined as the expectation of the time required to restore a device to an available status once it has entered a state of unavailability.

- **Mean time between failures (MTBF):** This is the average time between service outages applicable to a given period of observation [38]. It can be considered also as the mean time to failure plus the mean time to restoration.

The ITU–T recommendation E.800 notes that QoS can be characterized by the combined aspects of service support, operability, serveability, security (integrity) and other factors specific to each service. QoS thus depends on aspects directly related to network performance but is also influenced by so called human aspects, for example the ease of use of a particular service, which is a service aspect covered by operability performance. Each of the aspects should be considered as being characterized by many parameters.

2.3.2 Required Conditions for QoS

In order to provide QoS for more demanding of applications types (e.g., voice, multimedia), a network must satisfy two necessary conditions. The first condition is that bandwidth must be guaranteed for an application under various circumstances, including congestion and failures. The second condition is that as an application flow traverses the network, it must receive the appropriate class–based treatment, including scheduling and packet discarding. These two conditions could be thought as orthogonal. A flow may get sufficient bandwidth but get delayed on the way (the first condition is met but not the second). Alternatively, a flow may be appropriately serviced in most network nodes but get terminated or severely distorted by occasional lack of bandwidth (the second condition is met but not the first). Therefore, it is necessary to satisfy both of these conditions in order to achieve the hard QoS guarantees that are required by service providers and their customers.

2.4 QoS Technologies

A level of QoS assurance in an IP network depends on the amount of resource allocated to the traffic served. Different resource management techniques are used for resource allocation. In IP networks two resource management techniques can be used [26]. They are:

- Over-provisioning.
- Explicit resource management.

Over-provisioning is a resource allocation technique that aims at avoiding congestion and shortage of resources in a network In networks where such a technique is used, there is no differentiation of traffic flows i.e., all traffic is treated as a single service class. Therefore, all traffic is served with the same QoS level. The mechanism of guaranteeing QoS in such networks is to grant excess resources to traffic. The main argument for such a resource management technique is based on the fact that it is very simple and can easily be applied for a new traffic type. On the other side, this technique is less profitable for ISPs because there is no possibility to differentiate between services of different users, thereby causing problem for SLA, tariffing and billing.

Explicit resource management techniques are based on the concept of dividing all served flows into traffic classes that are served with various QoS levels. This requires additional traffic control mechanisms to be introduced to the standard IP network, such as admission control, policing, classification and scheduling. There are two well known QoS technologies that support explicit resource management. They are integrated service (IntServ) and differentiated service (DiffServ). There is also another technology that supports explicit resource management and is applicable to IP network called multiprotocol label switching (MPLS). These three technologies are described in the following subsections.

2.4.1 Integrated Services (IntServ)

The IETF has been examining how the Internet can be improved to provide QoS to traffic flows. They proposed a model for this purpose called integrated service (IntServ) [7]. This technique attempts to merge the advantages of two different paradigms: packet switched networks (which maximize network utilisation and adapt to network dynamics) and circuit switched network (which provide service guarantee and have difficulties in adapting to link failure) [13]. The IntServ makes the following assumption:

- Resources must be explicitly managed by applications in order to meet their requirements.
- New architecture should be an extension of the existing best effort IP network model, which supports real–time and elastic application with an expected QoS level.
- Data flows are independently served and cannot influence each other.

The IntServ model is characterized by per-flow resource reservation, which describes how an application negotiates the QoS level. IntServ support the integration of real–time and non–real–time traffic flows into a single Internet infrastructure, thereby enabling statistical sharing between these two traffic classes. Before a real–time application sents traffic over IntServ, it must first setup paths and reserve resources. Resource reservation protocol (RSVP) is a signalling protocol for setting up paths and reserving resources in the integrated service architecture.

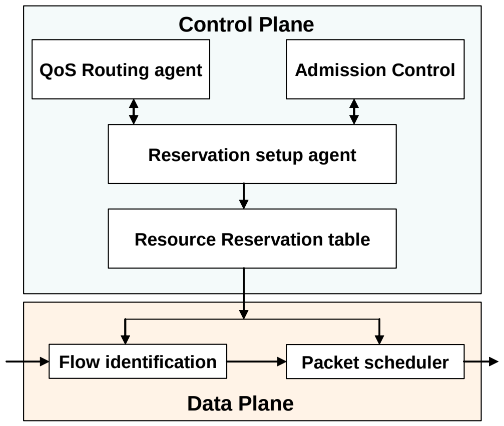


Figure 2.5: The integrated service model

The integrated service model can be divided into two parts: the control plane and the data plane as shown in Figure 2.5. The control plane is responsible for resource reservation and the data plane for forwarding the data packets based on the reservation state.

To setup a resource reservation, an application first characterizes its traffic flow and specifies the QoS requirements, a process often referred to in integrated service as flow specification. The reservation setup request can then be sent to the network. When a router receives the requests, it has to perform two tasks. First, it has to interact with the routing module to determine the next hop to which the reservation requests should be forwarded. Second, it has to coordinate with the admission control to decide whether there are sufficient resources to meet the requested resources. Once the reservation setup is successful, the information for the reserved flow is installed into the resource reservation table. The information in the resource reservation table is then used to configure the flow identification module and the packet scheduling module in the data plane. When packets arrive, the flow identification module selects packets that belong to the reserved flows and puts them to the appropriate queues; the packet scheduler allocates the resources to the flows based on the reservation information [33].

The control plane and the data plane must work together to guarantee the operation of the integrated service. A router that support the integrated service architecture, must at least implement the following components:

- Packet classification.
- Packet scheduling and queue management
- Admission control algorithm.

The router should also support the resource reservation protocol. In IntServ model, traffic flows are handled per–flow. A flow is defined as the unidirectional succession of packets relating to one instance of an application. Some times referred to as microflow [56].

2.4.1.1 Packet Classification

The process of categorizing packets into flows in an Internet router is called packet classification [32]. All packets belonging to the same flow obey predefined rules and are processed in the same manner by the router. Packet classification in the current IPv4 model is based on five IP header fields: source and destination addresses, source and destination ports, and protocol field [69]. These IP header fields are used to create a packet filter or rule, which is used to match the packets so as to differentiate them into flows. For example all packets with the same source and destination IP address maybe defined to form a flow. In IPv6, the 20-bit flow label field is introduced and used in flow identification. This label together with source and destination address can be used in 3-tuple packet classification as they are able to sufficiently distinguish packets belonging to various flows from different pairs of host.

2.4.1.2 Packet Scheduling and Queue Management

The task of the scheduler is to allocate resources to the individual traffic flows, and forward different packet flows using a set of queues. The packet scheduler must be implemented at the point where packets are queued [7], for example in the router. The router implements different queueing mechanisms ranging from very simple to complicated queueing mechanism. Examples of queueing mechanisms mostly found in routers today are: first in first out (FIFO)queueing, fair queueing (FQ), weighted fair queueing (WFQ), priority queueing (PQ), low latency queueing (LLQ), and round robin queueing (RR) [37]

The selection of a scheduling mechanism, which should operate at the output port of a router is one of the key design criterias for QoS networks. The router needs to distinguish between flows requiring different QoS (and possibly sort them into separate queues) and then, based on the scheduling algorithm send these packets to the outgoing link. Thus, a sophisticated scheduling algorithm is required to prioritize user traffic to meet various QoS requirements while fully utilizing network resources.

2.4.1.3 Admission Control Algorithm

Considering the fact that network capacity is limited, and the issue of guaranteeing QoS to flows, it is then necessary to control the access and the effects of new flows to the network. The solution to this problem can either be reactive (control schemes) or proactive (admission control) [56]. The reactive method detects and reacts immediately to congestion (e.g., flow control in TCP) but makes it difficult to guarantee QoS. The admission control method, assures that there are enough resources in the network for a new flow before accepting it.

In the IntServ model, the admission control component implements the decision algorithm used by a router or a host to decide whether a new data flow can be granted its required resources or not based on the fact of not compromising the QoS of the existing flow in the network. The admission control is also concerned with enforcing administrative policies on resource reservation [7]. Some of these policies may demand authentication of those requesting reservation.

The admission control mechanism should not be confused with policy control, which is performed at the edge of the network to ensure that host do not violate their traffic characteristics. This mechanism is rather considered a part of the packet scheduling mechanism [7].

2.4.1.4 Service Classes

The IntServ model defines three traffic flow classes. These classes are presented graphically on Figure 2.6 and are as follows:

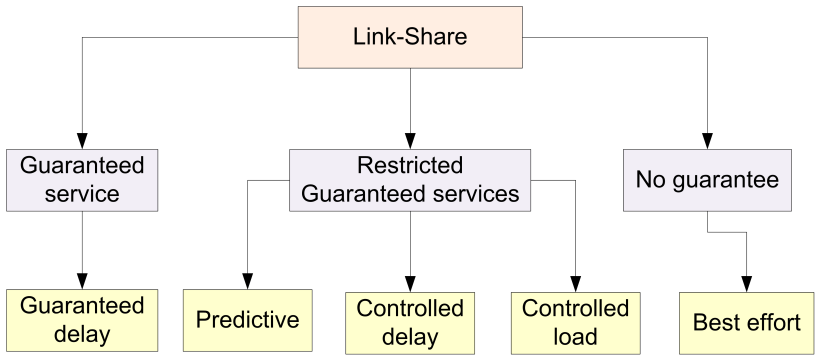


Figure 2.6: The integrated service class types

1. *Guaranteed class:* This class guarantees delay, bandwidth and packet loss. It can be used for real–time applications such as video and audio.

2. *Controlled load class:* This class offers better service than the best effort class. It is reliable for applications which require weak bound on maximum delay over the network and occasionally accept packet loss.

3. *Best effort class:* This is the default service class in the IP network. It is for applications that do not require strict quality of service like the email and file transfer protocol (FTP).

2.4.1.5 Guaranteed Service (GS)

Guaranteed service provides guaranteed bandwidth and strict bounds on end–to–end queueing delay for conforming flows. The service provides assured level of bandwidth or link capacity for the data flow. It imposes a strict upper bound on the end–to–end queueing delay as data flows through the network. The delay bound is usually large enough even to accommodate cases of long queueing delays. Guaranteed service in IP is described in RFC 2212 [76].

An application invokes guaranteed service by providing the traffic specification (*TSpec*) and the service specification (*RSpec*) to the network. The guaranteed service uses the general *token_bucket_TSpec* parameter to describe a data flow's traffic characteristics [76]. The flow specification *Flowspec* is used to set parameters in the node's packet scheduler or other link layer mechanisms, and the filter specification *Filterspec* is used to set parameters in the packet classifier, which is used to classify the incoming packet flows. Traffic specification describes traffic sources with the following parameters [33]:

- Bucket rate (r $[Byte/s]$) is the rate at which tokens arrive at token bucket.
- Peak rate (p $[Byte/s]$) is the maximum rate at which packets can transmit.
- Bucket depth (b $[Byte]$) is the size of the token bucket.
- Minimum policed unit (m $[Byte]$) is any packet with a size smaller than m which can be counted as m bytes.
- Maximum packet size (M $[Byte]$) is the maximum packet size that can be accepted ($m > 0$, $M > 0$, and $m \leq M$).

The service specification describes the service requirements with two parameters:

- Service rate (R $[Byte]$) which is the service rate or bandwidth requirement.
- Slack term (S $[\mu s]$) is the extra amount of delay that a node may add and still meets the end–to–end delay requirement.

Each router characterizes the guaranteed service for a specific flow by allocating a bandwidth R, and buffer space B, that the flow may consume. This is done by approximating the *fluid model* of service [66] so that the flow effectively sees a dedicated wire of bandwidth R between source and receiver.

A network element's (router or host's) implementation of guaranteed service is characterized by two error terms C and D, which represent how the element's implementation of the guaranteed service deviates from the fluid model. The error term C is the rate dependent one. It represents the delay a datagram in the flow might experience due to the rate parameters of the flow. The error term D is the rate independent, per element one and represents the worst case non rate based transit time variation through the service element. By definition [23, 76]each network element j must ensure the delay of any packet of the flow be less than:

$$d < \frac{b}{R} + \frac{C_j}{R} + D_j \tag{2.1}$$

Consider generalized processor sharing (GPS) [66, 67, 17] implementing the bandwidth guarantee mechanism. A newly arriving flow packet may experience a maximum delay of:

$$d_{max} = \frac{M}{R} + \frac{MTU}{link\ capacity} + link\ propagation\ delay \tag{2.2}$$

before it reaches the downstream element, where the first term is the worst case service time for this packet, the second term is the time for this element to transmit a packet with size equal to the maximum transmission unit (MTU) as its outgoing link capacity, and the third term is simply the physical propagation delay of the outgoing link [33].

2.4.1.6 Controlled–Load Service (CLS)

The controlled–load service (CLS) does not accept or make use of specific target values for control parameters such as delay or loss. Instead, the acceptance of a request for controlled–load service is defined to imply a commitment by the network elements to provide a service closely equivalent to that provided to uncontrolled (best effort) traffic under lightly loaded conditions. The service aims at providing the same QoS under heavy loads as under unloaded conditions. Though there is no specified strict bound on delay, it ensures that a very high percentage of packets do not experience delays highly greater than the minimum transit delay due to propagation and router processing [83]. The controlled–load service is conceived for adaptive real–time applications, which are highly sensitive to overloaded conditions. These applications have been shown to work well under unloaded networks but to degrade quickly under overloaded conditions. Thus some capacity (admission) control is needed to ensure that controlled–load application traffic flows are received even when the network element is overloaded.

A simple means of implementing controlled-load service is based on the existing capabilities of network elements, that support traffic classes based on mechanisms such as WFQ or class-based queueing. With these mechanism and a packet classifier, the CLS packet flows are mapped into a class with adequate capacity to avoid overload.
In order to achieve their stated goals and provide the proposed services, the IntServ models included various traffic parameters such as rate and slack term for guaranteed service; and average rate, peak rate and burst size for controlled load service [33]. To install these parameter values in a network and to provide service

guarantees for the real–time traffic, the resource reservation protocol (RSVP) was developed as a signalling protocol for reservations and explicit admission control.

2.4.1.7 RSVP signalling

The resource reservation protocol (RSVP) is an IETF–defined signalling protocol that uses IntServ to convey QoS requests to the network [8]. The IntServ architecture specifies extensions to the best effort traffic model. RSVP messages identify which application and user is requesting QoS, the service level requested from the network, the bandwidth requested, and the end nodes (source and destination addresses). Based on administrator defined admission control policies and network resource availability, the QoS request is either approved or denied by the host performing admission control duties. If the request is approved, QoS mechanisms are invoked to classify and schedule the traffic flow, logically allocate bandwidth, and notify the requesting host of the approval so that it might begin sending priority traffic flows. Until this occurs, the transmission is treated as standard traffic by the network. Information encapsulated in RSVP messages is per data flow and the messages may carry the following information:

- *Traffic classification information:* They are the source and destination IP addresses and the port numbers to identify the traffic flow (i.e., the filter specification *Filterspec*).
- *Traffic parameters:* They are expressed using IntServ's token–bucket model, these identify the data rate of the flow (i.e., flow specification *Flowspec*).
- *Service level information:* They originate from the IntServ–defined service types, and convey the flow requirements for the RSVP request.
- *Policy information:* This allows the system to verify that the requester is entitled to the resources and to the amount of resources being requested.

RSVP is a soft–state protocol, meaning that the reservation must be periodically refreshed or it expires. The reservation information, or state, is cached in each hop tasked with managing resources. If the network's routing protocol alters the data path, RSVP attempts to reinstall the reservation state along the new route. When refresh messages are not received, reservations time out and are dropped, releasing bandwidth. The sender refreshes *Path* messages, and the receiver refreshes *Resv* messages. Because RSVP sends its messages as best effort datagrams with no reliability enhancement, some messages might be lost, but the periodic transmission of refresh messages by hosts and routers compensates for the occasional loss of a signalling message. To ensure receipt of refresh messages, the network traffic control mechanism must be statically configured to grant some minimal bandwidth for signalling messages to protect them from congestion losses. At any time, the sender, receiver, or other network devices providing QoS, can terminate the session by sending a *Path_tear* or *Resv_tear* message. Figure 2.7 presents an example of the RSVP signalling mechanism.

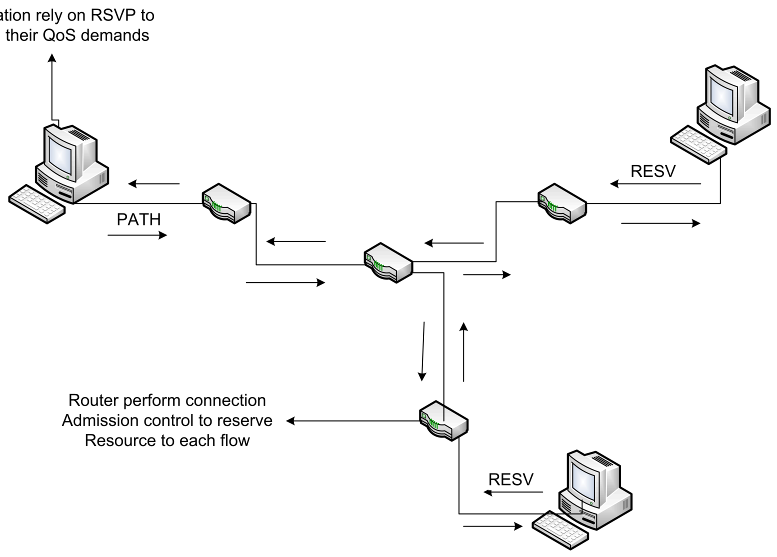

Application rely on RSVP to
Signal their QoS demands
PATH
RESV
Router perform connection
Admission control to reserve
Resource to each flow
RESV

Figure 2.7: RSVP signalling mechanism in IntServ

Policy is checked by the RSVP–aware routers and switches along the path. Devices might reject resource reservation requests based on the results of these policy checks. If the reservation is rejected due to lack of resources, the requested application is immediately informed that the network cannot currently support that amount and type of bandwidth or the requested service level. The application determines whether to wait and repeat the request later or to send the data immediately using best effort delivery [33]. QoS–aware applications, such as those controlling multicast transmissions, generally begin sending immediately on a best effort basis, which is then upgraded to QoS when the reservation is accepted.

The IntServ technology decouples routing from the reservation process, and uses the path established by the standard routing protocol like the open shortest path first (OSPF) [59] and the routing information protocol (RIP) to determine the next hops. Resource reservation protocol provides the highest level on QoS in terms of service guarantees, granularity of resource allocation and details of feedback to QoS–enabled applications [56]. However, per microflow service guarantee in the IntServ/RSVP architecture caused the well–know scalability problem [26]. This is not only a signalling processing problem but a problem of serving individual streams in all nodes in the network (microflow policing, classification and scheduling problems).

2.4.2 Differentiated Services (DiffServ)

The DiffServ is an alternative QoS model developed by the IETF to solve the scalability problem encountered in the IntServ/RSVP model [6]. The DiffServ is an architecture based on the idea of grouping traffic flows into a finite number of traffic classes [37]. As DiffServ is based on aggregates, it offers a scalable way of provisioning QoS, which was lacking in the IntServ architecture that uses per–flow resource reservation for individual traffic flows.

DiffServ networks have two main type of routers: edge and core router. The edge router is located at the boundary to the network and the core router in the heart of the network. Figure 2.8 presents an example of the DiffServ architecture.

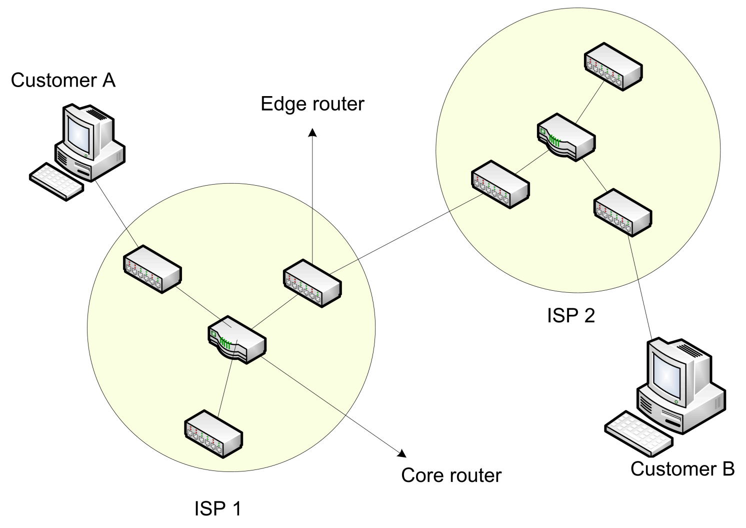


Figure 2.8: DiffServ network model

Traffic flows entering the DiffServ network are classified and conditioned at the boundary of the network only, and assigned to different behaviour aggregates. A DiffServ behaviour aggregate is a collection of packets with the same DiffServ code point (DSCP), crossing a link in a particular direction. The DSCP defines the service a packet should get in the network and its treatment within routers. Thus, independent flows select a predefined service and are served in the same way as other flows that choose the same service. Flows (packets) served by the same service are aggregated and experience the same QoS level. Aggregated packets processed by a network node are called per hop behaviour (PHB). A per hop behaviour describes the treatment for traffic (packets) belonging to a certain behaviour aggregate at an individual network node (router). If a PHB specifies to forward a packet preferential to all others, and is applied within all routers of a network this would result in a service providing noticeable better throughput and low delay to all packets with the appropriate DSCP.

The six bit wide DSCP allows to differentiate between $2^6 = 64$ different PHBs. Hence only twelve DSCPs are predefined for general usage. But there is no theoretical limit for an Internet service provider (ISP) to use non–specified DSCP. For example, a provider might map DSCPs at his border routers to provide a similar PHB within his own network. Router implementations should support the recommended code point to PHB mappings [33].

2.4.2.1 Differentiated Services Classes

During the introduction of DiffServ model by the IETF DiffServ working group, different service classes for different applications were proposed. Out of the many service classes proposals, only two are currently standardized as PHB. They are: expedited forwarding service and assured forwarding service.

Expedited Forwarding Service (EF): This service class is used to support applications with low–delay, low–jitter, low–loss, assured bandwidth requirement, such as VoIP [16]. The characteristics of an implementation which supports the requirements of this class are that, it is able to service the EF traffic at a specified rate or higher, measurable over a defined time interval and independent of the offered load of any non–EF traffic at the point where the EF PHB is applied [19]. The recommended code point for the EF PHB is 101110. The following properties characterize expedited forwarding service:

1. *Peak bit rate:* Used on flows or on aggregated flows.
2. *No bursts:* Allowed only within the peak bit rate.
3. *Low queueing delay:* Proposed for real–time applications.

Some scheduler implementations may attempt to support EF traffic using a scheduling algorithm such as weighted round robin (WRR) or WFQ. However, with such implementations the worst case delay bounds for EF traffic depends upon the particular scheduling algorithm used and may also be dependant upon the number of queues used in the particular scheduler implementation. Consequently, the EF PHB is typically implemented using a strict priority queueing mechanism. Using priority and short queues ensures that the arrival rate of EF traffic flows must not exceed the service rate at the interface.

Assured Forwarding Service (AF): This group defines a set of classes, which are designed to support data applications with assured bandwidth requirements such as absolute or relative minimum bandwidth guarantee, with a work–conserving property [19]. The key concept behind the AF PHB group definition is that a particular class could be used by a DiffServ domain to offer service, to a particular site for example, with an assurance that IP packets within that class are forwarded with a high probability as long as the class rate from the site does not exceed a defined contracted rate. If the rate is exceeded, then the excess traffic maybe forwarded, but with a probability that maybe lower than for traffic flows which are below the contracted rate. According to RFC 2597 [35], four AF service classes are defined. Each of these classes has three level of dropping precedence (low, medium, and high). Within a class, the drop precedence therefore indicates the relative importance of the packet. A set of twelve recommended DSCP values has been allocated to indicate the four classes and the three drop precedence levels within each class, as shown in Table 2.1.

Dropping precedence	**Class1**	**Class2**	**Class3**	**Class4**
Low	001010	010010	011010	100010
Medium	001100	010100	011100	100100
High	001110	010110	011110	100110

Table 2.1: Assured forwarding code point

Although only four AF classes are defined, in theory there is nothing, apart from the size of the DSCP field, to limit the number of different class of services that can be used with the AF forwarding behaviour. If more than four AF classes are required, then as the recommended DSCP markings are only defined for four classes, non–recommended DSCP values need to be used for the additional AF classes.

A particular AF class is realized by combining condition behaviours on ingress router at the boundary to the DiffServ domain, where a particular class is offered to a customer, which controls the amount of traffic accepted at each level of drop precedence within that class and mark the traffic accordingly. At that node and subsequent nodes, the AF class bandwidth is allocated to ensure that traffic within the contracted rate is delivered with a high probability. If congestion is experienced within the class, the congested node aims to ensure that packets of a higher drop precedence are dropped with a higher probability than packets of a lower drop precedence [19].

2.4.2.2 Differentiated Services Traffic Classification and Conditioning

The DiffServ model includes two conceptual elements in the ingress point of the network: a classifier and conditioning elements. These conditioning elements, are in general composed of *markers*, *meters*, *policers*, and *shapers*, as shown in Figure

2.9. When a traffic stream at the input port of a router is classified, it then might have to travel through a meter (used where appropriate) to measure the traffic behaviour against a traffic profile which is a subset of a SLA. The meter classifies particular packets as *in–profile* or *out–of–profile* depending on SLA conformance or violation.

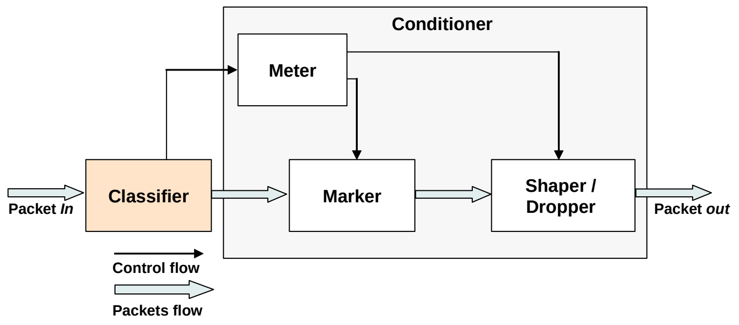


Figure 2.9: Traffic classification and conditioning clock

The traffic conditioning elements ensure that on average, each behaviour aggregate gets the agreed service. They can be applied at any congested network node when the total amount of in bound traffic exceeds the output capacity of the router. As the number of routers grows in a network, congestion increases due to expanded volume of traffic and hence proper traffic conditioning becomes more important. Traffic conditioners might not need all four elements. If no traffic profile exists, packets may only pass through a classifier and a marker.

Traffic Classifier: The traffic classifier identifies incoming packets and group them into aggregated streams based on the information in the packet header. It matches received packets to statically or dynamically allocated service profiles and passes those packets to an element of traffic conditioner for further processing. There are two types of classifier used for traffic classification [37]:

- *Behaviour aggregation (BA) classifier:* This type works on behaviour aggregates and classifies packets based on patterns of the DiffServ byte (DSCP) only. It is used mainly at the core network due to its simplicity.
- *Multi–field (MF) classifier:* It classifies packets based on any combination of the DiffServ field, protocol *ID*, source address, destination address, source port, destination port or even application level protocol information. It is usually used at the edge of the network to classify incoming packets.

Metering: This is a process to determine whether the behaviour of a packet stream after classification is within the specified profile for the stream (aggregate). The output result of metering is used to trigger events in other conditioning blocks. There are many estimators that can be used in implementing metering. According to [37], the most known and widely used estimator in the packet network is the token bucket estimator. (see Figure 3.2 in subsection 3.3.2 for detail explanation of token bucket)

Marker: Packet marking is a process where packets are marked to belong to a certain aggregated service class using predefined DSCP values. The marker can mark all packets which are mapped to a single code point, or mark a packet to one of a set of code points to select a PHB in a group, according to the state of a meter. The approach of DiffServ and especially the assured forwarding service with its different drop precedence levels, requires specialized marking components.

Shaper: The shaping process delays some packets in a traffic stream using a token bucket in order to force the stream into compliance with a predefined traffic profile. Dropping has similar objective as shaping but it drops packets in order to get the traffic stream into compliance with the predefined profile. A shaper usually has a finite–size buffer and packets are discarded if there is not sufficient buffer space to hold the delayed packets. Shapers are generally placed after each type of classifier. For example, shaping for EF traffic at the interior nodes helps to improve end to end performance and also prevents the other classes from being over flooded by a big EF burst. Hence, either a policer or a shaper is supposed to appear in the same traffic conditioner.

Policier: When classified packets arrive at the policer it monitors the dynamic behaviour of the packets and discards or re–marks some or all of the packets in order to force the stream into compliance (i.e., force them to comply with configured properties like rate and burst size) with a traffic profile. By setting the shaper buffer size to zero (or a few packets) a policer can be implemented as a special case of a shaper. Like shapers policers can also be placed after each type of classifier. Policers, in general, are considered suitable to police traffic between DiffServ domains (e.g., a customer and a provider) and after BA classifiers in backbone routers. However, most researchers agree that policing should not be done at interior nodes since it unavoidably involves flow classification. Policers are usually present in ingress nodes and could be based on simple token bucket filters.

2.4.3 Comparison of Differentiated Service and Integrated Service

The differentiated service model is different to the integrated service model in many aspects like the following:

- The amount of state information in DiffServ is proportional to the number of aggregated classes, rather than to the number of flows, as resources are allocated to individual classes that represents aggregated traffic.
- The fact that DiffServ classifies and marks packets only at the edge of the network and the core routers in the network performs only packet forwarding based on marking at the network edge makes DiffServ more scalable than the IntServ architecture which must perform classification, scheduling, admission control for each immediate node on the path.
- In the IntServ architecture, an application specifies the QoS requirements choosing a service and a related set of parameters, therefore a network must be ready to treat a large number of different QoS requests. In the contrary in DiffServ architecture, an application specifies a service selecting a per hop behaviour (PHB) in a limited set of choices.

Although the DiffServ architecture is quite scalable and simpler than the IntServ architecture, it still has some drawbacks in terms of guaranteeing quality of service. The DiffServ does not offer an end–to–end QoS guarantee, rather an edge–to–edge QoS. This makes IntServ to be better than DiffServ in this aspect. Figure 2.10 shows the comparison of DiffServ with IntServ and the default best effort service with regards to the level of QoS they can guarantee and the complexity of their implementation.

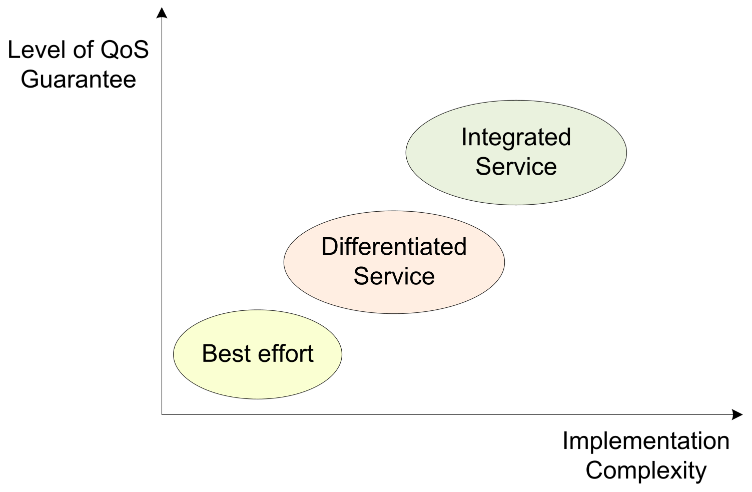


Figure 2.10: The comparison of DiffServ, IntServ and best effort in term of QoS guarantee level and implementation complexity

A more detailed comparison of IntServ architecture with DiffServ architecture, shows that they have some complementary features as shown in Table 2.2.

Feature	**IntServ**	**DiffServ**
QoS assurance	Per flow	Per aggregate
QoS assurance range	End–to–end (application–to–application)	Domain(edge–to–edge) or DiffServ region
Resource reservation	Controlled by application	Configured at edge node based on SLA
Resource management	Distributed	Centralized within DiffServ domain
Signalling	Dedicated protocol (RSVP)	Based on DSCP carried in IP packet header
Scalability	Not recommended for core network	Scalable in all parts of network
Class of service (CoS)	Guarateed service, controlled-load, best effort	Expedited forwarding, assured forwarding, and best effort

Table 2.2: IntServ and DiffServ complementary features

According to RFC 2998 [5], IETF has proposed an interoperability framework for the IntServ and DiffServ architectures. The integrated IntServ–DiffServ model is used to provide QoS in the end–to–end relation [26]. To avoid per microflow servicing in the core, the proposed architecture uses DiffServ in the core to support aggregated IntServ microflows. The IntServ model is used at the access part of the network to provide applications the signalling interface for them to place their resource reservation request and after granting them access, forwards their flows to the network. The QoS signalling is end–to–end. It takes place between the communicating terminals. Apart from per microflow resource reservation, RSVP signalling can be used to aid resource management in a DiffServ domain and some extension to RSVP supporting DiffServ are developed by the IETF.

2.4.4 Multi-Protocol Label Switching (MPLS)

The multiprotocol label switching (MPLS) is defined and standardized by the IETF [73]. It presents a core networking environment capable of carrying multiple traffic types over a common infrastructure while delivering class of service and true quality of service (QoS). It is a versatile solution to address the problems faced by today's network speed, scalability, QoS management, traffic engineering, and virtual private network support. MPLS has emerged as an elegant solution to meet the bandwidth management and service requirements for next generation

IP–based backbone networks [33].

By combing the best of network layer routing and link–layer switching, MPLS introduces a new forwarding paradigm for IP networks, and brings connection oriented properties similar to that of traffic engineering capabilities of ATM to IP networks, but in a very scalable and cost effective way. In addition, MPLS introduces a forwarding paradigm for IP networks by eliminating the need for routers to perform an address lookup for every packet. Thereby speeding up packet forwarding with improved efficiency.

2.4.4.1 MPLS Basic Architecture

The basic idea behind MPLS is to assign short, fixed–length labels (which are used as local identifiers) to packets at the ingress of an MPLS domain, based on the concept of forwarding equivalence classes (FEC). A forwarding equivalence class is a subset of packets that are all treated in the same way by a router. In the MPLS domain, the labels attached to packets are used to make forwarding decisions without recourse to the original packet headers.

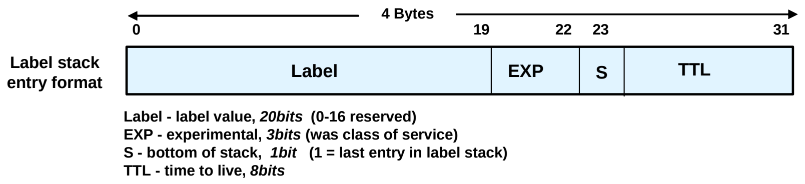

4 Bytes
0
19
22
23
31
Label stack entry format
Label
EXP
S
TTL
Label - label value, *20bits* (0-16 reserved)
EXP - experimental, *3bits* (was class of service)
S - bottom of stack, *1bit* (1 = last entry in label stack)
TTL - time to live, *8bits*

Figure 2.11: The MPLS header

An MPLS packet has a header, as illustrated in Figure 2.11 that is placed between the link layer (layer–2) header and the network layer (layer–3) header. The MPLS header, called a *shim header*, contains a 20–*bit* label, a three bit experimental (EXP) field (or class of service (CoS) field), a one bit label stack indicator, and an eight–*bit* time to live (TTL) field. When tunnelling labelled packets through multiple administrative MPLS domains, MPLS uses an ordered set of labels called label stack.
When a packet enters an MPLS domain, it is assigned a label, which specifies the path the packet must take while being inside this domain. Each MPLS router switches the packet to the outgoing port based only on its label. The experimental field is used to choose the correct service queue of the outgoing port. At the egress of the domain, the MPLS header is removed and the packet is sent to its destination using normal IP routing, as shown in Figure 2.12.

2.4.4.2 MPLS Label Distribution

In order that alabel switched path (LSP), which is a predetermined path, be used, the forwarding tables at each label switched router (LSR) must be populated with

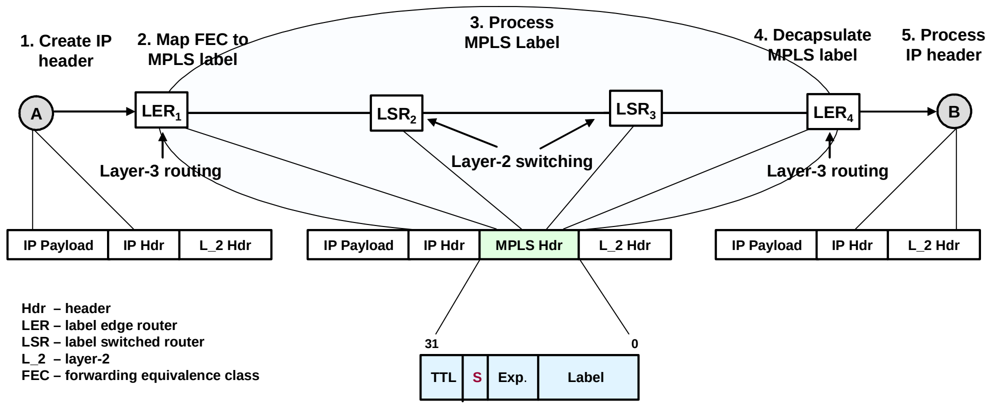


Figure 2.12: Creation and processing of MPLS header

the mappings from *incoming interface label value* to *outgoing interface label value*. This process is called LSP setup, or label distribution. The MPLS forwarding architecture document RFC 2547 [72], does not mandate a single protocol for the distribution of labels between LSRs. In fact it specifically allows multiple different label distribution protocols for use in different scenarios, including the following.

- Label distribution protocol (LDP).
- Constraint based routing label distribution protocol (CR–LDP).
- Resource reservation protocol extended for traffic engineering (RSVP–TE).
- Border gateway protocol (BGP).
- Open shortest path first extended for traffic engineering (OSPF–TE).
- Intermediate system intermediate system extended for traffic engineering (IS–IS–TE).

Several different approaches to label distribution can be used depending on the requirements of the hardware that forms the MPLS network, and the administrative policies used on the network. The underlying principles are that a path is setup either in response to a request from the ingress LSR (downstream–on–demand), or preemptively by routers in the network, including the egress LSR (downstream unsolicited). It is possible for both to take place at once, and for the LSP setup to meet in the middle.

2.4.4.3 Packet Forwarding in MPLS

In an MPLS network, layer–3 routing takes place at the edge and layer–2 switching is involved in the core. The labels effectively construct an LSP, which is used to forward the packets. After setting up the LSP, each core router in an MPLS network uses only the assigned label to make forwarding decision (layer–2 switching), without having to look into the original IP packet header. Label lookup and label switching are faster than an IP lookup because they could take place directly within the switched fabric and not in the central processing unit (CPU). An LSP is similar to an asynchronous transfer mode (ATM) virtual circuit and it is unidirectional from sender to receiver [85]. The packet forwarding using labels in MPLS network could be described as follows:

- When packets arrive at the ingress of an MPLS network, a decision is made based on the destination address or any other information contained in the IP header to determine the appropriate label value to be attached. This label value identifies the forwarding equivalence class (FEC). Apart from the packet's destination address, the ingress router may use some policy based consideration such as the packet's inbound port, its application type, or the class of service (CoS) written in the packet's header. The router attaches the label to the packet and forward it to the next hop. The label is locally important because two router can agree to use a label to signify a particular FEC among themselves.

- At the next hub the router uses the label as an index into a table that specifies the next hop and a new label for the destination. The label switch router attaches the new label to the packet and forwards it to the next hop in the destination direction.

- At the network boundary, the egress LSR receives the packet and removes the attached label, and then forwards the packet based on the content of the IP header to its destination.

2.4.4.4 Traffic Engineering in MPLS Networks

A practical function of traffic engineering in IP networks is mapping of traffic onto the network infrastructure to achieve specific performance objectives. High service quality, efficiency, survivability, and economy are crucial objectives in today's commercial, competitive, and mission–critical Internet [33]. Traffic engineering requires precise control over the routing function in order to achieve the objectives. An essential requirement for traffic engineering in IP networks is the capability to compute and establish a forwarding path from one node to another. This path must fulfil some requirements, while also satisfying network and policy constraints. Generally, performance objectives can be traffic–oriented and/or resource–oriented [3].

Traffic–oriented performance objectives relate to the improvement of the QoS provisioned to Internet traffic. Traffic–oriented performance metrics include packet loss, delay, delay variation, and throughput. The effectiveness of traffic–oriented policies can also be measured in terms of the relative proportion of offered traffic achieving their performance requirements. When service level agreements (SLAs) are involved, protecting traffic streams that comply with their SLAs from those that are non–compliant becomes an important factor in the attainment of traffic–oriented performance objectives.

Resource–oriented performance objectives relate to the optimization of the utilization of network assets. Efficient resource allocation is the basic approach to secure resource–oriented performance objectives. A traffic engineering system is said to be ”rational” if it addresses traffic–oriented performance problems while simultaneously utilizing network resource efficiently. Traffic engineering in conventional IP networks is a challenging problem. Singularities and discontinuities characterize Internet growth. Very rapid growth occurs over a relatively short interval time. This rapid growth is then followed by modest growth over relatively longer intervals of time. Accurate forecasting is therefore quite difficult. Furthermore, Internet traffic exhibits very dynamic behaviours with characteristics that are not yet well understood. Traffic also tends to be highly asymmetric. The operating environment is also in a continual state of flux. New resources are added constantly. New Internet applications with bandwidth requirements, which may have significant global impact, are introduced all the time. Facility location is also an issue. Sometimes network resources are sited in less then ideal locations due to facility constraints [33]. Additional complications are introduced by inter–domain traffic traversing autonomous system's boundaries. These environmental factors result in the network topology not usually correlating with the traffic matrix. The addressing of these issues requires continual monitoring and performance optimization of public IP networks. MPLS allows sophisticated control capabilities to be introduced in IP networks. These capabilities are based on the fact that MPLS efficiently supports origination connection control through explicit label switched paths (LSPs). For an explicit LSP the route is determined at the origination node. Once an explicit route is determined, a signalling protocol is then used to install the LSP. Through explicit LSPs, MPLS enables a reliable and efficient traffic engineering of core IP networks.

2.5 Summary

IP networks are originally designed for best effort services. Due to the introduction of real–time applications which are sensitive to resources availability and place performance demands on the underlying networks, the original IP networks could no longer satisfy the new requirements of these applications, hence there arises needs for improvement in the IP architecture to support quality of service. The notion of service level agreement laids the basis for specifying and agreeing on certain QoS for applications..

Some QoS technologies evolved over the past decade. The integrated service model together with resource reservation protocol are one of the first QoS mechanism introduced in the IP architecture. The IntServ had the assumption that resources must be explicitly managed by applications in order to meet their QoS requirements. IntServ with RSVP provided a genuine QoS architecture but however, had scalability and operational complexity problems. To solve these problems the IETF introduced the DiffServ model, which is scalable and does not require signalling protocol. Later, MPLS was introduced by IETF as connection–oriented approach to connectionless IP–based networks, and it supports traffic engineering. Figure 2.13 presents a possible combination of the three QoS assuring technologies.

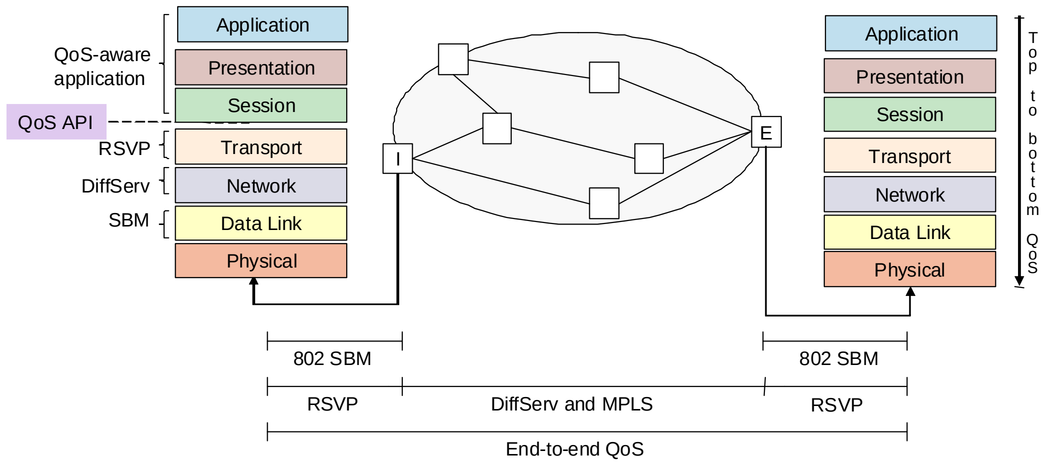


Figure 2.13: Combination of three QoS technologies

3 Admission Control Description

Emerging real–time multimedia applications traversing over IP networks place QoS requirements on their underlying transport medium. Connection–oriented network technologies such as ATM have an implicit admission control (AC) capability, which is used in establishing the path between the sender and the receiver, to ensure the required QoS for the connection. In contrast, IP network is connection–less, and has no implicit admission control capability.

In order to achieve tightly bound service levels for real–time traffic flows and to assure consistent service within the SLA bounds, resource reservation and admission control mechanisms are needed to ensure that the actual load of a class does not exceed acceptable levels. In the absence of admission control, a situation might occur where the available capacity for real-time traffic maybe exceeded implying a service quality degradation for that particular class. Thus, where admission control is not supported for traffic classes used for real–time applications like VoIP and packet video the bandwidth for that class must be over–provisioned with respect to the peak load in order to ensure that congestion does not occur [19].

Bandwidth over–provisioning causes significant financial cost. And, practically it may not be viable to provision every segment of the network to cope with the peak load. Further, if network planning and provisioning is inaccurate, or not reactive enough to new traffic demand, or there is a network failure, this may lead to situation where congestion is unavoidable. In such cases, all calls or streams in progress are degraded.

Admission control (AC) is in general a mechanism of traffic management, which consist of admitting a new traffic source if and only if the network can accommodate the new flow while still supporting existing commitments made to sources already accepted [57, 58]. An AC procedure is employed to maintain a high utilization of network resources while still preserving the QoS of existing flows. In this thesis, the considered admission control algorithms make their decision on a per flow basis.

3.1 Required situations for Admission Control

In general terms, admission control is practically useful, if the following situations are present:

1. ***The offered load may exceed the available capacity in the absence of admission control:*** If there is always enough bandwidth for a flow or a class to support the offered load then you simply do not need admission control. Thus, one approach to providing guaranteed support for service such as voice is to provision sufficient class bandwidth throughout the network to be able to ensure that the peak voice load can be serviced. However, one needs to consider the limitation of network component failure to provisioning of sufficent bandwidth throughout the network. In such cases, some part of the network may lack the required bandwidth to service the peak voice load. This leads to failure in delivering the required QoS. Therefore admission control can be used to overcome such situations.

2. ***Service utility degrades unaccaptably as a consequence of exceeding available capacity for that flow or class of service:*** For real–time applications, as bandwidth availble for its traffic class decreases, the utility of the applicatioin reduces. For example, consider a link, which has class capacity to support a maximum of twenty concurrent VoIP calls, within the bounds of the required SLA. If a twenty–first call is allowed to be setup, congestion will occur within that class and the service to all of the calls are degraded. In such a case, admission control can be used to block the twenty-first call, thereby preventing the existing calls from being degraded. The mantra for applications which need admission control is that it is much better to refuse a new call than to degrade service for many calls in progress [19].

3. ***The source application knows how to respond to an admission control failure:*** Admission control is only useful if there is some way of communicating an unsuccessful admission control decision back to the end–system application such that it does not establish the requested flow or stream, and such that it can communicate the failure back to the end user. e.g., for a VoIP call by returning a busy signal.

4. ***It is acceptable from a service perspective to reject a request:*** If from a service perspective, it is not acceptable for admission control to reject a requested call or session, then more bandwidth is needed instead of using admission control. For example, for a residential broadcast video service, it would be unacceptable to have an admission control failure while simply changing a channel.

3.2 Related Work and Research in this Area

Admission control has long been considiersed a key mechanism to ensure quality of service objectives in IP networks. There is a significant amount of research done, and many papers written in the area of admission control algorithms. Most of the admission control algorithms are concentrated on providing QoS for a single guaranteed service using parameter-based admission control algorithms or ensuring QoS for single predictive or single controlled-load services using the measurement-based admission control algorithms. The area of supporting QoS for multiclass of controlled-load or predictive service has not exprienced much research work when compared to the single class case.

Reference [82], presesents a tutorial on admission control in multi-service IP networks. The paper took on the ontological perspective within which to catagorize admissioin control schemes. It summarizes the characteristics of existing admission control schemes and went on to investigate the linkage of AC with the capacity planning process, examination of applications in typical core and access network architectures, and consideration of the factors involved in scaling up AC as an IP network function for large-scale, multiservice wide-area networks.

Within service management, admission control has been recognised as a convinent mechanism to keep service under controlled load and ensure the required QoS levels, bringing consistency to the services offered. The authors of [54], studied different AC approaches and the role of AC in multiservice IP networks. They identified the following high-level characteristics distinguishing AC approaches:

- **Underlying network paradigm:** The type of network models under which AC can operate, which ranges from single service to multiservice architectures.

- **Type of service to control:** The application characteristics usually specifies the type of service, whether they are rigid or adaptive and whether they have quantitative or qualitative QoS targets that determines service level guarantees to be provided.

- **Signalling supported:** The means of the application to explicitly inform the network of their needs. Commonly expressed in terms of service profile, using soft or hard state signalling.

- **Location of the AC decision:** This aspect involves the centralized or distributed nature of the AC. This is further dependant on which nodes are involved in the AC process.

- **Admission decision criteria:** The decision criteria is determined by the nature of the algorithm. Whether it is parameter-based, measurement based, or hybrid.

In [53], a distributed admission control for multiservices in IP network is presented. The paper discusses the handling of concurrent admission control decisions. It notes that distributed AC model by its nature is likely to involve multiple and simultaneous AC decisions, consequently presenting the need to handle the concurrency which arises from having multiple decision points. This offers a way to avoid over/false acceptance of flow entering the network, which causes resource overload and service degradation. This paper proposes some alternatives to tackle the problem of concurrent AC decision in multiservice IP network. The proposed alternatives include the definition of:

- A per-class concurreny index.
- A token-based system.
- A rate-based credit system controlled by the egress nodes.

These alternatives are an extension of the work done in [4]. The work in [4], defined an AC limit for acceptance of traffic flow within a class. The AC limits are defined off-line at an initial provisioning phase taking as input: (i) the network topology, (ii) the long-term expected traffic metrices, and (iii) the bandwidth sharing policies among classes. The initial static limit can be extended dynamically by sharing unused AC limit between egress routers. The authors of [53], also outline the issues with centralized AC approaches. They argued that the main advantage of centralized AC approaches is that centralizing state information and control task allows global vision of the domain's QoS and operation, relieving the control plane inside the network. The centralization process also supports creating and changing service policies and control mechanism such as AC algorithms. The cost of centralized approaches is however high. Central entities need to store and manage large amount of information, which in large and highly dynamic networks with many signalling messages and information updates requiring to be processed in real-time are even hard or impossible to realize.

3.3 Admission Control Approaches

As already discussed in section 3.2, there are different approaches to admission control mechanism. In this thesis, the admission decision criteria and distributed approaches to admission control are consider closely. Examples of these approaches include parameter-based admission control (PBAC), which is explained in subsection 3.3.1, and measurement-based admission control (MBAC), which is detailed in subsection 3.3.2. The main criteria ([49]) used in evaluating any admission control algorithm are the following:

- How well it fulfils its primary role of ensuring that service commitments are not violated.

- How high a level of network utilisation an admission control algorithm can achieve while still meeting its service commitment.
- The implementation and operational cost of the admission control algorithm.

These three criteria are going to be considered while discussing the different approaches to admission control.

3.3.1 Parameter-based Admission Control

In this approach, admission control is based on the assumption that the algorithm has perfect knowledge of each traffic source type that traverses every link. It also knows the current number of established service instances. This information enables admission control to compute the total amount of bandwidth required. Hence it will only accept a new service request if the minimum amount of bandwidth required by the total number of established service instances, including the new one is less than the available service rate (bandwidth). Typically sources are characterized by either peak and average rates [21] or a filter like token bucket [64]. These source characterizations provide upper bound on the traffic flows that can be generated by the source. It is obvious that this approach is optimal if the traffic is accurately characterized and used for admission control decision. Thus, the performance of this approach provides the upper bound for all other admission control approaches as far as the traffic sources are conformant.

Traditional real–time services provide a hard or absolute bound on the delay of every packet. According to reference [13], such service model is called guaranteed service. It uses a prior characterization of traffic sources. Network utilization under this model is usually acceptable when the provided traffic characteristics represents the actual behaviour of the source. However, when traffic characteristics do not depict the actual network behaviour, network utilization degrades inevitably low, since no traffic measurement is taken into consideration. PBAC algorithms can be analysed by formal methods, this makes the algorithms simple and easy to implement.

The simple sum algorithm is a representative of the parameter-based admission control algorithms. It can be described as follows:

Simple Sum: This admission control algorithm is based on the a prior well characterization of the traffic source. It simply ensures that the sum of reserved resources plus the new flow request does not exceed link capacity. Let ν be the sum of reserved rate, μ the link bandwidth, α the name of a flow requesting admission, and r_α the rate requested by flow α. This algorithm accepts a new flow if the following condition is true [49]:

$$\nu + r_\alpha < \mu \tag{3.1}$$

Due to the simplicity of this algorithm, it is one of the most widely implemented by switch and routers vendors.

The PBAC algorithms are simple and easy to implement. They can be used to provide guaranteed quality of service to hard real–time applications in terms of packet loss and delay. In reference [51], the authors evaluated many algorithms of this family. The problem of algorithms in this family is that, when the description of the source traffic does not match the traffic behaviour in the network, the algorithm may either incorrectly admit too many flow which leads to violation of the QoS commitment, or it may deny access to flows which could have been admitted successfully and this causes poor network utilization.

3.3.2 Measurement-based Admission Control

There are different types of measurement-based admission control algorithm. These result from work based upon a wide variety of theoretical foundations, different system requriements, different policies controlling the admission process and thus different behaviour requirements to satisfy certain service models [25, 57, 22, 46, 28, 49, 78, 13, 52, 75, 24, 29, 71, 70, 80, 81]. In this subsection, some introduction and explanation of MBACs and the services they can support are first presented. After that, different approaches to MBAC from different authors are disscussed. Lastly, the components of MBAC are described and four algorithms from the MBAC algorithm family are presented.

The MBAC algorithms provide an alternate approach to admission control. This approach tries to solve the problems of parameter-based admission control by shifting the task of source traffic specification from the user to the network [28]. Rather than the user specifying their traffic characteristics, the network attempts to learn the characteristics of existing flows by making on–line measurements [29]. This approach has a number of advantages such as the user–specified traffic descriptor can be very simple e.g., using peak rate which can be easily policed. An overly conservative specification does not result in over–allocation of resource for the entire duration of the service session [78].

Measurement–based admission control schemes were designed to statistically share network resource among flows so as to achieve high network utilization. The IETF developed an architecture in an effort to support real–time applications in the integrated service packet network [7]. The beauty of this architecture lies in the fact that many real–time applications can be adaptive, i.e., could adjust to network situations and tolerate some SLA violation in terms of packet loss and delay. This architecture is known as predictive service. Algorithms supporting predictive services use measurement of current network load instead of relying on a prior traffic characteristics. They use the a prior traffic characterization only for incoming flows, and measure the characteristics of the traffic already admitted into the network.

Based on the fact that MBAC algorithms rely on measurements and the traffic source behaviour is not always static, service commitment made by such algorithms can never be absolute [49]. Thus measurement–based approach to admission control can only be used in the context of predictive service and other more relaxed service commitment. The gain in network utilization becomes very significant when there is a high level of traffic multiplexing, because when different flows are multiplexed, the quality of service experienced depends often on their aggregate behaviour. According to the law of large numbers, the statistic of an aggregate traffic is easier to estimate than those of individual flows [56].

Applications requesting controlled–load service [83] may assume that its packet loss rate is on the order of the transmission medium's error rate and its typical experienced delay should be on the order of the path's transmission and propagation delays. More specifically, average packet queueing delay should not be greater than the flow burst time (*where flow burst time can be defined as the time required to serve the flow's maximum burst at the flow's reserved rate*) and there should be minimal loss rate averaged over time–scales larger than the burst time. For a flow described by the token bucket filter (see subsection 3.3.2.3), the burst time is b/r, where b is the token bucket depth and r is the arrival rate.

3.3.2.1 Different MBAC Paradigma

In this subsection, different approaches of measurement-based admission control mechanisms are presented.

In [22], Floyd presented a measurement-based admission control procedure for the controlled-load service that is based on the approach of equivalent capacity. The paper defined equivalent capacity of a class of traffic as that value $C(\epsilon)$ such that the stationary arrival rate for the class exceed $C(\epsilon)$ with probability at most ϵ. An admission control scheme that esimates the equivalent capacity of a class is proposed. A connection is admitted to a class of traffic, if with the addition of the new connection, the equivalent capacity of that class would be less than the allocated bandwidth for the class. This allows a simple fairly easily computed admission control procedure. For this MBAC algorithm, it is assumed that traffic description include token bucket parameters with the token rate and bucket size as in the controlled-load service, and that these token bucket parameter are policed at the router. Traffic from admitted connection that exceeds these token bucket parameter is forwarded on a best effort basis if resources are available but does not have to be considered by the admission control procedure. The admission control procedure uses the policed token bucket parameter to infer a peak rate for each flow over some fixed time interval. For such an admission control procedure, a request for admission is most likely to be accepted if the token rate is set close to the peak rate of the flow, and the bucket size is set to some small value to accommodate the accumulated jitter as packets are pushed in the network.

The authors of reference [46], described a measurement–based admission control algorithm for predictive service. Their admission control algorithm mechanism consist of two logically distinct aspects. One aspect is a set of criteria controlling whether to admit a new flow. This aspect is based on an approximate model of traffic flows and used measured quantities as inputs. The second aspect is the measurement process itself. In this mechanism, sources requesting the service must characterize the worst-case behaviour of their flows. Flows are characterized by token bucket filter, which has two parameters - its token generation rate r and the depth of its token bucket b. It is assumed that packets are of fixed size and each token is worth of a packet. Sending a packet consumes one token. When admitting a new flow not only must the admission mechanism decide whether the flow can get the service requested, but it must also decide if admitting the flow will prevent the network from keeping its prior commitments. Let us assume for the moment, that admission control can not allow any delay violation. Then the admission mechanism must analyze the worst-case impact of the newly arriving flow on existing flows' queueing delay.
The second aspect, which is the measurement process, is based on a simple time window mechanism. The measurement process uses the constants λ, S, and T. The authors of [46] made two measurements: the experienced delay and utilization. To estimate delay, the queueing delay ($\hat{d}$) of every packet is measured. To estimate utilization, the usage rate of guaranteed service, V_G^S and of each predictive class j, V_j^S, over a sampling period of length S packet transmission unit is sampled. To measure the delay, the measurement variable D_j tracks the estimated maximum queueing delay for class j. A measurement window of T packet transmission units is used for the basic measurement block. The value of D_j is updated on three occasions. At the end of the measurement block, D_j is updated to reflect the maximal packet delay seen in the previous block. Whenever an individual delay measurement exceeds this estimated maximum queueing delay, it shows that the estimated delay is wrong and D_j should be immediately updated to be λ times this sampled delay. The parameter λ allows the estimation to be more conservative by increasing D_j to a value higher than the actual sampled delay. Finally D_j is updated whenever a new flow is admitted. The utilisation is measure analog like the delay.

Authors of [75], proposed a dynamic call admission control in ATM networks. This procedure is based on measurement of the number of cells arriving during a fixed interval. They pointed out that an admission procedure based on actual measurement of the arrival rates at the gateway tolerates possible errors in the policing at the edge of the network. In the paper, after a new connection is admitted, the traffic parameters of the new connection are used to update the estimate of the distribution of cell arrival. This estimated distribution is then updated using exponential weighted moving averages.

In reference [28], the authors presented a framework for robust measurement-based admission control. They studied the performance of admission control schemes under measurement uncertainty and flow dynamics. They proposed the use of appropriate amount of memory for the estimator. By means of heavy-traffic approximations, the analysis of the resulting dynamical system is simplified through linearization around a nominal operating point and by Gaussian approximations of the statistic. The gain they obtain showed the impact estimation error could have on the QoS performance of MBAC schemes. As a consequence, it was demostrated by the authors how the memory time scale affects the performance, and they went on to show which memory time scale choice could achieve robust performance.

There are principally different motivations to MBAC, but in general they show some common features like [9]:

1. Many of the existing MBAC algorithms are greedy, in the sense that they admit new flows as soon as resources become available. For example in the presence of heterogenous flows (in terms of bandwidth requirement) the greedy nature of the algorithms leads them to discriminate against larger flows. In the limit of large demand, the MBACs will only admit small flows. Anytime there is room for a small flow one will be admitted, so there will never be enough bandwidth available for a larger flows.

2. The algorithms are local in the sense that an independent admission control is made at each hop along a path. Consider a situation where the traditional per-hop admission is applied, a flow is then only admitted if the flow passes the local admission criterion at every hop. If it fails at a single hop the flow is rejected. In the limit of large demand, the network will tend to admit only flows that traverse relatively short path.

These two features tend to define an implicit policy to admit small flow rather than large ones and flows traversing short paths (in terms of network hops) rather than long ones. Nevertheless, the authors of [48], pointed out another feature, which several of the existing MBAC have in common, that is their admission control equations give essentially the same performance. So they suggested that further research on MBAC should not focus on the design of the admission control equations, instead should focus on the settings of the measurement parameters and on global issues involed in MBAC such as the issue of admitting small flows and flows traversing short paths.

3.3.2.2 MBAC Components

The measurement-based admission control is made up of three components, which work together to admitt or reject flows so as to control the network load. They are named as follows:

- Traffic descriptor.
- Admission decision algorithm.
- Measurement mechanism.

The third component is made up of estimation modules. The interaction among the components is presented in Figure 3.1:

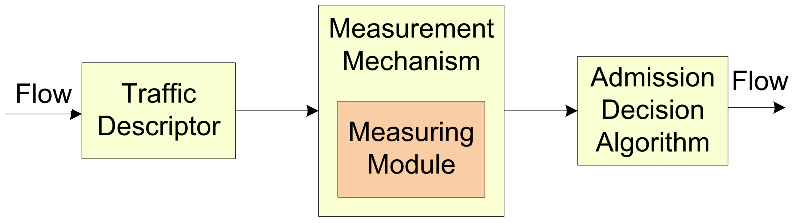


Figure 3.1: Components of measurement-based admission control

3.3.2.3 Traffic Descriptor

The traffic descriptor is a set of parameters that is used to characterize a traffic source. A typical traffic descriptor is the token bucket. The token bucket is a policing unit in the network [45]. It monitors the traffic that is generated by a single source and if necessary, limits the traffic flows by dropping individual packets. Figure 3.2 presents the function of a simple token bucket (STB). The token bucket can be described by two parameters: token generation rate r and the token bucket size b. The depth of the token bucket determines the maximum burst size that can be sent to the link. If the token overflows, they are simply discarded and not stored. To transmit a packet of B size, the corresponding amount of tokens are reduced from the token bucket. Each token represents a number of bytes, and the packet can only be sent if there are enough tokens in the bucket. Thus, when the token bucket is empty arriving packets are either queued or dropped.
If the token bucket is full, a maximum burst of b bytes can pass the token bucket without being affected. However, in the long run the average data rate cannot be greater than r.

The complex token bucket (CTB) is a more advanced type of token bucket filter [45]. In addition to STB's functionality, it has the capability of limiting the peak rate p of a source. Even if the bucket is full, the source cannot necessary sent packet burst with link speed [56]. Every traffic source can now be characterized by the given token bucket parameter (r,b) or (r,b,p) for STB and CTB respectively. For example a source seeking admission will characterize its traffic with token parameter r and b such that over a period of time T, the traffic generated by the source will not exceed $rT + b$ [11].

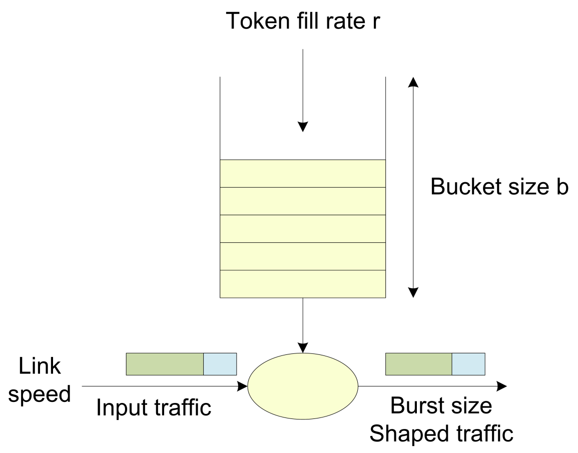


Figure 3.2: Token bucket operation

3.3.2.4 Admission Decision Algorithms

In this masters thesis, four measurement-based admission control algrithms are discussed. The algorithms are as follows:

- **Measured Sum (MS)**
- **Equivalent Bandwidth based on Hoeffding Bounds (HB)**
- **Acceptance Region (Tangent at Peak (TP) and Tangent at Origin (TO))**

Measured Sum (MS): The measured sum algorithm ([49]) uses measurement to estimate the load of existing network traffic flows. Let ψ be the measured load of existing traffic, μ the link bandwidth, α the name of a flow requesting admission, and r_α the rate requested by the flow α. This algorithm admits the new flow if the following condition is true:

$$\psi + r_\alpha < \upsilon\mu \tag{3.2}$$

where υ is a user-defined utilization target intended to limit the maximum link load. This algorithm uses the time-window estimating mechanism to derive the estimated rate of existing flows. This mechnism will be explained in subsection 3.3.2.5 later in this work. Upon admission of a new flow, the load estimate is increased using $\hat{\psi} = \psi + r_\alpha$. According to [47], in a simple $M/M/1$ queue, variance in queue length diverges as the system approaches full utilization. A measurement-based approach is doomed to fail when delay variation is very large, which will occur at very high utilization [49]. Thus, it is deemed necessary to specify a utilization target and require that the admission control algorithm works towards keeping the link utilization below this specified target level.

Equivalent Bandwidth based on Hoeffding Bounds (HB): This MBAC algorithm described in [22], computes the equivalent bandwidth for a set of flows using the Hoeffing bounds. The equivalent bandwidth for a set of flows is defined in references [22, 30] as the bandwidth $C(\epsilon)$ such that the stationary bandwidth of a set of flows exceeds this value with probability at most ϵ. The measured equivalent bandwidth based on Hoeffding bounds (C_H) of n flows, assuming peak rate policing, is:

$$C_H(\psi, \{p_i\}_{1\leq i\leq n}, \epsilon) = \psi + \sqrt{\frac{ln(1/\epsilon)\sum_{i=1}^{n}(p_i)^2}{2}} \tag{3.3}$$

where ψ is the measured average arrival rate of existing traffic and ϵ is the probability that arrival rate exceeds the link capacity. This algorithm makes use of the exponential averaging measurement mechanism (see subsection 3.3.2.5) to estimate the measured average. The admission control decision to admit a new flow's (α) requests, is then stated as:

$$C_H + p_\alpha \leq \mu. \tag{3.4}$$

Upon admission of a new flow, the load estimate is increased using $\hat{\psi} = \psi + p_\alpha$. If a flow's peak rate is unknown, it is derived from its token-bucket filter parameters (r, b) using the following equation:

$$p = r + b/U, \tag{3.5}$$

where U is a user-defined averaging period. Similarly to the algorithm in [25], if a flow is denied admission, no other flow of a similar type will be admitted until an existing one departs [48].

Acceptance Region: The MBAC algorithm proposed in [25], computes an acceptance region that maximizes the reward of utilization against the penalty of packet loss. Given link bandwidth, switch buffer space, a flow's token bucket filter parameters, the flow's burstiness, and the desired probability of actual load exceeding bounds, one can compute an acceptance region for a specific set of flow types, beyond which no more flows of those particular type should be accepted [49]. For example, let a and p be the average and peak rates of an ON/OFF source, the equivalent bandwidth (C) of the source can be computed using the following equation [24, 50] :

$$C(s) = \frac{1}{s}log\left[1 + \frac{a}{p}(e^{sp} - 1)\right], \tag{3.6}$$

where s is a space parameter, and $s > 0$. One can then draw an equivalent bandwidth curve varying the average rate on the $x - axis$ and with the resulting equivalent bandwidth on the $y-axis$. A linear bounds at different points of this curve can be composed as tangent at that point:

$$c + \alpha\psi \leq \mu \tag{3.7}$$

where c determines the location and α the slope of the tangent. This linear bound at different points can then be used as MBAC algorithms [48]. The reference [24], presented four MBACs, each based on a different tangent of the equivalent bandwidth curve. In this thesis, only the following two of the four algorithms are of interest:

1. **Tangent at Peak:** The first algorithm based on the tangent at the peak of the equivalent bandwidth curve computed from the Chernoff Bounds, admits a new flow if the following condition is true [10]:

$$n(1 - e^{-sp}) + e^{-sp}\psi \leq \mu, \tag{3.8}$$

   where n is the sum of the number of admitted flow peak rates, p is the peak rate of the flows, s is the space parameter of the Chernoff Bounds, ψ is the estimate of the current load and μ is the link bandwidth.

2. **Tangent at the Origin:** The second algorithm is based on the tangent to the equivalent bandwidth curve at the origin. Here, a new flow is admitted if the following equation is true:

$$e^{sp}\psi \leq \mu. \tag{3.9}$$

Both of the algorithms uses the point sample estimation mechanism (see subsection 3.3.2.5) to measure the current load. The measured load used in equation 3.8 and 3.9 is not artificially adjusted upon admittance of a new flow. For flows described by a token bucket filter (r,b) but not peak rate, equation 3.5 is used to derive the peak rate. If a flow is rejected, the admission control algorithm does not admitt another flow until an existing one departs.

3.3.2.5 Measurement Mechanism

In this subsection, the three measurement mechanisms[1] used by the MBAC algorithms described in the previous subsection are discussed. The measurement mechanisms are the following:

1. **Time-window (TW)**.
2. **Point Sample (PS)**.

[1]Measurement mechanism is synonymously used as the estimator

3. **Exponential Averaging (EA)**.

These measurement mechanisms may not be the most efficient nor the most rigorous measurement mechanisms. They are however, very simple, which helps to isolate the admission patterns caused by particular admission control algorithm from those caused by the measurement mechanisms itself [49].

Time-window: According to [47], a simple time-window measurement mechanism can be used to measure the network load as input to the MS algorithm. As shown in Figure 3.3, the average load is computed every S sampling period. At the end of a measurement window T, the highest average load from the just ended T is used as the load estimate for the next T window. When a new flow is admitted to the network, the estimate is increased with the parameters of the new flow admitted into the network. If a newly computed average is above the estimate, the estimate is immediately raised to the new average. At the end of every T, the estimate is adjusted to the actual load measured in the previous T. A smaller S will give a higher maximal average, which results in a more conservative admission control algorithm. A larger T keeps longer measurement history, again resulting in a more conservative admission control algorithm. To get a statistically meaningful number of sample, reference [49], suggests keeping the value of $T/S \geq 10$.

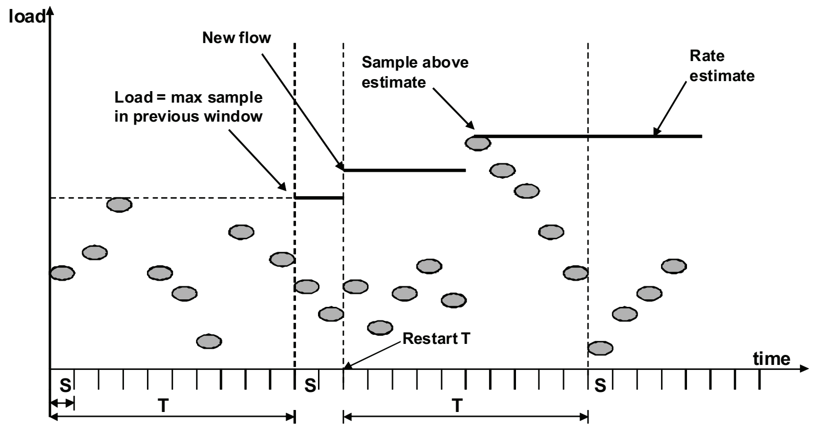


Figure 3.3: Time-window measurement mechanism

Point Sample: The measurement mechanism used with both of the acceptance region MBAC algorithms (ACTO and ACTP). It takes an average load sample every $\hat{S}$ period. Or this can be equivalently described as the time-window measurement mechanism with a T/S ratio of 1.

Exponential Averaging: An exponential average is used as input to the hoeffding bounds MBAC algorithm. The average arrival rate (ψ_S) is measured once every S sampling period [49, 47]. The average arrival rate is then computed using an infinite impulse response function with weight w:

$$\hat{\psi} = (1 - w) * \psi + w * \psi_S, \tag{3.10}$$

where w is an averaging weight which determines how fast the estimated average adapt to the new measurement. A larger w makes the averaging process more adaptive to load changes, a smaller w gives a smoother average by keeping a longer history. As stated in section 3.3, the equivalent bandwidth based admission control algorithms require peak rate policing, and derive a flow's peak rate from its token bucket parameters using equation 3.5, when the peak rate is not explicitly specified. To be on the safer side, the averaging period U in equation 3.5 should be smaller than or equal to S, the measurement sampling period. A smaller S not only makes the measurement mechanism more sensitive to burst, it also makes the peak rate derivation more conservative. A larger S may result in lower averages, however it also means that the measurement mechanism keeps a longer history because the averaging process (Eqn 3.10) is invoked less often [49].

3.4 A Multiservice Framework Using MBAC in ns-2

This section presents a proposal for a multiservice framework, which uses the four MBAC algorithms for simulating the effects of transmitting multiclass traffic flows over a packet network. In the previous section the mathematical formulations and theoretical descriptions of the four measurement-based admission control algorithms were presented. These algorithms as they are described in subsection 3.3.2.4 together with some other components (described later in this section) formed a framework already embedded in network simulator version two tool (ns-2) (detail explanations of ns-2 tool are the main topic of chapter 4 of this thesis) for simulating a single predictive[2] application traffic. In this section, this already exiting framework is described together with its components after which the new multiservice framework is described.

The already existing framework in ns-2 shown in Figure 3.4, is based on the IntServ architecture ([7]).

As mentioned previously, this framework contains the four MBAC algorithms and one PBAC algorithm for controlling the network traffic load. Basically, it is made up of the following two components:

- End-to-End signalling mechanism for requesting a new connection.

[2]Please note that predictive service and controlled-load service are used synonymly in this work.

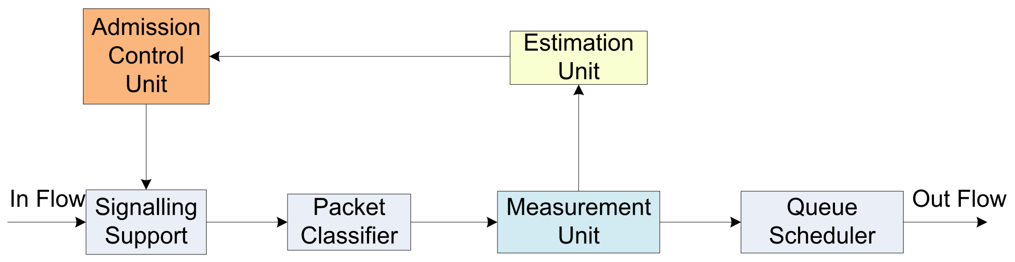


Figure 3.4: Existing single service framework in ns-2

- An enhanced link structure.

End-to-End signalling mechanism: This is a very simple end-to-end signalling protocol based on RSVP (described in subsection 2.4.1.7). It is implemented in ns-2 for requesting services. It uses a three way handshake. The signalling protocol is sender–initiated, whereby the sender sends a request message (PT_REQUEST) with the token bucket parameters (r,b). The message goes all the way to the receiver through the IntServ enhanced link (explained below).

The receiver reverses the request message as a reply message (PT_REPLY) to the sender. The sender then resends the reply message as confirm message (PT_CONFIRM) to the receiver. The reply message is needed to indicate to the sender about the successful establishment of a connection, after which it may start transmitting data packets. If the reply is a reject, then the source sends the confirm message and no more packets.

Since in this framework, the reply message traverses through a different simple-link than the request message, a confirm message is required to indicate to the signal-support component about the fate of the connection. This is particulary important for the state of those links which said “yes” to a connection but the connection got rejected on a downstream link. Finally the sender sends a tear down message (PT_TEARDOWN) at the end of the connection.

The enhanced link: The enhanced link is the medium through which data (messages) travel from source to destination. It is made of four components for supporting controlled-load service. The components are the following:

- **Signal-Support:** This component is used to maintain some transient state about flows requesting service. Specifically the link need to remember its decision for a new flow until it gets the PT_CONFIRM message.
- **Queue Scheduler:** This is a simple queue scheduler for the controlled-load service. It is implemented as a two level service priority queue.

First level high priority for controlled-load service and the second level low priority for best-effort service. In the practice, this framework simulates only controlled-load service. Also all signalling messages are explicitly prevented from drops.

- **Classifier:** The classifier in this framework is a flow classifier. It treats all the flow with flow ID greater than zero as controlled-load traffic and flows with zero flow ID as best-effort traffic[3].
- **Measurement, Estimator, AC:** The measurement object is a very simple object that measures per-class packets. The estimator estimates the used bandwidth based on the estimation algorithm in play which could be one of Time-Window, PointSample, or Exponential Average (see subsection 3.3.2.5). The admission control object (AC) makes an admission control decision based on the load estimate from the estimator and the admission control algorithm in play which could be one of Measured Sum, Hoeffding bounds, Acceptance region-Tangent at Origin, or Acceptance region-Tangent at Peak (see subsection 3.3.2.4). These are the same algorithms and estimators presented in the last section.

When a new connection request comes to a link, the signal-support module ask the admission control for a decision which will in turn look at the current load estimate and the new flow parameters. Interested readers are directed to reference [56] for full details of this framework.

Based on this existing framework, a new framework for simulating multi-controlled-load services in ns-2 is proposed. This framework is designed to support three classes of service:

- VoIP class service.
- Video class service.
- Best effort class service.

Among these service classes, the VoIP class has the highest priority, followed by the video class and the best effort class has the lowest priority. In this new framework, the logic of the end-to-end signalling mechanism component is not changed. So changes (or extensions) are only made to the components of the enhanced link component. The next subsection describes the changes made to these enhanced link components. After describing them, a new dynamic bandwidth allocation mechanism designed for the new framework is presented.

[3]This framework does not practically simulate best-effort traffic.

3.4.1 The Enhanced Link for the New Framework

This subsection decribes the enhanced link for the new proposed multiservice framework. In this framework the components of the enhanced link in the already existing framework are extended to accommodate multiservice functionalities. Figure 3.5 presents the enhanced link.

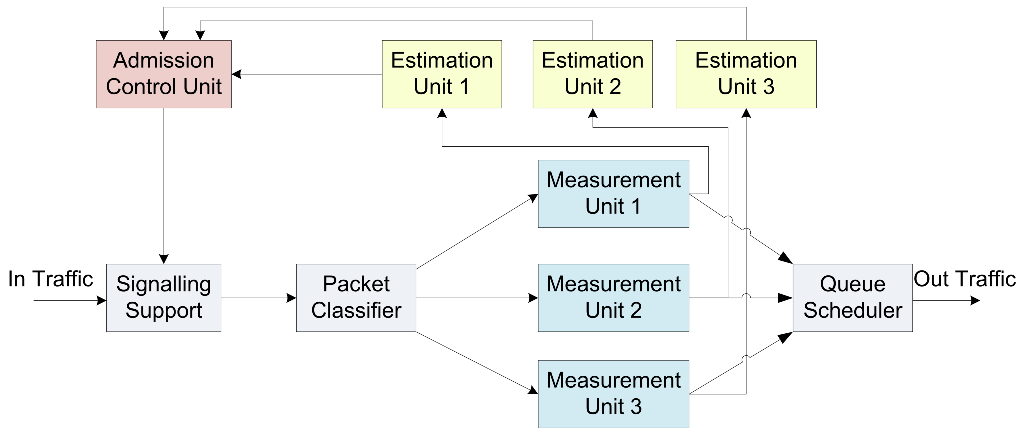


Figure 3.5: Enhanced link

The changes made to the components are described in the subsequent subsections.

3.4.1.1 Signal-Support

This component is only extended to allow traffic flows with the predefined flow IDs to place connection request to the admission control algorithm. And then maintain transient states for these flows.

3.4.1.2 Queue Scheduler

For the controlled-load multi-flow classes, a queue scheduler with three service level priority queues were implemented. The scheduler schedules the traffic flows based on the flow ID, which indicates the priority of the flow class. The low priority class has a smaller queue size, which means that they are easily dropped when there is congestion. It also explicitly prevents the signalling message packets from drops.

3.4.1.3 Classifier

In the new framework, a flow classifier is also used to distinguish the traffic flow packets into different classes. The classifier uses the flow ID to filter the traffic flows. It then groups the packets with the same flow ID into one class, thereby making it possible to treat all traffic packets of this class equally.

3.4.1.4 Measurement, Estimation, Admission Control

These three modules are been explained together because they are highly dependant on each other. The three modules are not logically changed but extended to accommodate multiclass service. The changes made to the estimation module can be better explained together with the code which will be done in the next chapter (chapter 5). The changes to the admission control algorithms are shown in the following:

- **Measured Sum:** To accomodate the multiclass, the admission decision equation (Eqn 3.2) of the measured sum algorithm is extended to become:

$$\psi_i + r_{\alpha i} < \upsilon \mu_i. \tag{3.11}$$

- **Equivalent Bandwidth based on Hoeffding Bounds:** This algorithm admission decision equation (Eqn 3.3) is extended to accommodate the multiclass services as shown below:

$$C_{Hi} + p_{\alpha i} \leq \mu_i. \tag{3.12}$$

- **Acceptance region-Tangent at Origin:** The admission decision equation (Eqn 3.9) of this algorithm is extended in the following way to accommodate the multiclass service:

$$e^{sp_i} \psi_i \leq \mu_i. \tag{3.13}$$

- **Acceptance region-Tangent at Peak:** This algorithm's admission decision equation (Eqn 3.8) is equally extended to support the multiclass services as follows:

$$n_i(1 - e^{-sp_i}) + e^{-sp_i} \psi_i \leq \mu_i. \tag{3.14}$$

where $i \in [1, m]$. For all equations i represents the class of the traffic flow and m represents the maximal number of traffic classes. The extension of these admission decision algorithms is based on the fact that the different traffic class are assigned different portion of the total bandwidth according to the class priority. So each of the admission decision algorithms has to make its decision based on the utilization of its assigned bandwidth portion.

3.4.2 Dynamic Bandwidth Allocation Mechanism

The static sharing of bandwidth among different classes of service in a network, produces under ideal conditions an acceptable utilization of the network resources. In practice, ideal conditions are far from the reality. Thus, some mechanism has to be devised to regulate the bandwidth sharing in some unusual situations so as to support high network utilisation.

The static bandwidth sharing among the traffic classes is done as follows:

- The VoIP class has highest priority and 50% of the total bandwidth.
- The video class has lower priority and 35% of the total bandwidth.
- The best effort class has lowest priority and 15% of the total bandwidth.

The problem with this type of bandwidth portioning is that, when a class bandwidth is not good utilised, the unused bandwidth is wasted. This causes poor utilisation of the total network resources. To alleviate this problem, the dynmanic bandwidth allocation mechanism is devised. This mechanism borrows bandwidth from best effort class to the higher priority classes, when they have exceeded a certain threshold of their class bandwidth and the best effort class bandwidth is uderutilised. The mechanism can be explained with the following psuedocodes:

Algorithm 3.4.1: VoIP Class Bandwidth Borrow($Mbit/s$)

```
if voipClUtil > 85% voipClBw && beClUtil < 50% beClBw
  then {
    if beClUtil < 15% beClBw
      then {voipClBw = voipClBw + 60% of 85% beClBw
    if beClUtil > 15% beClBw
      then {voipClBw = voipClBw + 60% of 50% beClBw
  }
```

Where voipClUtil = VoIP class utilization, voipClBw = VoIP class bandwidth
beClUtil = Best effort class utilization, beClBw = Best effort class bandwidth

Algorithm 3.4.1, presents the dynamic bandwidth borrowing mechanism for the VoIP traffic class. When the VoIP utilization level is over 85 % of its class bandwidth, it checks if there is unused bandwidth in the best effort class that can be borrowed. When this condition is true, it borrows 60 % of the free bandwidth from best effort class, and when it is false, it borrows nothing. There are two types of bandwidth borrowing from best effort traffic class.

- The first type is when the best effort class utilization level is below 15 % of the class bandwidth capacity. In this case 85 % of the best effort class bandwidth is made available for borrowing.
- The second type is when the best effort utilization level is above 15 % but below 50 % of the class bandwidth capacity. In this case 50 % of the best effort class bandwidth is made available for borrowing.

Note, these two bandwidth borrowing types are exclusive to one another (i.e., only one of them can be executed).

Algorithm 3.4.2: VIDEO CLASS BANDWIDTH BORROW($Mbit/s$)

```
if viClUtil > 90% viClBw && beClUtil < 50% beClBw
  then
    if beClUtil < 15% beClBw
      then viClBw = viClBw + 40% of 85% beClBw
    if beClUtil > 15% beClBw
      then viClBw = viClBw + 40% of 50% beClBw
```

Where viClUtil = Video class utilization, viClBw = Video class bandwidth
Algorithm 3.4.2, describes the dynamic bandwidth borrowing mechanism for the video traffic class. When this class utilization level is above 90 % of its class bandwidth, it checks if there is unused bandwidth in the best effort class that can be borrowed. When this condition is true, it borrows 40 % of the free bandwidth from best effort class, and when it is false, it borrows nothing.

Algorithm 3.4.3: BEST EFFORT CLASS BANDWIDTH RECOVERY($Mbit/s$)

```
if beClUtil > 10% beClBw && beClUtil < 47% beClBw
  then
    exclusiveCheck = false
    if voipClBw > voipClBw + (1/2 borrowed bw) && voipClUtil < initial voipClBw
      then
        voipClBw = voipClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw
    if viClBw > viClBw + (1/2 borrowed bw) && viClUtil < initial viClBw
      then
        viClBw = viClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw

else if beClUtil > 47% beClBw && exclusiveCheck
  then
    if voipClBw > voipClBw + (1/2 borrowed bw) && voipClUtil < initial voipClBw
      then
        voipClBw = voipClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw
    if viClBw > viClBw + (1/2 borrowed bw) && viClUtil < initial viClBw
      then
        viClBw = viClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw

if beClUtil > 70% beClBw
  then
    if voipClBw > initial voipClBw && voipClUtil < initial voipClBw
      then
        voipClBw = voipClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw
    if viClBw > initial viClBw && viClUtil < initial viClBw
      then
        viClBw = viClBw - 50% borrowed bw
        beClBw = beClBw + 50% borrowed bw
```

Where bw = Bandwidth

Algorithm 3.4.3 depicts the bandwidth recovery mechanism for the best effort traffic class. Similar to the borrowing mechanism, there are two type of recovery mechanisms - the first is carried out when 85 % of the best effort class bandwidth is borrowed, and the second type is performed when 50 % is borrowed. For each of the recovery mechanism, the recovery process is performed in two steps. The first step recovers 50 % of the borrowed bandwidth. This step is performed when the best effort utilization level is over 10 % or 47 % [4]of the class bandwidth capacity. The second bandwidth recovery step returns the rest 50 % of the borrowed bandwidth. It is carried out, when the best effort utilization level is over 70 % of the class bandwidth capacity. The bandwidth recovery is only possible if the VoIP and video classes are not using the borrowed bandwidth otherwise the bandwidth cannot be returned and the best effort class is starved.

3.5 Summary

It can be observed that supporting today's Internet service heterogeneity and integration of current emerging applications, while at the same time ensuring consistent QoS level requires enhanced service management and control mechanisms. The admission control mechanism provides a convienent means to ensure high-quality communication by safeguarding enough resources availability for customer traffic. There are different approaches to admission control mechanisms. The parameter-based approach uses pre-specified traffic characteristics to compute the network load and thus make its admission decision whether to accept or reject a new traffic flow. This approach is conceived for guaranteed services because it can ensure absolute delay bounds. It has the disadvantage of not utilising the network resources well, thereby causing poor network utilization. The measurement-based (MBAC) approach provides an alternative solution by making an on-line measurement of current network load and based on this measured load, takes its admission decision to accept or reject a new flow. The MBAC approach solves the problem of poor network utilization experienced in parameter-based admission control. This approach is conceived for relaxed and adaptive application in term of delay bounds and packet loss, because it uses traffic measurements and for the fact that measurements can't reflect the exact nature of the traffic flows, it doesn't ensure absolute bounds. There is a framework implemented in the ns-2 tool to simulate admission control mechanism for a single controlled-load service. This framework is extended to support applications with different class priorities. To support adequate resource utilization for the multi-class controlled-load service network, a dynamic bandwidth allocation mechanism is designed.

[4]Depending on the type of bandwidth borrowing performed.

4 Studying Network Performance with the Network Simulator Tool (ns-2)

This chapter presents the network simulation tool called network simulator version 2 (ns-2). The ns-2 covers a very large number of applications, protocols, network types, network elements, and traffic models. These are named *simulated objects.* This chapter has two main goals: on one hand, to introduce and explain in details the ns-2 simulator tool, and on the other hand to practically apply the knowledge gained from this tool in teaching the operations of some of the simulated objects. This helps the reader to gain more understanding of the new multiservice framework proposed in this thesis (chapter 3, section 3.4). Therefore this chapter presents not only some basics and introduction to ns-2 simulator tool, but also some lectures on OTcl programming, how to design simulation script, and how to add new class to the simulator. The chapter starts with ns-2 introduction, describing its basics and features. It goes on to discuss the sources from where ns-2 tool can be downloaded and the platform where ns-2 could be installed. Next, the lectures on ns-2 are presented. The lectures provide fundamental information and guiding steps to fast mastering of this tool. It also offers the reader practical usage examples of the tool. Finally, the chapter is summarized at the end.

4.1 Basics of ns-2

The ns-2 tool is an object-oriented discrete event simulator targeted at networking research. It is written in C++, with an OTcl (Object-oriented extension of Tcl script language) interpreter as frontend. The simulator supports a class hierarchy in C++ known as compiled hierarchy, and a similar class hierarchy within the OTcl interpreter known as interpreted hierarchy [20]. The two hierarchies are closely related to each other. From a user's perspective, there is a one-to-one correspondance between a class in the compiled hierarchy and a class within the interpreted hierarchy. The root of these hierarchies is the class *TclObject.* Figure 4.1 presents an example of the class hierarchy.

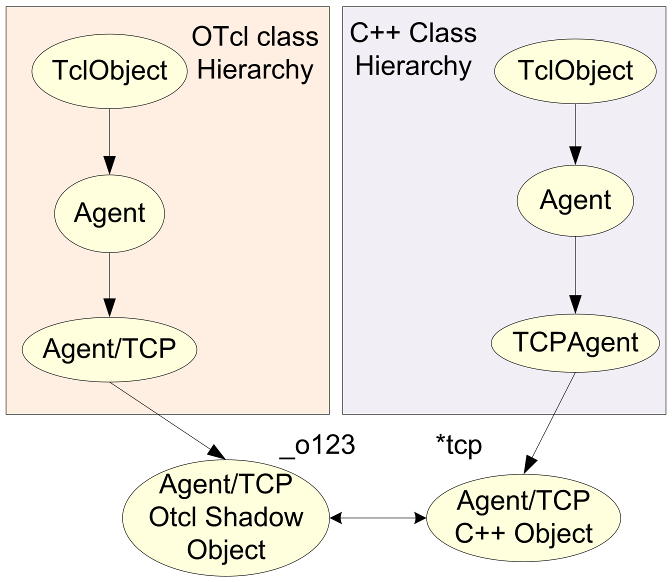


Figure 4.1: Example of class hierarchy design

A user can create new simulator objects through the interpreter, the objects are instantiated within the interpreter and are closely mirrored to the corresponding objects in the compiled hierarchy (see subsection 4.3.3.2). The interpreted class hierarchy is automatically established through methods defined in the *TclClass*. User instantiated objects are mirrored through methods defined in the *TclObject* class.

The ns tool began as a variant of the *REAL network simulator* in 1989 and has evolved substantially over the past few years. In 1995 the ns development was supported by Defense Advanced Research Project Agency (DARPA) through the Virtual InterNetwork Testbed (VINT) project at LBL, Xerox PARC, UCB, and USC/ISI. Currently ns development is supported through DARPA with Simulation Augmented by Measurement and Analysis for Networks (SAMAN) project and through National Science Foundation (NSF) with Collaborative Simulation for Education and Research (CONSER), both in collaboration with other researches. The ns tool has always included substantial contributions from other researches including wireless code from UCB Daedelus and CMU Monarch project and Sun Microsystems [62]. The ns-3 project is currently being developed with the aim of replacing the ns-2 tool.

4.1.1 Tool Concept

The ns-2 is realised with two programming languages due to the fact that it has two primary objectives: detailed simulations and network research. On the one hand, detailed simulation of protocols requires a system programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that can run over large set of data. For this task run-time speed is important and turn-around time (run simulation, fix bug, re-run) is less important.

On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important.

The ns-2 tool uses C++ and OTcl to meet these two requirements. C++ is fast to run but slower to change, making it suitable for implementation of detailed protocols and algorithms. OTcl is slower to run but can be changed quickly, making it suitable for easy varying of simulation configurations. ns-2 provides glue through *TCLCL* component to make objects and variables appear and have same values on both language spaces [20].

The ns-2 tool is a discrete even simulator, where the advance of time depends on the timing of events which is maintained by a scheduler. An event is an object in the compiled hierarchy with a unique ID, a scheduled time, and the pointer to the object that handles the event [2]. There are presently four schedulers availble in the simulator, each of which is implemented using a different data structure: a simple linked-list, heap, calender queue (used by default), and a special type called real-time [20]. The scheduler runs by selecting the next earliest event, executing it to completion, and returning to execute the next event. The unit of time used by the scheduler is seconds. At the present, the simulator is single-threaded, which means only one event can be executed at any given time. If more than one event are scheduled to execute at the same time, their execution is performed on the first-scheduled first-executed manner.

4.1.2 Target Groups, Goals, Components, and Features

The ns-2 tool is developed and designed for the following target user group [36]:

- Students interested in learning networking protocols.
- Engineers.
- Professors/teachers who are interested in illustrating TCP/IP protocol dynamics using animated examples (ns-2 and Nam combination).
- Researchers who need to evaluate their design using simulations but does not have a trusted simulation tool.
- Users of commercial simulators who are considering switching to a free simulator.

- Users of home-bred simulators who are considering switching to a more extendible simulator.

The simulator is designed for two primary goals, which will address two compelling needs in networking today. They are:

1. **Research:** For development and evaluation of protocol.
2. **Teaching:** For teaching of existing networking concept, and protocols of new networking models.

In respect of the primary goals of this tool, it presents much features and functionalities, which include the following [60]:

- Support for wired networks which includes:
  - Local area networking
  - Routing DV, LS, PIM-SM
  - Transportation protocol: TCP and UDP
  - Traffic sources: web, FTP, CBR, Telnet
  - Queueing disciplines: Drop-Tails, RED, FQ, SFQ, DRR
  - Quality of service: Integrated service architecture and differentiated service architecture.
- Support for wireless networks made up of:
  - Ad Hoc wireless networks
  - Sensor networks
  - Mobile IPv4, IPv6, UMTS, GPRS
- Support for satellite networks.
- Support for large-scale network topology generating, packet tracing, and visualising.

The ns-2 software package in a whole is made up of the following components:

- The simulator itself,
- The network animator (Nam) used to visualize the output of simulations,
- The Tcl/Tk component,
- Pre-processing component used to generate network topology and data traffic flows,
- Post-processing utilities used to analyse and process the results of the simulation such as Xgraph.

4.1.3 Cautions for ns-2

The developers and maintainers of ns-2 ([62]) gave some warnings, which anybody interested in ns-2 has to be aware of. The warnings are as quoted:

- *While we have considerable confidence in ns-2, it is not a polished and finished product, but the result of an on-going effort of research and development. In particular, bugs in the software are still being discovered and corrected.*
- *Users of ns-2 are responsible for verifying for themselves that their simulations are not invalidated by bugs. We are working to help the user with this by significantly expanding and automating the validation tests and demos.*
- *Similarly, users are responsible for verifying for themselves that their simulations are not invalidated because the model implemented in the simulator is not the model that they were expecting.*

4.2 Sources and Installation

The ns-2 tool is a free open source software package which is still currently evolving. That is, undergoing continual development, bug fixing and maintaince. There are many stable released version of the ns-2 software package.

ns-2 depends on some external components such as Tcl/Tk, OTcl, Nam, etc. These components and ns-2 itself are available for free download at the build home page ([61]). The ns-2 tool releases are in two varients:

1. The Allinone package: Made for people who want to quickly and easily try out ns-2.
2. Single packages: For people doing detail development, or people who want to save disk space or having problem with the allinone package.

ns-2 is fairly large. The allinone package requires about 320MB of disk space to build. Building ns-2 from pieces can save some disk space.

4.2.1 System Requirements and Platform

To build and install ns-2, a computer and C++ compiler are needed. The computer should at least have a minimum of 500MB free disk space. It does not place strict requirements for RAM and processor speed. But it is recommended to use a computer with a minimal of 256MB RAM and processor speed of 500MHz and above.

ns-2 is developed on several kinds of UNIX platforms like:

- FreeBSD
- Linux
- Solaris
- SunOS

It runs very smoothly on those platforms. *ns-2* is also developed to run on window platforms under cygwin for Windows 9x/2000/XP. For interested readers, detail information on running ns-2 on windows using cygwin can be found in reference [74].

4.3 Methodic Teaching with ns-2 Tool

The ns-2 tool is an asset in the teaching and learning environments. This section presents some lectures on this tool. The lectures are designed to teach the usage of ns-2, the process of simulating protocol in ns-2, and the process of extending the tool with new custom functionalities. The essence of presenting this part of the chapter in lectures is to offer the reader a systematical learning sequence and to demonstrate a possible practical application of ns-2 tool in a teaching environment.

At the end of the lectures, the reader should be in the position to master the following:

- Designing and writting simulation scripts.
- Carrying out simulation processes.
- Interpreting and post-processing simulation results.
- How ns-2 tool could be extended with custom functionalities.

4.3.1 Lecture 1: Introduction to Tcl/OTcl Programming Language

Before starting to learn how to write simulation script and run simulations with ns-2, it is a good idea to start with the learning of the Tcl/OTcl programming language which is a basic step and for the fact that this language is not so common like the C or C++ programming language. Another reason for starting with the learning of OTcl programming languages is for the fact that many simulation script examples are included in ns-2 tool, which are ready for immediate trial. So without understanding this scripting languages, it is very difficult to understand the example simulation scripts provided with the ns-2 tool. Please note that OTcl is an extension of Tcl, thus they have mostly the same buit-in commands. So references to Tcl implicitly refer to OTcls as well, but the reverse is not necessarily true.

The goals of this lecture is to teach the reader the following:

- The general syntax of Tcl programming language.
- The construct and syntax of writing procedures in Tcl.
- How Linux commands can be executed in Tcl scripts
- Object-oriented programming in OTcl programming language.

The tool command language (Tcl) is a simple scripting language created by John Ousterhout for use by scripted applications. Tcl has the following characteristics:

- It is a free package.
- It allows fast development.
- It provides a graphical interface.
- It allows easy integration with other programming languages.
- It is easy to use.

4.3.1.1 Tcl/OTcl Basics and Syntax

In Tcl, to assign a value to a variable, the command *set* is used. For example:

```
set x 2
```

This assigns to x the value of 2. This is equivalent to *x=2* in C programming language. If one wants to use the value assigned to a variable, one has to place the $ sign before the variable. For example, if one wants to assign the value of the variable x to another variable g, it can be done as follows:

```
set g $x.
```

A mathematical operation in Tcl, is executed with the expression (*expr*) command. For example, if one wants to assign to a variable d the sum of two variables u and u, one writes:

```
set d [expr $u + $v]
```

The square brackets ([ ]) return the result of the expression command. It is equivalent to a *return* statement in C functions.

Tcl variables are not typed. So a variable can be a string or an integer depending on the value one assigns to the variable. For example, assuming one wants to print out the result of the division 1/60. One may write:

```
puts [expr 1 / 60]
```

This expression outputs the value 0. To get the correct result, one has to indicate that the operands are not integer values but double values and the desired result is a double value by writing:

```
puts [expr 1.0 / 60.0]
```

The *puts* command is used to print by default to the standard output the value of its argument. If the argument is more than one, they have to be enclosed in double quote or in a brace ({}). By default, this command prints a new line after each call. This behaviour can be supressed by specifying the command option *−nonewline*.

The # sign indicates the beginning of a line of comment in a Tcl program, so Tcl intepreter will not execute this line.

4.3.1.2 File Operation and Application Command execution

Tcl provides several methods to read from and write to files on disk [79]. It provides the following command options to regulate file accessing:

- r: Open the file for reading. The file must already exist.
- r+: Open the file for reading and writing. The file must already exist.
- w: Open the file for writing. Create the file if it doesn't exist, or set the length to zero if it does exist.
- w+: Open the file for reading and writing. Create the file if it doesn't exist, or set the length to zero if it does exist.
- a: Open the file for writing. The file must already exist. Set the current location to the end of the file.
- a+: Open the file for writing. The file does not exist, create it. Set the current location to the end of the file.

To create a file, one has to give a name (i.e. *filename*) and assign the file descriptor to a variable (pointer), that can be used in Tcl program to access the file. For example:

```
set fileptr [open filename w]
```

The file is opened for writing and the file descriptor is assigned to the variable *fileptr*. To write something into the file, one could do the following:

```
puts $fileptr 4
```

This will write the value 4 into the file. After using a file, it is recommended to close the file so that the operating system releases the resource consumed by this file. File closing is done with the *close* command. For example:

```
close $fileptr
```

The execution of an application command within a Tcl script can be done with the *exec* command, passing to it the name of the application's command and its options. For example, one might want to plot the result of a simulation dumped in a file named *data* using the xgraph application. The calling statement looks like:

```
exec xgraph data &
```

The & sign on the argument list, tells the Tcl intepreter to run the xgraph application in the background.

4.3.1.3 Tcl Control Structures

Like most programming languages, Tcl supports control structures like *if, for, while, switch,* and so on. For example, the structure of an *if* command is as follows:

```
if {expression} {
    <execute some commands>
  }else {
    <execute some other commands>
}
```

The *if* command can be nested with other *ifs* and their corresponding *else* counterpart. The test expression returns a string *yes/no* or *true/false*, the case of the return value is not checked, i.e., *True/FALSE* or *YeS/nO* are legitimate returns. If the test expression returns *true* the <*execute some commands*> body is executed, otherwise the <*execute some other commands*> body is executed. It should be noted that when testing for equality, one should use " ==″ sign like in C and not the sign " =″. Inequality is also tested with "! =″ as in C.

The *for* loop in Tcl has an iterated construct similar to the *for* loop in C. The *for* command in Tcl takes four arguments; an initialization, a test, an increment, and the body of code to evaluate on each pass through the loop. An example of the *for* command is:

```
for {set i 0} {$i < 10} {incr i} {
    <execute some code>
}
```

During evaluation of the *for* command, the initialization code is evaluated once, before any other arguments are evaluated. After the initialization code has been evaluated, the test is evaluated. If the test evaluates to true, then the body is evaluated, and finally, the increment argument is evaluated. After evaluating the increment argument, the interpreter loops back to the test, and repeats the process. If the test evaluates to false, then the loop exits immediately. Details of the other control structures can be found in reference [79].

4.3.1.4 Adding New Commands to Tcl

The Tcl commands are equivalent to functions in C language. In Tcl, functions are known as procedures and have the abbreviation *proc*. There is a list of built-in commands that the intepreter loads when it starts up. One can also create some commands that the intepreter should execute using the *proc* command. The *proc* command creates a new command in Tcl. The syntax for the command is:

```
proc name args body
```

When *proc* is evaluated, it creates a new command with name *name* that takes arguments *args*. When the procedure name is called, it then runs the code contained in *body*. For example, one can define the following new command *sum* using the *proc* command as follows:

```
proc sum {arg1 arg2} {
    set x [expr {$arg1 + $arg2}];
    return $x
}
```

The procedure can have parameters that could be files, objects, or variables. In this example the *sum* procedure takes two parameters *arg1* and *arg2* and returns a value. These parameters are used as named within the body of the procedure to do an addition and the result of the addition is returned as the sum. The procedure can be invoked as:

```
sum a b
```

where *a* and *b* represent the two parameters declared.

To demonstrate all of the above explanations, a command named *test* is written as follows:

```
proc test {a b} {
  set c [expr $a + $b]
  set d  [expr [expr $a - $b] * $c]
  puts "c = $c  d = $d"

  for{set k 0} {k < 10} {incr k} {
    if{$k < 5} {
       puts "k < 5, pow = [expr pow($d, $k)]"
       } else {
          puts "k > 5, mod = [expr $d % $k]"
        }
    }
}
```

The command *test* takes two integer values as parameter, does some arithmetic operations in the body of the procedure and in one case in the loop, power operations are executed and the result printed to the standard output, and in the other case, modula operations are executed and the result printed to the standard output. The command can be invoked for example, as:

```
test 43 27
```

4.3.1.5 Object-Oriented Programming in OTcl

In this part of the lecture, it is assumed that the reader is already familiar with one of the object-oriented programming languages like Java, C++ and so on. For simplicity, the lecture omits many details of the OTcl programming language. The details can be found in reference [63].

An experienced C++ programmer may feel uncomfortable writing object-oriented programing in OTcl at first contact. So to help the C++ programmer to quickly get used to the OTcl language, some important characteristics of OTcl and their equivalence in C++ are presented below:

- Instead of a single class declaration in C++, multiple class definitions can be done in OTcl. Methods are referred to as an instance procedure with abbreviation *instproc*[1]. Each definition of a method with *instproc* command adds a method to a class. Instance variable is defined with the *set* command but within the body of a method, with the *instvar* command. Each instance variable definition adds an instance variable to an object.

- Instead of a constructor in C++, write an *init* instproc in OTcl. Instead of a destructor in C++, write a *destroy* instproc in OTcl. Unlike constructors and destructors, init and destroy instruction procedures do not combine with base classes automatically. They should be explicitly combined with the help of the *next* command.

- Unlike C++, OTcl methods are always called through the object. The name *self*, which is equivalent to *this* in C++, may be used inside method bodies. Unlike C++, OTcl methods are always virtual.

[1]Note that this method is only used in object-oriented programming context in OTcl. This is a difference between Tcl and OTcl.

- Instead of calling shadowed methods by naming the method explicitly as in C++, call them with *next* command. *next* searches further up the inheritance graph to find shadowed methods automatically. It allows methods to be combined without naming dependencies.
- Avoid using static methods and variables, since there is no exact analogue in OTcl. Place shared variables on the class object and access them from methods by using *$class*. This behavior is inherited. If inheritance is not needed, use *proc* methods on the class object.
- The word *-superclass* is used to declare the inheritance of a class from another.

The following example demonstrate object-oriented programming in OTcl:

```
 Class mom
mom instproc init {age} {
        $self instvar age_
        set age_ $age
}

mom instproc greet {} {
        $self instvar age_
        puts   $age_ years old mom: How are you doing?
}

Class kid -superclass mom
kid instproc greet {} {
        $self instvar age_
        puts   $age_ years old kid:  What s up, dude?
}

set a [new mom 45]
set b [new kid 15]
```

In the example a class *mom* is defined with a constructor, which accepts one parameter *age*, and a method *greet*, which does not accept any parameter but prints out a greetings for a *mom*'s object. A second class *kid* is defined to inherit from the base class *mom*. The *kid* class inherits the constructor from the base class and defines its own version of the *greet* method to print out greetings for its own objects. At the end in the example, the two classes are instantiated. The execution of this example outputs the following results:

```
 45 years old mom: How are you doing?
 15 years old kid: What's up, dude?
```

4.3.1.6 Lecture Assignments

At the end of this lecture the reader is recommended to do the following assignments which will help the reader to practically assess his/her understanding of the lecture content:

- Write a procedure that takes three integer values as parameter and outputs the square root of the highest value.
- Write a base class *hat* that has a constructor to set the colour of a hat. Write extra three classes to inherit from this class and let them print out the colour of their hats.

4.3.2 Lecture 2: Writing Simulation Scripts and Simulating in ns-2

This lecture describes the structure of a simulation script, the simulation process in ns-2, and how to post-process the results of simulations.

The goal of this lecture is to teach the reader the following points:

- How to setup a simulation script.
- How to create a network topology.
- How to create traffic sources.
- How to trace traffic packets.
- How to run simulations.
- How to setup and use the network animator tool (Nam)
- The post-processing of simulation results.

The simulation script in ns-2 tool has the following generic structure:

- Create simulator object.
- Setup for tracing by opening files where the trace data are written. The step can be optional.
- Create topology, which includes setting up nodes and creating links to link them up.
- Create traffic agents.
- Create applications or traffic sources to run over the traffic agents.
- Post-process procedures.
- Start simulation.

These steps are grouped together and are described in the subsequent subsections.

4.3.2.1 Initialization and Trace Setup

An ns-2 simulation script starts with the creation of the simulator object ([27]), which is done with the following statement:

```
set ns  [new Simulator]
```

This should be the first statement in the Tcl script after maybe some comments. The line declares a new variable *ns* using the *set* command. One can name this variable as one wishes but in general, it is declared as *ns* because it holds an instance of the *Simulator* class, thus an object. The code *[new Simulator]* is indeed the instantiation of the *Simulator* class using the reserved word *new*. So with the declared variable *ns*, all the methods of the *Simulator* class can be used. When a new *Simulator* object is created in tcl, the initialization procedure performs the following three operations [20]:

1. Initialize the packet format: This sets up field offsets within packets used by the entire simulation.

2. Create a scheduler: The scheduler runs the simulation in an event-driven manner and maybe replaced by an alternative scheduler, which provides somewhat different semantics. The default scheduler is the calender scheduler.

3. Create a null agent: This agent is generally useful as a sink for dropped packets or as a destination for packets that are not counted or recorded.

There are several approaches to collecting simulation results. In this lecture, the tracing approach is applied. Tracing can be done in two formats: the general trace format and the Nam trace format. They can be setup as follows:

```
1 set tracefile [open out.tr w]
2 $ns trace-all $tracefile
3
4 set namfile [open out.nam w]
5 $ns namtrace-all $namfile
```

The above setups create a data trace file called *out.tr* and a Nam visualization trace file named *out.nam*. Within the Tcl script, these files are not explicitly referred to by their names, instead by the pointers declared to hold their file descriptors. In the setup, the pointers are *tracefile* and *namfile*. In the code, at line one a file is opened for writing trace data, at line two the simulator member method *trace-all*, which accepts a file pointer as parameter, is set up for recording the simulation results in the general trace format. At line four, a file is opened for writing Nam trace data, at line five the simulator member method *namtrace-all*, which accepts a file pointer as parameter, sets up for record the simulation result in Nam input format. The Nam data are written by the execution of *flush-trace* member method (This method is explained later).

4.3.2.2 Create Topology and Nam Setup

Network topologies in ns-2 are made up of nodes and link. A node can be created with the statement:

```
set n0 [$ns node]
```

The statement instantiates a node and assigns it to the declared variable *n0*. The instance procedure *node* constructs a node out of more simple classifier objects. All nodes has the following components:

- an address or **id_**, monotonically increasing by one across the simulation namespace as more nodes are created. The initial value is zero,
- a list of neighbors (**neighbor_**),
- a list of agents (**agent_**),
- a node type identifier (**nodetype_**), and
- a routing module.

After creating nodes, they have to be linked up so that packets can travel from one node to the other. There are two link types in ns-2. They are *simplex-link* and *duplex-link*. The *simplex-link* is the basic link and it connects only in one direction. The *duplex-link* is bi-directional and it is made up of two *simplex-link*s. The link is a compound object containing some basic objects and could be extended by inserting more objects. The link has the following make ups:

```
simplex-link <node0> <node1> <bandwidth> <delay> <queue-type>
```

In this basic form, the link is made up of a source node (*node0*) to be connected to a destination node (*node1*), a link capacity (*bandwidth*), a propagation delay (*delay*), and a queue type (*queue-type*). The queueing type can be fair queueing, stochastic fair queueing, droptail, or a custom queueing type defined by the user. These queueing types provide different mechanism for handling buffer overflows. For example, two nodes can be connected as follows:

```
$ns duplex-link $n0 $n1 10Mb 1ms DropTail
```

This statement means that two nodes *$n0* and *$n1* are connected to communicate bi-directionally with a link capacity of *10 Mbit*, a propagation delay of *1 ms*, and a *DropTail* queueing type. The droptail queueing handles buffer overflows at the output queue by dropping the last arriving packets. The buffer size can be defined with the following code:

```
$ns queue-limit $n0 $n1 30
```

This states that the link between node *$n0* and node *$n1* has a queue buffer of 30 packets. i.e., it can only hold 30 packets pending transmission, after which subsequent ones are dropped. If this limit is not set, it takes the default value of 50[2]. Figure 4.2 shows the internal structures of a *simplex-link*. A queue overflow is implemented by sending dropped packets to the *Null Agent*. The time to live (TTL) object computes the time each received packet has to live

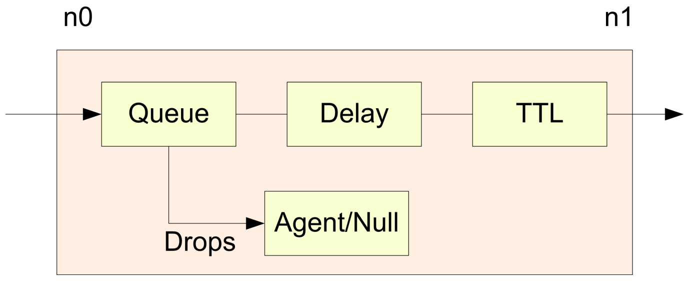


Figure 4.2: A simplex link

The Nam tool has a graphical interface for displaying the network topology. It can be configured to give the nodes in the topology certain positions. For two nodes in a topology, this can be done with the statements:

```
$ns duplex-link-op $n0 $n1 orient right-down
```

This statement means to position the two nodes *$n0* and *$n1* so that *$n1* should be in down right side of *$n0* position. More details of this command and other configuration parameters for positioning nodes in a topology can be found in the ns-2 manual ([20]).

After discussing how nodes and links are created, they can be put together to create a network topology. For example, a network topology with four nodes can be created with the following code:

```
#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#Create link between the nodes
$ns duplex-link $n0 $n2 10Mb 1ms DropTail
$ns duplex-link $n1 $n2 10Mb 1ms DropTail
$ns duplex-link $n2 $n3 5Mb 1ms DropTail
```

[2]The default value is set in ns-default.tcl file, which contains default values for the ns-2 tool.

4.3.2.3 Create Transport Agents and Application Sources

Now the basic network setup is done. The next thing is to make traffic packets flow through the network by creating traffic agents such as TCP and UDP, and traffic sources such as FTP and CBR, and attach them to the nodes and agents.

Agents represent endpoints where network-layer packets are constructed or consumed, and are used in the implementation of protocols at various layer. The **class Agent** has an implementation partly in OTcl and partly in C++. The C++ **class Agent** includes enough internal states that can be assigned to the different fields of a simulated packets before it is sent [20]. These states are the following:

- **addr_**: Node address of myself (source address of a packet).
- **dst_**: Destination address of a packet.
- **size_**: The size of the packet in bytes placed in the common header.
- **type_**: Type of packet in the common header.
- **fid_**: The IP flow identifier.
- **prio_**: The IP priority field.
- **flags_**: Packet flags.
- **defttl_**: Default IP TTL value.

These state variables maybe modified by any class derived from **Agent**, although not all of them maybe needed by any particular agent. There are several agents supported in the ns-2 simulator such as TCP, UDP, TCPSink, and so on (see ns-2 manuel [20] for full list).

Applications in ns-2, sit on top of transport agents. There are two basic types of applications: Traffic generator and simulated applications. Traffic generator objects generate traffic and can be of four types namely: exponential, pareto, CBR, and traffic trace.

Application/Traffic/Exponential Objects: Exponential traffic objects generate On/Off traffic. During "on" periods, packets are generated at a constant burst rate. During "off" period no packet is generated. Burst times and idle times are taken from exponential distributions. Configuration parameters are:

- **PacketSize_**: constant size of packets generated.
- **burst_time_**: average on time for generator.

- **idle_time_**: average off time for generator.
- **rate_**: sending rate during on time.

Application/Traffic/Pareto Objects: Pareto objects generate On/Off traffic packets with burst times and idle times taken from pareto distributions. Configuration parameters are:

- **PacketSize_**: constant size of packets generated.
- **burst_time_**: average on time for generator.
- **idle_time_**: average off time for generator.
- **rate_**: sending rate during on time.
- **shape_**: the shape parameter used by pareto distribution.

Application/Traffic/CBR Objects: CBR objects generate packets at constant bit rate. **$cbr start** causes the source to start generating packets and **$cbr stop** causes the source to stop generating packets. Configuration parameters are:

- **PacketSize_**: constant size of packets generated.
- **rate_** sending rate.
- **interval_**: (optional) interval between packets.
- **random_**: Whether or not to introduce random noise in the scheduled departure times. Default is off.
- **maxpkts_**: Maximum number of packets to send.

Application/Traffic/Trace Objects: Trace objects are used to generate traffic from a trace file. For example, **$trace attach-tracefile tfile** attaches the *Tracefile* object *tfile* to this trace. The *Tracefile* object specifies the trace file from which traffic data is to be read. Multiple *Application/Traffic/Trace* objects can be attached to the same *Tracefile* object. A random start place with the *Tracefile* is chosen for each *Application/Traffic/Trace* object. There is no configuration parameter for this object.

Tracefile objects are used to specify the trace file that is used for generating traffic. **$tracefile** is an instance of the Tracefile object. **$tracefile filename <file-input>** sets the file name from which the traffic trace data is to be read to *trace-input*. There are no configuration parameters for this object. A trace file consists of any number of fixed length records. Each record consist of two 32 bit fields. The first indicates the interval until the next packet is generated in microseconds. The second indicates the length of the next packet in bytes.

A simulated application object can be of two types, Telnet and FTP.

Application/Telnet: TELNET objects produce individual packets with inter-arrival times as follows. If interval_ is non-zero, then inter-arrival times are chosen from an exponential distribution with average interval_. If interval_ is zero, then inter-arrival times are chosen using the "tcplib" telnet distribution. **$telnet start** causes the *Application/Telnet* object to start producing packets. **$telnet stop** causes the *Application/Telnet* object to stop producing packets. **$telnet attach <agent>** attaches a *Telnet* object to agent. Configuration parameters are:

- **interval_**: The average inter-arrival time in seconds for packets generated by the Telnet object.

Application/FTP: FTP objects produce bulk data for a TCP object to send. **$ftp start** causes the source to produce maxpkts_ packets. **$ftp produce <n>** causes the FTP object to produce *n* packets instantaneously **$ftp stop** causes the attached TCP object to stop sending data. **$ftp attach agent** attaches an *Application/FTP* object to agent. **$ftp producemore <count>** causes the *Application/FTP* object to produce *count* more packets. Configuration parameters are:

- **maxpkts_**: The maximum number of packets generated by the source.

For example, the following OTcl code creates a TCP agent, which transmits FTP data traffic flows, and a UDP agent, which sents CBR data traffic flows:

```
#Setup TCP Connection
set tcp [new Agent/TCP]          ;#create sender agent
$tcp set fid_ 2                  ;#set IP-layer flow ID
$ns attach-agent $n0 $tcp        ;#put sender on node n0
set sink [new Agent/TCPSink]     ;#create receiver agent
$ns attach-agent $n3 sink        ;#put receiver on node n3
$ns connect $tcp $sink           ;#establish TCP connection

#Setup an FTP over TCP connection
set ftp [new Application/FTP] ;#create an FTP source application
$ftp attach-agent $tcp   ;#associate FTP with the TCP sender
$ftp set type_ FTP       ;#set the traffic type to FTP

#Setup UDP Connection
set udp [new Agent/UDP]          ;#create sender agent
$udp set fid_ 2                  ;#set IP-layer flow ID
$ns attach-agent $n1 $udp        ;#put sender on node n1
set null [new Agent/Null]        ;#create receiver agent
```

```
$ns attach-agent $n3 $null      ;#put receiver on node n3
$ns connect $udp $null          ;#establish UDP connection

#Setup a CBR over UDP Connection
set cbr [new Application/Traffic/CBR] ;#create a CBR source application
$cbr attach-agent $udp  ;#associate CBR with the UDP sender
$cbr set type_ CBR      ;#set the traffic type to CBR
$cbr set packet_size_ 1000 ;#set the packet size
$cbr set rate_ 1mb          ;#set the packet rate
```

4.3.2.4 Simulation Process and Termination

Before describing how the simulation is started, a description of the termination procedure is necessary. This procedure is scheduled to stop the simulation at the appropriate time. The termination procedure contains commands such as those to write trace data into an already opened trace file, or to start the Nam application to visualize the simulation, or close opened file descriptors. The most important command in this procedure is the *exit* command which terminates the simulation process. The following code shows an exmaple of a termination procedure:

```
proc finish {} {
 global ns tracefile namfile
 $ns flush-trace
 close $tracefile
 close $namfile
 # Execute the name tool
 exec nam out.nam &
 exit 0
}
```

The word *global* is used to access variables declared outside the procedure. The command *flush-trace* is responsible for writting the Nam trace data into the Nam trace file. The *close* command closes the opened file descriptors. The *exec* command is used to start the Nam application which visualizes the simulation results gathered in Nam trace file. The *exit* command terminates the simulation.

The ns-2 tool is a discrete event-driven simulator. The events on the simulation script are scheduled to execute at a certain point in time. The event scheduler is created when the simulator object is instantiated (see subsection 4.3.2.1). Events are scheduled using the format:

```
$ns at <time> <event>
```

The time unit is in seconds. For example, the FTP and CBR applications examples described in the previous subsection (4.3.2.3) could be schduled with starting and stopping times, as shown by the following piece of code:

```
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"
```

At this point, all the necessary steps in writing a simulation script have been learned. So the simulation process can be started with the following command:

```
$ns run
```

Combining all the steps learned so far, one can easily write a simulation script for wired packet network. For example the following code:

```
#Create a simulator object
set ns [new Simulator]

#Define different colours for data flows (for Nam)
$ns color 1 Blue
$ns color 2 Red

#Open a general trace file
set tracefile [open out.tr w]
$ns trace-all $tracefile

#Open the Nam trace file
set namfile [open out.nam w]
$ns namtrace-all $namfile

#Define a "finish" procedure
 proc finish {} {
    global ns tracefile namfile
    $ns flush-trace
    close $tracefile
    close $namfile
    # Execute the name tool
    exec nam out.nam &
    exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#Create link between the nodes
$ns duplex-link $n0 $n2 10Mb 1ms DropTail
$ns duplex-link $n1 $n2 10Mb 1ms DropTail
$ns duplex-link $n2 $n3 5Mb 1ms DropTail

#Set queue size of link (n2-n3) to 20
$ns queue-limit $n2 $n3 20

#Give node position (for Nam)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3) (for Nam)
$ns duplex-link-op $n2 $n3 queuePos 0.5

#Setup TCP Connection
set tcp [new Agent/TCP]
$tcp set fid_ 1
$tcp set class_ 2   ;#mark the traffic with red color
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink

#Setup an FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

#Setup UDP Connection
set udp [new Agent/UDP]
$udp set fid_ 2
$udp set class_ 1  ;#mark the traffic with blue color
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null

#Setup a CBR over UDP Connection
```

```
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb


#Schedule events for CBR and FTP applications
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"

#call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#start the simulation process
$ns run
```

This code can be copied into a file (e.g. exam.tcl) and run it with the ns-2 tool. The code statements for setting packet colours are optional and have no effect on the actual simulation [12].

4.3.2.5 Nam Visualization and Simulation Result Post-Processing

After simulating a network protocol, the simulation process can be virtualized by the network animator tool. Nam is a Tcl/TK based animation tool for viewing network simulation traces and real world packet trace data [65]. The first step to use Nam is to produce the trace file. The trace file should contain topology information, e.g., nodes, links, as well as packet traces. Usually, the trace file is generated by ns-2 during a simulation process.

When the trace file is generated, it is ready to be animated by Nam. Upon startup, Nam will read the trace file, create topology, pop up a window, do layout settings if necessary, then pause at the time of the first packet in the trace file. Through its user interface (Figure 4.3[3]), Nam provides control over many aspects of animation.
If the example simulation script described in the last subsection (4.3.2.4) is runned, the Nam tool will display a four node network as shown in Figure 4.4.
The position of the nodes in Figure 4.4 could have been chosen at random, if the following piece of code were not added to the simulation script:

```
#Give node position (for Nam)
```

[3]This graphic is extracted from reference [27]

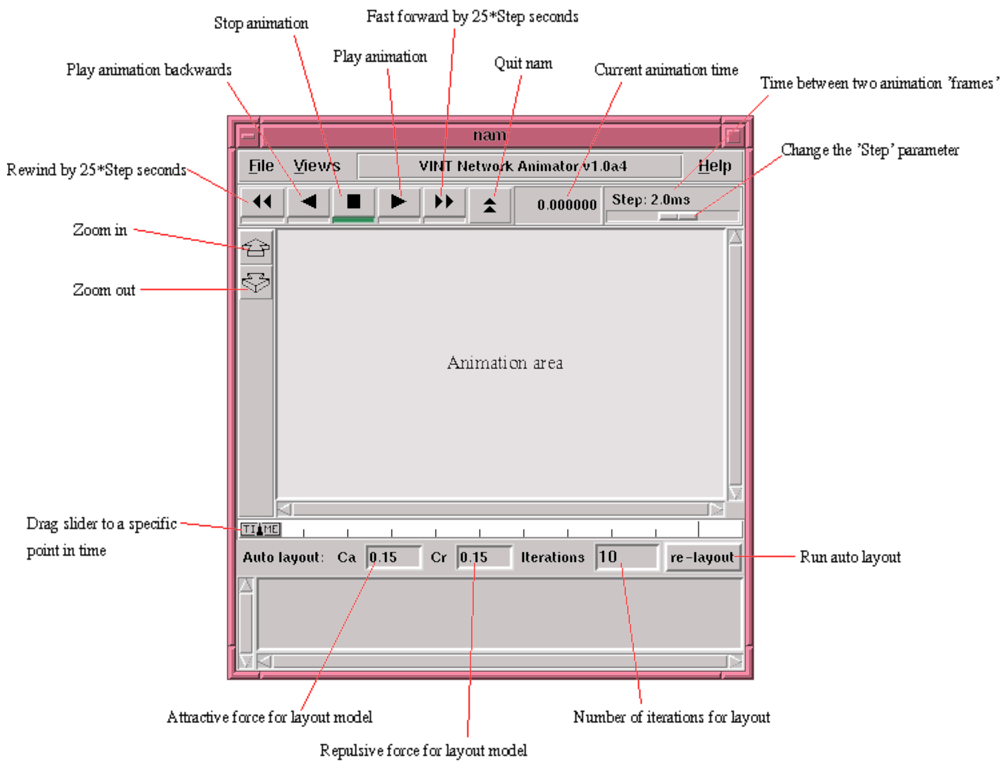

Stop animation
Fast forward by 25*Step seconds
Play animation
Quit nam
Current animation time
Play animation backwards
Time between two animation 'frames'
Change the 'Step' parameter
Rewind by 25*Step seconds
Zoom in
Zoom out
Animation area
Drag slider to a specific point in time
Run auto layout
Attractive force for layout model
Repulsive force for layout model
Number of iterations for layout
nam
File
Views
VINT Network Animator v1.0a4
Help
0.000000
Step: 2.0ms
Auto layout:
Ca
0.15
Cr
0.15
Iterations
10
re-layout

Figure 4.3: Nam user interface

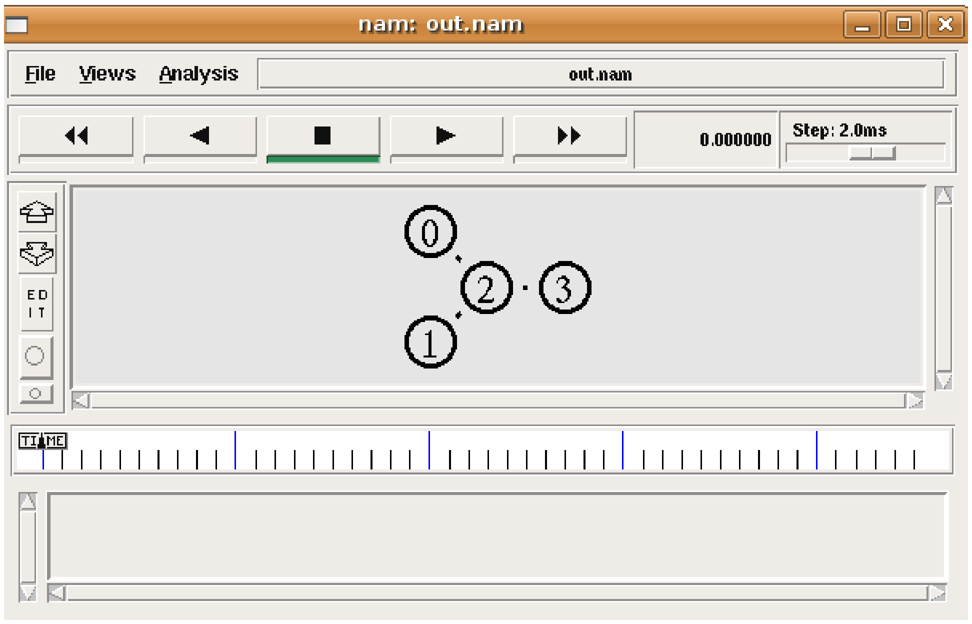


Figure 4.4: Nam output of simulated example script

```
$ns duplex−link−op $n0 $n2 orient right−down
$ns duplex−link−op $n1 $n2 orient right−up
$ns duplex−link−op $n2 $n3 orient right
```

Nevertheless, if a script is wrote to choose random positions for nodes, when the position is not satisfactory, one can press the "re-layout" button (see Figure 4.3) which makes the tool choose another random position. If the positioning code is writting, this button will be unavailable. In the Nam display, the CBR packets flowing from node *$n1* to node *$3* are coloured blue and the TCP packets flowing from node *$n0* to node *$n3* are coloured red. Also the TCP acknowledgement packets from node *$n3* to node *$n0* are coloured red but they are smaller in size. The colour is obtained with the following code in the simulation script:

```
$ns color 1 Blue
$ns color 2 Red

$udp set class_ 1   ;#mark the traffic with blue color
$tcp set class_ 2    ;#mark the traffic with red color
```

If a Nam trace file already exist, one does not need to run ns-2 in order to display the simulation on the Nam graphical interface but instead run the command *nam <file name>* from the directory containing the Nam trace file. There are some other things that can be configured for Nam like:

- **Colouring node**: For example, if node *$n0* is to appear in red, one could write: *$n0 color red*
- **Shape of node**: By default they are round, but can appear differently. For example, to have a node with the shape of a box, one can write: *$n1 shape box*
- **Colouring link**: Links can be given different colours. For example, to colour a link green, one can write: *$ns duplex-link-op $n0 $n1 color green*

The general trace file registers all the events that occur in the network. If tracing is set up, ns-2 inserts four objects in the link: **EnqT**, **DeqT**, **RecvT**, and **DrpT** as shown in Figure 4.5.
EnqT registers information concerning a packet that arrives and is queued at the input queue of the link. If the packet overflows, then information concerning the dropped packets is handled by **DrpT**. **DeqT** registers information at the instance the packet is queued. Lastly, **RecvT** records information about packets that have been received at the output of the link.

Tracing all the events in the network, creates a large trace trace file, which occupies much disk space and is difficult to process. There is a means of tracing only subset of events in the network, this could be achieved by replacing the command

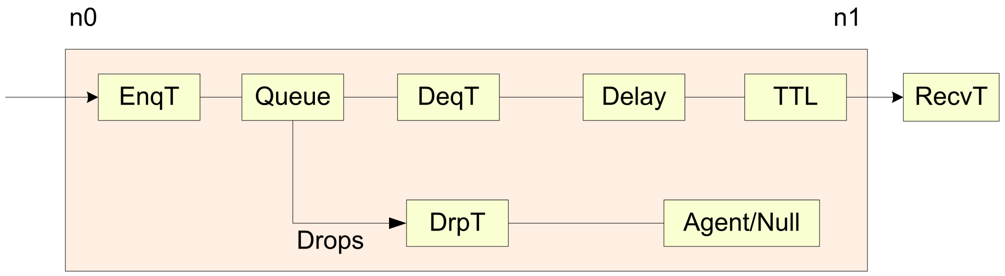


Figure 4.5: Tracing objects in simplex-link

$ns trace-all <filename>* with the command *$ns trace-queue. For example, one can type:

```
$ns trace−queue $n2 $n3 $filedescriptor
```

which results in an output trace file that contains only events that occurred over the link between node *$n2* and node *$n3*. There is a similar command *namtrace-queue* that has the same effect for tracing in Nam input format. The command *trace-queue* should ofcourse appear after the definition of the link in the simulation script.

The trace file entries are organised in twelve fields as shown in Figure 4.6.

Event	Time	From Node	To Node	Pkt Type	Pkt Size	Flags	Fid	Src Addr	Dst Addr	Seq Num	Pkt ID

Figure 4.6: Trace entry fields

The fields have the following meaning:

1. The first field represents the event type, which can be one of four possible symbols $r, +, -, d$ meaning *receive* (at the output of the link), *enque*, *deque*, and *drop* respectively.

2. The second field is the time at which event occurs.

3. The Third field is the input node from which event occurs.

4. The fourth field is the output node to which event occurs.

5. The fifth field is the packet type. e.g. TCP, UDP, CBR

6. The sixth field is the packet size.

7. The seventh field is a set of packet flags.

8. The eighth field is the IP flow ID.

9. The ninth field is the source address and port number given in the form "node.port".
10. The tenth field is the destination address and port number given in the form "node.port".
11. The eleventh field is the network layer protocol's sequence number. Even though UDP implementation in a real network does not use sequence number, ns-2 keeps track of UDP packet sequence number for analysis purposes.
12. The twelfth field is the packets unique ID.

The following snipet from the trace file produced by the example simulation script, shows an example of entries in a trace file:

```
r 1.002096 2 3 tcp 40 − − − − − − − − 2 0.0 3.0 0 113
+ 1.002096 3 2 ack 40 − − − − − − − − 2 3.0 0.0 0 114
− 1.002096 3 2 ack 40 − − − − − − − − 2 3.0 0.0 0 114
r 1.00316 3 2 ack 40 − − − − − − − − 2 3.0 0.0 0 114
+ 1.00316 2 0 ack 40 − − − − − − − − 2 3.0 0.0 0 114
− 1.00316 2 0 ack 40 − − − − − − − − 2 3.0 0.0 0 114
+ 1.004 1 2 cbr 1000 − − − − − − − − 1 1.0 3.1 113 115
− 1.004 1 2 cbr 1000 − − − − − − − − 1 1.0 3.1 113 115
r 1.004192 2 0 ack 40 − − − − − − − − 2 3.0 0.0 0 114
+ 1.004192 0 2 tcp 1040 − − − − − − − − 2 0.0 3.0 1 116
− 1.004192 0 2 tcp 1040 − − − − − − − − 2 0.0 3.0 1 116
+ 1.004192 0 2 tcp 1040 − − − − − − − − 2 0.0 3.0 2 117
− 1.005024 0 2 tcp 1040 − − − − − − − − 2 0.0 3.0 2 117
r 1.0058 1 2 cbr 1000 − − − − − − − − 1 1.0 3.1 113 115
```

There are fourteen trace entries in the snipet. Four receive events (indicated by r), five enque events (indicated by +), and five deque events (indicated by -). The trace file can be processed by several utilities such as awk scripts, perl script, etc., to extract information for calculating factors like jitter, delay, or for plotting purposes using xgraph or gnuplot.

As an example, one can calculate the number of packets received from a node with flow ID 1 from the trace file, using the following awk script:

```
BEGIN {
# Initialization. Set a variable to count the
#number of packets received with flow id 1
        numPack = 0;
}
{
   action = $1;
   flow_id = $8;
```

```
        if (action == "r" && flow_id==1)
                numPack++;
}
END {

    printf("number of packets received with flow id 1 is: %d\n", numPack);
}
```

if the above awk script code is saved in a file e.g., *pktcount.awk* and the trace file was saved in a file named *out.tr*, then the awk script can be called to post-process the trace file as follows:

```
awk -f pktcount.awk out.tr
```

The *-f* option tells the awk interpreter, that the script code to execute is saved in a file. Interested readers are referred to the awk user guide ([31]) for details about the interpreter and its command options.

The ns-2 tool allows also other means of getting information from simulated networks such as queue monitoring and flow monitoring mechanisms. Interested readers are referred to ns-2 manuel ([20]) for detailed information about these mechanisms.

4.3.2.6 Lecture Assignments

At the end of this lecture, the reader is recommended to do the following assignments.

- Rewrite the example simulation script to use exponential traffic source over UDP instead of CBR. Visualize the simulation with the Nam tool.
- Extend the example simulation script with two nodes. Detach the *sink* and *null* agents from node *n3*, attach each of the agents to one of the two new nodes. The connections should not be changed. Visualize the simulation with the Nam tool.

4.3.3 Lecture 3: Adding Custom Functionality to ns-2

The third lecture in this series, gives insight of the directory structures of the ns-2 software package at a whole, which includes the external components such as Xgraph, Nam, Tcl, and so on. It also describes how C++ and OTcl class hierarchies can be linked, and how to access a C++ member variable from OTcl class hierarchy. An illustrative example of adding a new class object to the ns-2 software package is presented.

The aim of this lecture is to give the reader some impressions of:

- How the source code of ns-2 is structured.

- Where to find which source file.
- How a new class object can be added to the ns-2 software.
- The linkage of C++ and OTcl class hierarchies.
- How to access C++ member variable from OTcl class hierarchy.

This lecture is spread across the following subsections.

4.3.3.1 ns-2 Directory Structure Overview

The ns-2 tool is a big software package. So for a new user or developer to easily find the source code of a particular protocol, he/she must have a concept of how the source code is organised in the package. Also the fact that ns-2 is written in two programming languages, makes it more difficult to locate the implementation files of a protocol.

Figure 4.7 presents an overview of a part of the important directories of ns-2 software package. It is assumed that the user installed the *ns-allinone-2.33* package.

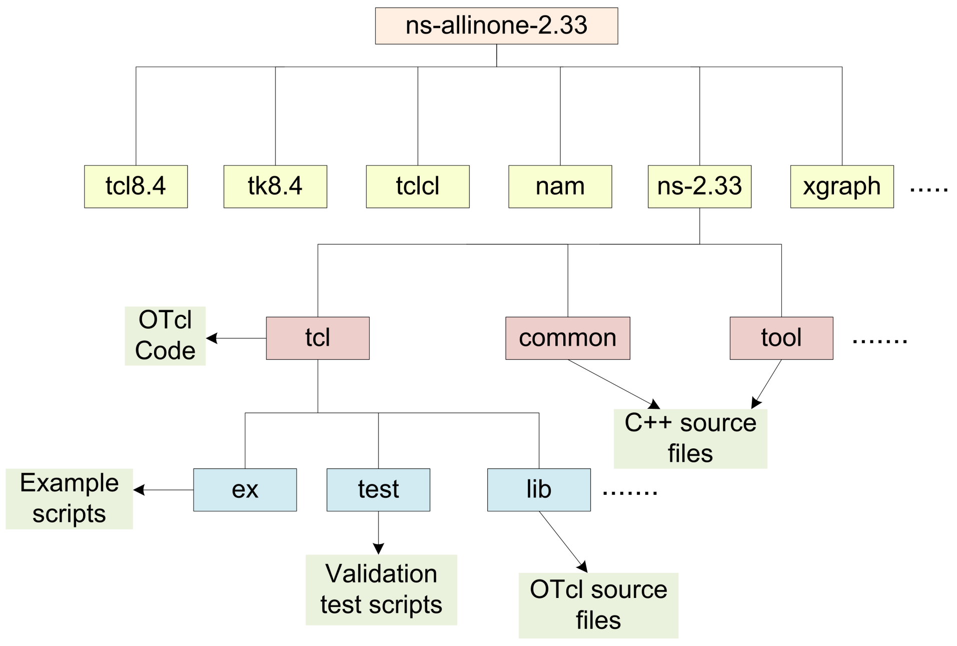


Figure 4.7: ns-2 directory structure

Among the subdirectories of the *ns-allinone-2.33* as shown in Figure 4.7, *ns-2* is the directory containing all of the simulator implementation files (either in C++ or in OTcl). Within this directory, all OTcl code including the test/example

OTcl scripts are located in the subdirectory *tcl*. C++ codes implementing something like event scheduling, agents, queue monitor, flow monitor are found in the directories *common* and *tool*. There are many other subdirectories inside the *ns-2* directory. For example, if one wants to see the implementation of the measured sum admission control algorithm, it is located in the directory *ns-2/adc* with the file name *ms-adc.cc*.

The directory *tcl*, has subdirectories among them is the *lib* directory which contains the OTcl implementation of some of the most important basic components like agent, address, link, queue and so on. Note that the OTcl source codes for LAN, web, and multicast implementations are located in separate subdirectories of *tcl*. The following are some explanations of the contents of some of the files in the *tcl/lib* directory:

- **ns-lib.tcl**: This file contains the simulator class and most of its member functions except those for LAN, web, and multicast. If one is looking for the implementation of the simulator class and its member function, this file is a nice place to start.

- **ns-default.tcl**: The default values for configurable parameters of various network components/protocols are contained in this file. Since most of the network components are implemented in C++, the configuration parameters are actually C++ member variable made available to OTcl through and an OTcl linkage function *bind*. This function is explained in subsection 4.3.3.2 together with how to make OTcl linkage from C++ code.

- **ns-packet.tcl**: The implementation of the packet header format initialization is located here. If one creates a new packet header, it should be registered in this file to make the packet header initialization process to include the new packet header in the header stack format and give it an offset in the stack.

- **Other OTcl files**: Other OTcl files in this directory, contain the implementations of compound network objects or the front end (control part) of network objects implemented in C++. The link network component is entirely implemented in OTcl and the source code is contained in the file *ns-link.tcl*. More information about these files can be found in the ns-2 manuel ([20]).

The two subdirectories of the *tcl* directory, *ex* and *test* might be interesting for someone looking for practical examples of how to design a specific simulation script.

4.3.3.2 How to Add a New Class Object to ns-2

Adding a real protocol to ns-2 is not so easy and requires much experience with the tool, C++ and OTcl programming languages. So to put a beginner on a fast track to gain much experience with ns-2 tool, a simple new class addition example is used here to demonstrate how to extend a class hierarchy and to show how the linkage (Figure 4.8) between OTcl and C++ class hierarchies is realised.

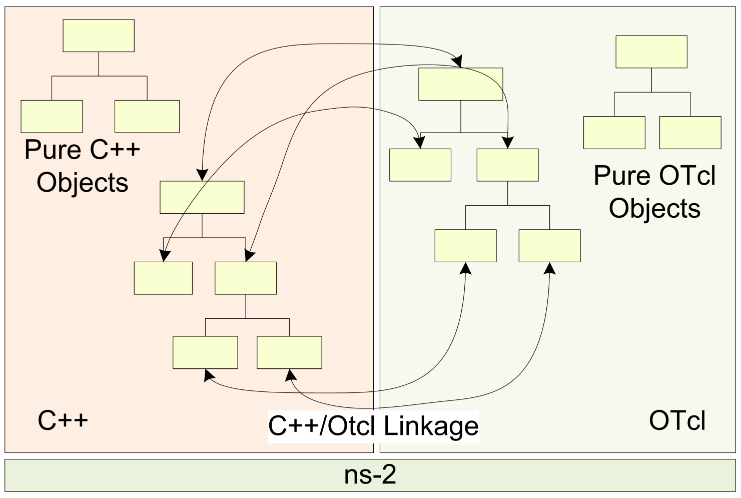


Figure 4.8: Class linkage between hierarchies

Figure 4.8 shows how a class hierarchy in C++ is connected one-to-one with an equivalent class hierarchy in OTcl. There are also some class hierarchies that only exist in compiled space or in interpreted space.

The new class example used for demonstration in this part of the lecture, extends the *Agent* class. The reason for this extension is to add the new class to the agent class hierarchy. It could have been possible to make the class a standalone, i.e., to have its own class hierarchy but it is more complex to implement. So it is considered less complex and simple (at least for beginner) to add the new class to an existing class hierarchy. This class simply accepts commands to output prepared messages to the standard output.

Like other networking protocols in ns-2, the new class named *MyTest* is implemented in C++ and mirrored to OTcl so that its object can be instantiated from OTcl space. The following C++ code shows the new class declaration:

```
#include <stdio.h>
#include "agent.h"

 class MyTest : public Agent{
```

```
 public:
      MyTest();
      int command(int argc, const char*const* argv);
      void sayhello(void);
      void outputmydata(void);
      void saygoodbye(void);
 protected:
      int mysize;
      int mystate;
};
```

This code could be saved in a file named *mytest.h*. To be able to instantiate an object of this class in OTcl, a linkage object e.g., *MyTestClass*, which is derived from *TclClass*, has to be created. This linkage object creates an OTcl object with a specific name (e.g., MyTestOtcl), and creates a linkage between the OTcl object (*MyTestOtcl*) and the C++ object (*MyTest*), using the TclObject member function *create*. As shown in the following code:

```
 static class MyTestClass: public TclClass {
 public:
         MyTestClass (): TclClass("Agent/MyTestOtcl") {}
         TclObject* create(int, const char*const*) {
                 return (new MyTest ())
         }
} class_my_test;
```

When ns-2 is first started, it executes the constructor for the static variable *class_my_test*, and thus an instance of *MyTestClass* is created. In this process, the *MyTestOtcl* class and its methods (member functions) are created in OTcl space (class hierarchy). Whenever a user in OTcl space tries to create an instance of this object using the command *[new MyTestOtcl]*, it invokes *MyTestClass::create* method, which creates an instance of *MyTest* object and returns the address. Be aware that creating a compiled (C++) object instance from intepreted (OTcl) space does not mean that one can freely invoke member functions or access member variables of the compiled object instance from intepreted space.

There are mechanisms implemented in compiled classes for preparing compiled member variables and member functions for accessing or invoking from interpreted space. These mechanisms are described as follows:

Setting compiled member variables from intepreted space: In normal case, access to compiled member variables is restricted to compiled code, and access to interpreted member variables is likewise confined to access via access interpreted code. However, it is possible to establish bi-directional binding such that both the compiled member variables and the interpreted member variables access the same data, and changing the value of either varible, changes the value of the corresponding paired variable to the same value.

The binding is established by the compiled constructor when that object is instantiated. It is then automatically accessible by the interpreted object as an instance variable. The ns-2 tool supports the binding of five different data types (Real, Integer, Bandwidth, Time, Boolean) and they have different syntax for specifying their values in OTcl:

1. **Real and Integer variables**: The method for binding these types of data variable is known as *bind()*. Their values in OTcl can be specified in normal form like for exmaple, *$object set realvar 1.2e3* or *$object set intvar 12*.
2. **Bandwidth variable**: The method for binding bandwidth variable is known as *bind_bw()*. Bandwidth in OTcl can be specified as a real value, optionally suffixed by a *k* or *K* to mean kilo-quantities, or *m* or *M* to mean mega-quantities. By default bandwidth is specified in bits per seconds. e.g., *$object set bwvar 1500k* or equivalently *$object set bwvar 1.5m*.
3. **Time variable**: The method for binding time variables is known as *bind_time()*. In OTcl, time variable can be specified as real values, optionally suffixed by a *m* to express time in milli-seconds, *n* to express time in nano-seconds, and *p* to express time in pico-seconds. By default time is expressed in seconds. e.g., *$object set timevar 1500m* which is equivalent to *$object set timevar 1.5*.
4. **Boolean variable**: The method for binding boolean variable is known as *bind_bool()*. In OTcl, boolean values can either be expressed as integer or *T* or *t* for true. If the value is neither an Integer value, nor a true value, it is then assumed to be false. e.g., *$object set boolvar t* or equivalently *$object set boolvar 1*.

In the new class, the two member variables can be binded in the constructor definition with the following code:

```
MyTest::MyTest () : Agent(PT_UDP) {
        bind("mysize", &mysize);
        bind_bool("mystate", &mystate);
}
```

The first parameter in the *bind* method is the name of the variable in the interpreted space and the second parameter is the address of the variable in the compiled space. Be aware that whenever a compiled member variable is binded, it is recommended to set a default value for it in the interpreted space (in the file *ns-default.tcl*). Otherwise a warning message will be issued when one tries to create an instance of the compiled object.

Invoking compiled member function from interpreted space: To invoke a C++ member method from OTcl, a *command()* method is defined in C++ object class, which works as an OTcl command interpreter. This *command()* method is invoked by the following means: for every Tclobject that is created, ns-2 establishes the instance procedure *cmd*{}, as a hook to executing methods through the compiled shadow object. The procedure *cmd*{} invokes the method *command()* of the shadow object automatically passing the arguments to *cmd*{} as an argument vector to the *command()* method.

The user can call the *cmd*{} procedure by implicitly specifying the name of the desired C++ member function as if there exist an instance procedure of the same name. For example if the member function name is *distance*, one can invoke it as:

```
$object distance <args>
```

since there is no instance procedure called *distance*, the interpreter will invoke the instance procedure *unknown*{}, defined in the base class Tclobject. The unknown procedure then invokes the *cmd*{} as shown below:

```
$object cmd distance <args>
```

to execute the operation through the compiled object's *command()* method.

As an example, the command member method of the new class object is defined with the following C++ code:

```
int MyTest::command(int argc, const char*const* argv) {
     if(argc==2) {
         if (strcmp(argv[1],"sayhello") == 0) {
                sayhello();
                return(TCL_OK);
         }else if (strcmp(argv[1],"outputmydata") == 0) {
                outputmydata();
                return(TCL_OK);
         }else if (strcmp(argv[1],"saygoodbye") == 0) {
                saygoodbye();
                return(TCL_OK);
         }
    }
     return(Agent::command(argc, argv));
}
```

The member functions are defined as follows:

```
//sayhello method definition
void MyTest::sayhello() {
  printf("\nHello! Welcome to the testing of OTcl/C++ linkage.\n");
  printf("The configured data are outputted in a while\n\n");
}

 //outputmydata method definition
void MyTest::outputmydata() {
  if(mystate) {
    printf("The variable mystate is true and mysize is %d\n\n", mysize);
   } else {
    printf("The variable mystate is false and mysize is %d\n\n", mysize);
  }
}

//saygoodbye method definition
void MyTest::saygoodbye() {
  printf("I hope you enjoyed the lectures! Till next time\n");
}
```

The new class constructor definition and the member functions definitions can be saved in a file named *mytest.cc*, which includes the class declaration (*mytest.h*) as a header file. The two files could be put in a folder named *myclass* and added to the *ns-2* directory as shown in Figure 4.9.
The default configuration values should be written to the *lib/ns-default.tcl* file. In order to compile the new class, the path of the output file (*myclass/mytest.o*) should be added to the class object list in the *Makefile.in* file. Then the *configure* command should be called to generate the makefile and the *make* command should be called to compile the whole files.

4.3.3.3 Testing and Result of the New Class Object

The new class implemented in the last subsection to demonstrate the linkage between compiled class hierarchy and interpreted class hierarchy can be tested with a simulation script. The script configures the member variables and invokes the member functions at scheduled times.

The test simulation script is named *test-suite-mytest.tcl* and designed as follows:

```
#configure the member variables
#this will overwrite the values configured in ns-default.tcl
Agent/MyTestOtcl set mysize 40
Agent/MyTestOtcl set mystate t
```

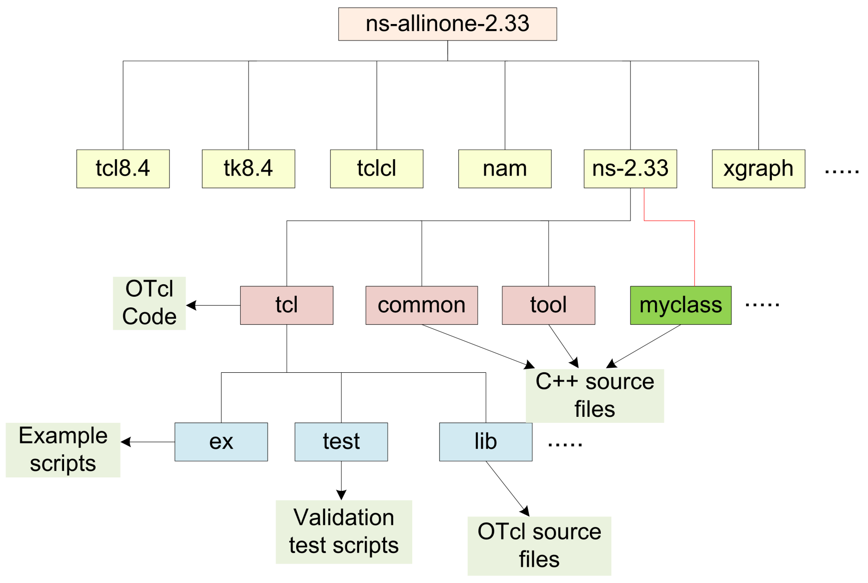


Figure 4.9: ns-2 directory structure extended

```
#create simulator instance for scheduling
set ns [new Simulator]

#create instance of the new class
set mytest [new Agent/MyTestOtcl]

#write a finish procedure
proc finish {} {
  exit 0
}

#schedule the time to call member functions
$ns at 0.2 "$mytest sayhello"

$ns at 0.8 "$mytest outputmydata"

$ns at 1.0 "$mytest saygoodbye"

$ns at 1.2 "finish"

#start the simulator
$ns run
```

This test script can added to the *ex* directory in the ns-2 directory structure (see Figure 4.9). If the installation path of ns-2 is added to the system path, the simulation can be executed with the command:

```
ns test-suite-mytest.tcl
```

The simulation produces the following result:

```
Hello! Welcome to the testing of OTcl/C++ linkage.
The configured data are outputted in a while

The variable mystate is false and mysize is 30

I hope you enjoyed the lectures! Till next time
```

If the default values are not configured for the member variables, the following warning message is outputted:

```
warning: no class variable Agent/MyTestOtcl::mysize

 see tcl-object.tcl in tclcl for info about this warning.

warning: no class variable Agent/MyTestOtcl::mystate
```

Note: The new class example presented here is a dummy class object. That is why its implementation and integration is fast and straight forward. Readers should not think that adding a real network protocol to ns-2 is so simple and easy. It requires fairly much experience with the tool and much more changes or extensions to the source files. Interested readers are referred the tutorials in reference [27, 12] for examples of how to add real network protocol to ns-2 tool.

4.4 Summary

The ns-2 tool is an object-oriented discrete event driven simulator targeted for network research. It is widely used today in many institutions and research centers for teaching and carrying out research in internetworking area. It covers a large set of networking protocols, which includes wired networks, wireless networks, and satellite networks. The tool is an open source software package, which is free to download and it is possible to make changes to the source code. The network simulator is written in two programming languages (C++ and OTcl), with the primary aim of supporting detailed simulations and fast testing of different parameter configurations. C++ is fast to run and can be used for detailed system programming such as byte/packet header manipulations, and algorithm implementations but it is slow to change. OTcl is slow to run but fast to change making it suitable for quickly testing various configurations for different scenarios. The classes written in C++ are known as the compiled hierarchy and the classes written in OTcl are known as the interpreted hierarchy. The compiled hierarchy and the interpreted hierarchy are linked together so that member variables in compiled hierarchy could be accessed and configured from the interpreted hierarchy making it possible to have the same variable values on both sides. Simulation scenarios are designed and written only in OTcl. There are the possibilities of extending ns-2 with custom functionalities. Users of ns-2 must be aware that inspite the considerable confidence of the developers in the tool, ns-2 is not a polished and finished product, but the result of an on-going effort of research and development. In particular, bugs in the software are still being discovered and corrected. So users are responsible for verifying for themselves that their simulations are not invalidated by bugs. The developers are aiding the users in this aspect by develping many validation scripts and test scripts included in the tool. The ns-3 project currently being developed to replace the current ns-2 version of the tool.

5 Code Description of the Multiservice Framework

This chapter describes the implementation of the multiservice framework introduced in chapter 3, section 3.4. The framework is added to ns-2 as a component, designed for simulating and measuring quality of service of simultaneous data traffic flows traversing in a packet network. The multiservice framework is an extension of an already existing single service framework for simulating and measuring quality of service of a single controlled-load service.

This framework is made up of four basic components (Figure 5.1) namely:

1. Queue scheduler.
2. Signalling mechanism.
3. Enhanced link.
4. Admission control, which includes the estimation mechanism.

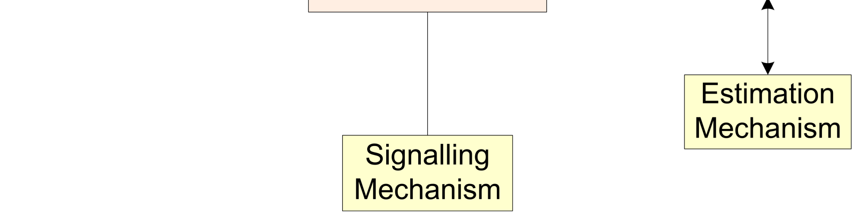


Figure 5.1: Multiservice framework components

These components work together to achieve the functionalities of the multiservice framework. The components are lined up in a multiservice link through which the packets of each traffic class traverse and they handle each packet according to its class priority. The codes of the components are explained in the subsequent sections of this chapter.

5.1 Multiservice Queue Scheduler Implementation

This section deals with the code description of the multi-queue scheduler class. This class is named *MultiClassServ*, its declaration and members definitions are contained in the C++ file *ns-2.33/adcextension/multi-class-serv.cc*. It is a queue management class derived from a hierarchy of C++ base classes with the direct base class *Queue*. Figure 5.2 shows UML class diagrams depicting the *MultiClassServ* class and its direct parent *Queue* class.

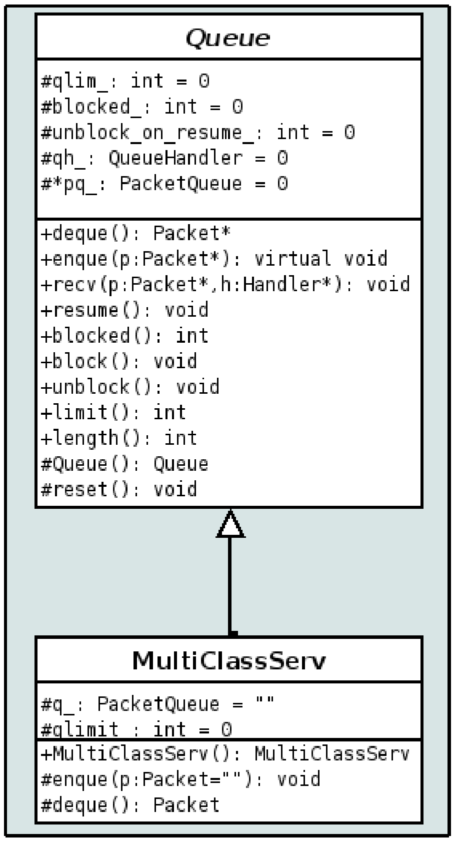


Figure 5.2: UML class diagram of multi-queue scheduler

In Figure 5.2, the class attributes and operations[1] are listed. More attension

[1]In UML, the class member variables are known as attributes and the class member functions are known as operations.

is paid to the *MultiClassServ* queue management class and not its base class[2]. The queue management is used to schedule the data packets generated from the three different classes of traffic flows defined in the multiservice framework. The scheduling mechanism is based on the FIFO scheduling with the improvement of queues being assigned different level of network resources based on the priority of the traffic class.

5.1.1 Queue Components description

This queue management class (*MultiClassServ*) is made of two member variables and three member functions. Their importance in the class are described as follows:

- **MultiClassServ()**: This is the class constructor, which is used for creating the packet queues and initializing member variables. It also binds the array member variable *qlimit_* so that it can be configured from OTcl scripts.

- **enque()**: A member function with *void* return value. It is used for adding the data traffic flows to their respective packet queues. The function drops packets to any queue that is full, but it ensures that the signalling protocol packets are never droped.

- **deque()**: A member function with *Packet* type return value. It deques packets from their queues and passing them to the next hop along their destination direction.

- **q_**[..]: An array pointer variable from data type *PacketQueue*. It holds pointers to the addresses of the three different packet queues created in this class. The three packet queues are conceived for the three data traffic types defined in the multiservice framework.

- **qlimit_**[..]: An Integer array variable of size three, defined to specify the sizes of the three packet queues.

5.2 Signalling Mechanism Implementation

The signalling mechanism is a simple end-to-end signalling protocol, that uses a three way handshake in requesting flow connections. The signalling mechanism is based on the RSVP protocol, but it does not implement a full RSVP protocol suite.

This section describes the C++ implementation of the signalling mechanism. It is realised as a C++ class named *MultiSALink*. The class declaration and

[2]Interested readers are referred to the ns-2 manual for details about the queue class.

member variables definitions are contained in a C++ header file named *ns-2.33/adcextension/multisalink.h*, and the member functions definitions are found in the C++ source file with the name *ns-2.33/adcextension/multisalink.cc*. The class is derived from a hierarchy of base classes. The direct base class is the *Connector* class, which provides the functions for sending packets to next hop. Figure 5.3 presents UML class diagrams of the *Connector* and *MultiSALink* classes.

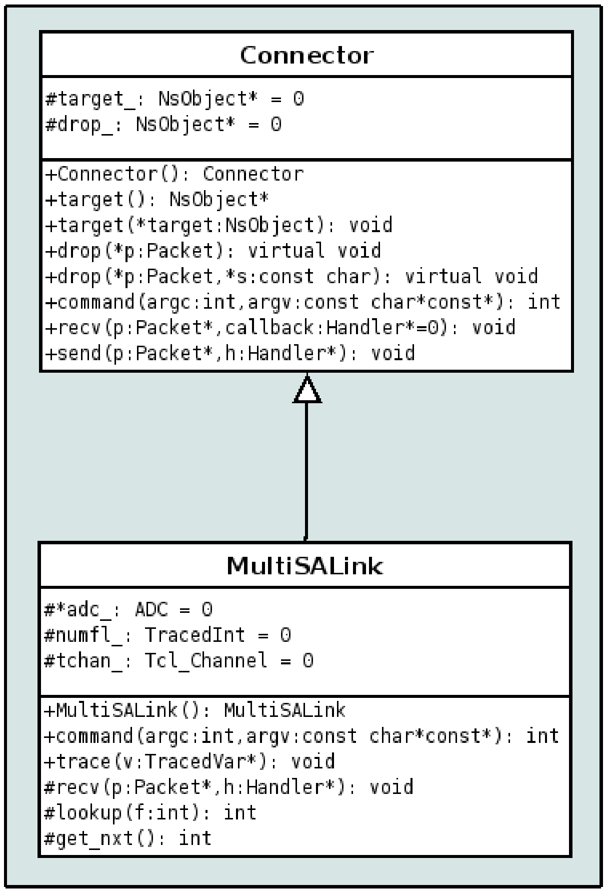

Connector
#target_: NsObject* = 0
#drop_: NsObject* = 0
+Connector(): Connector
+target(): NsObject*
+target(*target:NsObject): void
+drop(*p:Packet): virtual void
+drop(*p:Packet,*s:const char): virtual void
+command(argc:int,argv:const char*const*): int
+recv(p:Packet*,callback:Handler*=0): void
+send(p:Packet*,h:Handler*): void
MultiSALink
#*adc_: ADC = 0
#numfl_: TracedInt = 0
#tchan_: Tcl_Channel = 0
+MultiSALink(): MultiSALink
+command(argc:int,argv:const char*const*): int
+trace(v:TracedVar*): void
#recv(p:Packet*,h:Handler*): void
#lookup(f:int): int
#get_nxt(): int

Figure 5.3: Signalling mechanism class diagram

The inclusion of the *Connector* class in the class diagram is done due to completeness, the discussion here is focused on the *MultiSALink* class. The UML class diagrams present only some member functions and member variables of the classes. Interested readers are referred to the list of source codes at the end of the thesis for the full list of member functions and member variables of the classes.

5.2.1 Signalling Components Description

The *MultiSALink* class is made up of member functions and member variables. These components together achieve the functionalities of this class. The meaning and importance of these components are described as follows:.

- **MultiSALink()**: This is the class constructor. It is responsible for initializing member variables, the reservation state table, and binding member variables so that they can be accessed from the OTcl space.

- **command()**: This function is used for executing C++ member functions invoked from OTcl space. It accepts the command invoked in OTcl space with it arguments vectors and executes the corresponding member function in the C++ space. The two parameters of this function: *argc* represents the number of arguments passed to the invoked command in OTcl space, and *argv* represents the arguments passed. It returns an error code to the OTcl interpreter in the case of error, otherwise success code is returned.

- **trace()**: A member function with return type *void*. It is used for logging trace events during the network operations. It takes one parameter:*v* from type *TracedVar*, which gives the name of the event to be logged.

- **recv()**: This function returns a *void* type. It receives the reservation messages and carrys out the resource reservation processes, which includes calling the admission decision algorithm to decide on flow request and then send the flow down to the next hop on the reservation path. It also controls the inserting and removal of flow states in the resource reservation table. The two parameters: *p* represents the packet carrying the reservation message, and *h* is a handler instance for the event scheduler.

- **lookup()**: It returns an *Integer* value. The function is responsible for checking if a flow already exists in the resource reservation state table. It takes one parameter *flowid* which is the ID of the flow to be looked up in the table. It returns the flow ID on success and minus one if it doesn't exist in the table.

- **get_nxt()**: A member function with return type *Integer*. The function checks for available space in the resource reservation state table. On success, it returns the available space otherwise it returns an error message showing that the table is full.

- **adc_**: A pointer member variable from the type *ADC*. It is used here to invoke the admission decision algorithms.

- **numfl_**: A member variable from the type *TracedInt*. It used to count the number of reserved flows.

- **tchan_**: A member variable from the type *Tcl_Channel*. It represents the trace file descriptor ID for writing trace events during network operations.

5.3 Multiservice Enhanced Link Implementation

The enhanced link in ns-2 is realised as a class entirely written in OTcl. This section describes the OTcl implementation of the link class and the composition of the link as a network component.

The multiservice link class has the name *MultiServLink*. Its declaration and the definition of the components is contained in the Tcl file *lib/ns-multiserv.tcl*. The *MultiServLink* class is derived from a hierarchy of OTcl base classes with *SimpleLink* as its direct base class. Figure 5.4 shows UML diagrams of the link classes.

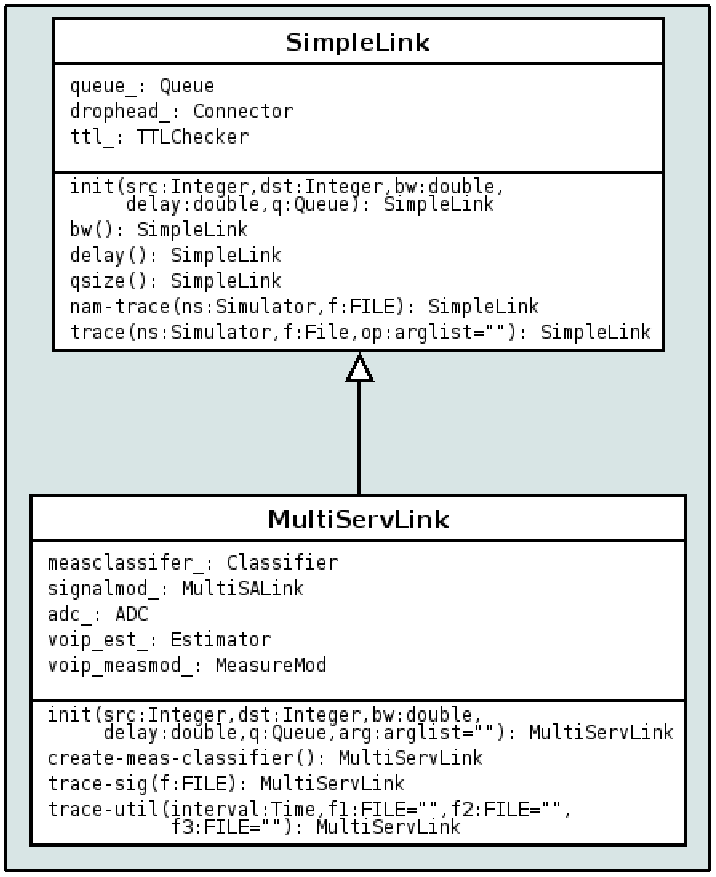

SimpleLink
queue_: Queue
drophead_: Connector
ttl_: TTLChecker
init(src:Integer,dst:Integer,bw:double, delay:double,q:Queue): SimpleLink
bw(): SimpleLink
delay(): SimpleLink
qsize(): SimpleLink
nam-trace(ns:Simulator,f:FILE): SimpleLink
trace(ns:Simulator,f:File,op:arglist=""): SimpleLink
MultiServLink
measclassifer_: Classifier
signalmod_: MultiSALink
adc_: ADC
voip_est_: Estimator
voip_measmod_: MeasureMod
init(src:Integer,dst:Integer,bw:double, delay:double,q:Queue,arg:arglist=""): MultiServLink
create-meas-classifier(): MultiServLink
trace-sig(f:FILE): MultiServLink
trace-util(interval:Time,f1:FILE="",f2:FILE="", f3:FILE=""): MultiServLink

Figure 5.4: Multiservice link class diagram

The class diagrams present the *MultiServLink* class and the *SimpleLink* class. The member functions and member variables of the *SimpleLink* class are not complete in the diagram and the description of this class is not the focus in this section. Interested readers are referred to the ns-2 manual ([20]) for complete details of this class. To avoid confusion, note that member functions in C++ are known as instance procedures in OTcl as already explained in chapter 4, section

4.3.1.5.

The multiservice link as a network component is a compound object, composing of queue object, admission control object, resource reservation mechanism object, measurement object, estimator object, and the classifier object. These objects work together to guarantee quality of service to application traffic flows traversing the link.

5.3.1 Multiservice Link Components Description

As shown in Figure 5.4, the *MultiServLink* class diagram is made of attributes and procedures, which are known as instance procedures and instance variables in OTcl language. This section describes their essences in the *MultiServLink* class.

- **init{}**: This instance procedure is equivalent to the constructor in C++ programming language. It is responsible for initializing the class object. As mentioned earlier that the multiservice link as a network object is made up of other objects, the set up and initialization of those objects are done in this procedure. Class instance variable declarations and the calling of inheritted instance procedures are made in this instance procedure. The parameters of this procedure are: *src* the source address of a traffic traversing the link, *dst* the destination address of a traffic, *bw* the link bandwidth value, *delay* the delay component, *q* the queueing type, and *arg* the list of the rest arguments passed to the link.

- **create-meas-classifier{}**: This instance procedure is responsible for creating the flow classifier object and setting up the classification table, which is then used to classify the traffic packets according to their flow ID.

- **trace-sig{}**: An instance procedure used to invoke a C++ function to log trace events. For example to log trace events of reserved flow in the resource reservation mechanism. It has one parameter *f*, which is the file descriptor of the trace file to store the trace events.

- **trace-util{}**: This instance procedure is designed to log the network utilization value and the network average load every specific interval of time. This logging is done for each of the traffic types traversing the network. The data written by this instance procedure are used to plot the network utilization and network estimation graphs. This procedure has four parameters, one required and three are optional. The required parameter *interval* is the time value at which to log the utilization and estimation values. The three optional parameters *f1*, *f2*, *f3* are the file descriptors to store the values. When no file descriptor is given for a class of traffic flow, the utilization and estimation values of this class is not logged.

5.4 Implementation of Admission Control Algorithms

This section describes the implementation of the four measurement-based admission control algorithms (see chapter 3, section 3.3.2.4) integrated in ns-2 to support the single service framework (explained in chapter 3, section 3.4) and the modifications made to these algorithms to support the multiservice framework. The admission control mechanism relies on the estimation mechanism for making its admission decisions. Thus, the estimation mechanism is an indispensible part of the admission control mechanism.

The admission control mechanism is realised as a C++ abstract base class with the name *ADC*. The class declaration and member variables definitions are contained in the header file *ns-2/adc/adc.h*, and the member functions definitions are contained in the C++ file *ns-2/adc/adc.cc*. The *ADC* class is an abstract class[3] derived from a hierarchy of base classes. It forms the base class for the admission control algorithms.

The estimation mechanism is implementated as a C++ abstract base class with the name *Estimator*. The class declaration and member variables definitions are stored in the header file *ns-2/adc/estimator.h*, and the member functions definitions are stored in the C++ file *ns-2/adc/estimator.cc*. The *Estimator* class like the *ADC* class is an abstract class derived from a hierarchy of base classes. It forms the base class for the three measurement mechanisms[4] described in chapter 3, section 3.3.2.5.

Figure 5.5 presents UML class diagrams of the *ADC* class with its child classes, and the *Estimator* class with its child classes. The class diagrams expose the dependencies of the admission control mechanism on the estimation mechanism and how the admission control classes use the estimation classes. Generally, for reasons of better performance, it is recommended to pair the admission decision algorithms with the estimation mechanism as follows:

- Measured sum algorithm with time-window estimation mechanism.
- Hoeffding bound algorithm with the exponential averaging estimation mechanism.
- Acceptance region tangent at origin and acceptance region tangent at peak with the point sample estimation mechanism.

This recommended pairing is also depicted in Figure 5.5. They are realised through the estimator interface in the admission control base class. Note that

[3]An abstract class can not be instantiated, i.e., create an object of the class. It is mainly used as a base class to implement general functions for child classes.

[4]Note that the estimation mechanism is based on measurement, so it is sometimes referred to as measurement mechanism.

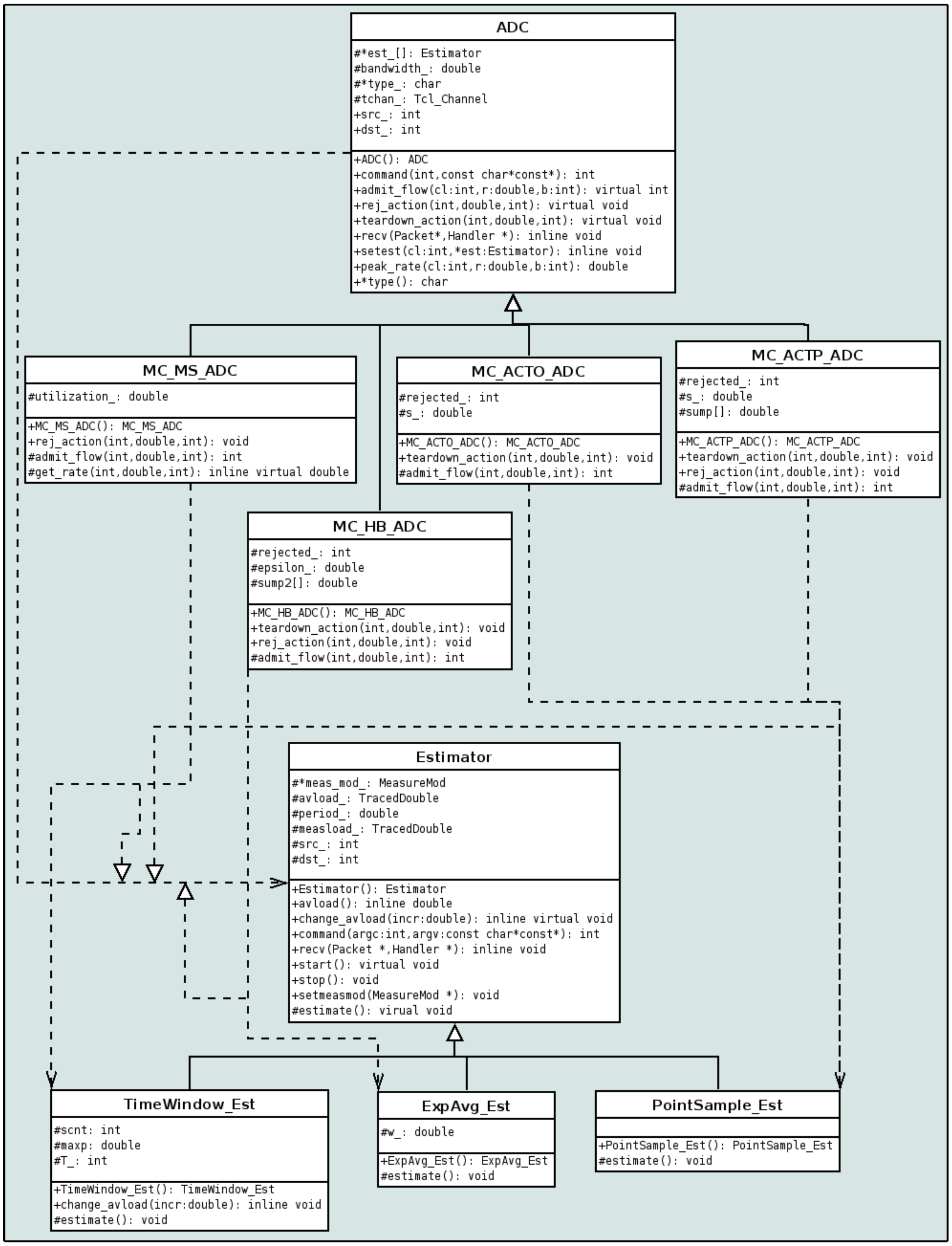

ADC
#*est_[]: Estimator
#bandwidth_: double
#*type_: char
#tchan_: Tcl_Channel
+src_: int
+dst_: int
+ADC(): ADC
+command(int,const char*const*): int
+admit_flow(cl:int,r:double,b:int): virtual int
+rej_action(int,double,int): virtual void
+teardown_action(int,double,int): virtual void
+recv(Packet*,Handler *): inline void
+setest(cl:int,*est:Estimator): inline void
+peak_rate(cl:int,r:double,b:int): double
+*type(): char
MC_MS_ADC
#utilization_: double
+MC_MS_ADC(): MC_MS_ADC
+rej_action(int,double,int): void
#admit_flow(int,double,int): int
#get_rate(int,double,int): inline virtual double
MC_ACTO_ADC
#rejected_: int
#s_: double
+MC_ACTO_ADC(): MC_ACTO_ADC
+teardown_action(int,double,int): void
#admit_flow(int,double,int): int
MC_ACTP_ADC
#rejected_: int
#s_: double
#sump[]: double
+MC_ACTP_ADC(): MC_ACTP_ADC
+teardown_action(int,double,int): void
+rej_action(int,double,int): void
#admit_flow(int,double,int): int
MC_HB_ADC
#rejected_: int
#epsilon_: double
#sump2[]: double
+MC_HB_ADC(): MC_HB_ADC
+teardown_action(int,double,int): void
+rej_action(int,double,int): void
#admit_flow(int,double,int): int
Estimator
#*meas_mod_: MeasureMod
#avload_: TracedDouble
#period_: double
#measload_: TracedDouble
#src_: int
#dst_: int
+Estimator(): Estimator
+avload(): inline double
+change_avload(incr:double): inline virtual void
+command(argc:int,argv:const char*const*): int
+recv(Packet *,Handler *): inline void
+start(): virtual void
+stop(): void
+setmeasmod(MeasureMod *): void
#estimate(): virual void
TimeWindow_Est
#scnt: int
#maxp: double
#T_: int
+TimeWindow_Est(): TimeWindow_Est
+change_avload(incr:double): inline void
#estimate(): void
ExpAvg_Est
#w_: double
+ExpAvg_Est(): ExpAvg_Est
#estimate(): void
PointSample_Est
+PointSample_Est(): PointSample_Est
#estimate(): void

Figure 5.5: Admission control and estimator class diagrams

other pairing forms could also achieve good performance in certain situations. The admission control classes and the estimation classes are described in more details in the subsequent subsections.

5.4.1 Admission Control Classes Description

The admission control base class *ADC* presents abstract interfaces through which the decision algorithms classes are invoked. The concrete implementation of the interface functions declared in this class is done in the child classes. This class defines the following basic member functions:

- **ADC()**: This is the class constructor used to initialize member variables and also bind some of the variables that is desired to be configured from the OTcl space.

- **setest()**: A member function used in pairing an admission control object with an estimator object. It has two parameters: *cl* an Integer variable which identifies the position of the estimator object in an array of estimators object, and *est* a pointer to the estimator object.

- **peak_rate()**: A member function that returns a *double*. It is used to calculate a traffic's peak rate, using the token bucket parameters. The function has three parameters: *cl* an Integer value to specify the class of the traffic flow, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size.

- **command()**: This function implements the possibilities of executing *ADC* and *Estimator* member functions from OTcl space. For the multiservice framework an extension is made to this function to start the estimator objects for the three class of services. The two parameters of this function: *argc* represents the number of arguments passed to the invoked command in OTcl space, and *argv* represents the arguments passed. It returns an error code to the OTcl interpreter in the case of error, otherwise success code is returned.

This class is extended by the four admission decision algorithms. The mathematical description and explanation of these algorithms have been done earlier in chapter 3, section 3.3.2.4. The implementation of the dynamic bandwidth allocation mechanism is integrated with the implementation of these algorithms but due to the fact that this mechanism is fully explained in chapter 3, section 3.4.2, the code description is not done here. Thus, only the implementational description of the algorithms are feautured in the next subsections.

5.4.1.1 Measured Sum Algorithm Code Description

The measured sum algorithm is realised as a C++ class named *MC_MS_ADC*. The class declaration and the definition of its member functions and member variables are contained in the C++ file *ns-2/adcextension/multi-ms-adc.cc*. The composed member functions and member variable are of the following importance:

- **MC_MS_ADC()**: The class constructor, responsible for initializing the member variables and binding the member variable *utilization_* thereby making it possible to be configured from OTcl space. It also initializes the bandwidth allocation and borrowing mechanism.

- **admit_flow()**: This function is very essential to the class. It is used for admitting new flows into the network as far as bandwidth is available for the existing flow plus the requirements of the new flow. During the admission process, this function calls the time-window estimator class, which produces the average current network load of each traffic class. It then adds the class traffic rate to the class average load and compare the sum aginst the class bandwidth before admitting a new flow of the class type. When a new flow is admitted, the average network load of that flow class is increased by the rate of that flow. The dynamic bandwidth allocation mechanism is invoked in this function to dynamically allocate the best effort class bandwidth to the higher priority VoIP and video flow classes when appropriate. This function has three parameters: *cl* an Integer value to specify the class of the traffic flow, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. The function returns an *Integer*. It returns the value one when the flow is admitted otherwise the value zero.

- **rej_action()**: A member function that returns a *void*. It is used to decrement the average network load when traffic flows are rejected at the admission point in the network. It has three parameters: *cl* an Integer value to specify the class of the traffic flow, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. The parameters are used to characterize the traffic flow.

- **get_rate()**: This member function is used to extract the rate of the traffic flow. It returns the traffic rate as a double value.

- **utilization_**: A member variable for setting the utilization factor for the algorithm that controls the network utilization threshold.

5.4.1.2 Hoeffding Bounds Algorithm Code Description

The implementation of this algorithm is achieved with a C++ class named *MC_HB_ADC*. The class declaration and definitions of its member functions and member variables are contained in the C++ file *ns-2/adcextension/multi-hb-adc.cc*. The member functions and member variables together achieve the functionalities of this class. The details of these components are as follows:

- **MC_HB_ADC ()**: This function is the class constructor. It initializes the member variables and binds the *epsilon_* member variable so that it can be configured from the OTcl space. It also initializes the bandwidth allocation and borrowing mechanism.

- **admit_flow()**: This function returns an *Integer* value. It is the function that implements the admission decision functionality. It admits a new flow into the network if there is available bandwidth for the existing flows plus the requirements of the new flow. During the admission process, it invokes the exponential averaging estimator object, which provides the current network average load of each of the traffic classes. The sum of this average load with the other parameters of the algorithm equation is compared against the class bandwidth before accepting a new flow of a class traffic type. When a new flow is admitted, the square of the peak rate of the flow is added to the sum of the squares of the peak rates of already admitted flows of this class and the network average load of this class is artificially increased. It also calls the dynamic bandwidth allocation mechanism used to allocate best effort class bandwidth to the higher priority VoIP and video classes when it is appropriate. This function has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. It calculates the peak rate of the traffic flows with the help of these parameters, which is used in the decision algorithm. When a flow is admitted, it returns the value one, otherwise it returns the value zero and sets the *rejected_* member variable.

- **teardown_action()**: A member function that is called when a connection in the network is terminated to decrease the sum of the peak rates of accepted flows in the network by the square of the flow's peak rate and clears the *rejected_* variable to indicate to the admission decision function to accept once more the flows of this type that just exited the network. It has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. These parameters are used to calculate the peak rate of the flow. The function returns a *void*.

- **rej_action()**: This member function is responsible for decrementing the sum of the peak rates of flows in the network when a flow is rejected by the square of the flow's peak rate. It has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and b an Integer value specifying the token bucket size. These parameters are used to calculate the peak rate of the flow. The function returns a *void*.

- **rejected_**: This member variable is set to indicate to the admission function that a flow of this traffic type should be denied connection until a flow of its traffic type leaves the network.
- **sump2**[..]: An array of member variables to store the sum of the squared peak rates of the different traffic classes.
- **epsilon_**: A member variable representing the equivalent bandwidth in the decision algorithm.

5.4.1.3 Acceptance Region Tangent at Peak Algorithm Code Description

The acceptance region tangent at peak algorithm is implemented as a C++ class named *MC_ACTP_ADC*. The class declaration and definitions are contained in the C++ file *ns-2/adcextension/multi-actp-adc.cc*. It is composed of member functions and member variables. The essence of the member functions and some of the member variables are detailed as follows:

- **MC_ACTP_ADC()**: This is the class constructor, it initializes the member variables and binds the *s_* member variable so that it can be configured from the OTcl space. It also initializes the bandwidth allocation and borrowing mechanism.
- **admit_flow()**: The member function is very essential in the class. It returns an *Integer* value. The function implements the admission decision process which admits a new flow if there is enough class bandwith for the existing flows in the network and for the resources required by the new flow. During the admission process, it invokes the point sample estimator object, which provides the current network average load for each of the traffic classes. This average load is added to the other parameters of the algorithm equation and compare the sum against the class bandwidth before accepting a new flow for a class of traffic. When a new flow is admitted, it adds the flow's peak rate to the sum of the peak rates of the existing flows of this traffic class. It is manually decided whether to artificially increase this flow class average network load by the peak rate of the new flow or not. The function also calls the dynamic bandwidth allocation mechanism used to allocate best effort class bandwidth to the higher priority VoIP and video classes when it is appropriate. This function has three parameters: cl an Integer value to specify the class of the traffic flow, r a double value which specifies the token rate, and b an Integer value specifying the token

bucket size. It calculates the peak rate of the traffic flows with the help of these parameters, which is used in the decision algorithm. When a flow is admitted, this function returns the value one otherwise it returns the value zero and the *rejected_* member variable is setted.

- **teardown_action()**: This member function is called when a connection in the network is terminated to decrease the sum of the peak rates of accepted flows in the network by the flow's peak rate and clears the *rejected_* variable to indicate to the admission decision function to accept once more the flows of this type that just exited the network. It has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. These parameters are used to calculate the peak rate of the flow. The function returns a *void*.

- **rej_action()**: This member function is used for decrementing the sum of the peak rates of flows in the network when a flow is rejected. It has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. These parameters are used to calculate the peak rate of the flow. The function returns a *void*.

- **rejected_**: This member variable is set to indicate to the admission function that a flow of this traffic type should be denied connection until a flow of its traffic type leaves the network.

- **sump**[..]: An array of member variables to store the sum of the peak rates of the different traffic classes.

- **s_**: A member variable representing space parameter of the chernoff bounds in the decision algorithm.

5.4.1.4 Acceptance Region Tangent at Origin Algorithm Code Description

The implementation of this algorithm is realized as a C++ class with the name *MC_ACTO_ADC*. The class declaration and definitions are contained in the C++ file *ns-2/adcextension/multi-acto-adc.cc*. The class is made up of member functions and member variables. These components collectively achieve the functionality of the class. The importance of these components are described as follows:

- **MC_ACTO_ADC()**: This is the class constructor, it initializes the member variables and binds the *s_* member variable so that it can be configured from the OTcl space. It also initializes the bandwidth allocation and borrowing mechanism.

- **admit_flow()**: This member function is very important in the class. It returns an *Integer* value. This function implements the admission decision function, which admits a new flow if there is enough class bandwith for the existing flows in the network plus the resources required by the new flow. During the admission process, it invokes the point sample estimator object, which provides the current average network load for each of the flow traffic classes. This average load is added to the other parameters of the algorithm equation and the resulting sum is compared against the class bandwidth before accepting a new flow for a traffic class. When a new flow is admitted, it is manually decided whether to artificially increase the average network load of this flow class by the flow's peak rate or not. The function also calls the dynamic bandwidth allocation mechanism used to allocate best effort class bandwidth to the higher priority VoIP and video classes when it is appropriate. This function has three parameters: *cl* an Integer value to specify the class of the flow traffic, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. It calculates the peak rate of the traffic flows with the help of these parameters, which is used in the decision algorithm. When a flow is admitted, the function returns the value one otherwise it returns the value zero and the *rejected_* member variable is set.

- **teardown_action()**: This member function is called when a connection in the network is terminated to clear the *rejected_* variable, which indicates to the admission decision function to accept once more the flows of this traffic type that just exited the network. It has three parameters *cl* an Integer value to specify the class of the traffic flow, *r* a double value which specifies the token rate, and *b* an Integer value specifying the token bucket size. These parameters are used to calculate the peak rate of the flow. The function does not return any value.

- **rejected_**: This member variable is set to indicate to the admission function that a flow of this traffic type should be denied connection until a flow of its traffic type leaves the network.

- **s_**: A member variable representing space parameter of the chernoff bounds in the decision algorithm.

5.4.2 Estimator Classes Description

The admission control mechanism is dependent on the estimation mechanism. The *Estimator* base class provides abstract interfaces through which the admission control classes access the estimation classes. This base class is composed of some other classes like the *MeasureMod*, which performs the bits/packets measurement, and *TracedDouble* class, which provides a data type for storing the average network load.

The *Estimator* base class provides some general member functions and member variables, which are inherited by its three child classes. The details of some of these components are given as follows:

- **Estimator()**: This is the class constructor. It initializes the member variables including the composed objects and binds the *period_*, *dst_*, and *src_* member variables so that they can be configured from the OTcl space.
- **avload()**: A member function that returns *double* values. It is called to return the measured average newtork load casted to a double value.
- **change_avload()**: A member function that is used to artificially increase or decrease the network average load. It has one parameter: *incr* from type double, which represents the value by which the network average load is artificially increased or decreased. The function returns no value due to its *void* return type.
- **command()**: This member function is used to execute other member functions invoked from the OTcl space. It executes the commands to log the network utilization, the estimated average load, and to attach trace files. It returns an error code back to OTcl interpreter in case of error occurance otherwise a success code is returned.
- **start()**: A member function to start the estimation process. It resets the *avload_* and *measload_* member variables and then schedules the estimation period. It has no return value.
- **stop ()**: This member function is used to cancel the estimation process. It has no return value.
- **setmeasmod()**: This member function hooks the measurement object to the estimation object. It returns no value.
- **period_**: A member variable that specifies the interval of time between different estimation processes.
- **avload_**: A member variable used to store the calculated network average load.
- **measload_**: A member variable used to store the current measured network average load.

The three different estimation mechanisms have been theoretically and mathematically described earlier in chapter 3, section 3.3.2.5. Thus, the subsequent subsections describe only their implementations.

5.4.2.1 Time-Window Estimator Code Description

The time-window estimation mechanism is realised as a C++ child class derived from the *Estimator* base class. The class is named *TimeWindow_Est* and its declaration and definitions are contained in the C++ file *ns-2/adc/timewindow-est.cc*. The class is made up of three member functions and three member variables. The essence of these components are detailed as follows:

- **TimeWindow_Est()**: This function is the class constructor. It initializes the member variables and binds the *T_* member variable so that it could be configured from the OTcl space.

- **estimate()**: This member function implements the estimation process. It invokes the measurement object to measure the current network average load every specified interval of time and then use a time window frame to actualize the average network load. The function continually repeats this process. It does not return a value but continually actualizes the average network load.

- **change_avload()**: A member function that is used to artificially increase or decrease the network average load. It has one parameter: *incr* from type double, which represents the value by which the network average load is artificially increased or decreased. When the average network load is increased, the sampling count variable (*scnt*) is cleared to restart the time window frame. The function returns no value due to its *void* return type.

- **scnt**: A member variable that keeps the count of sampling periods to indicate the end of a time window frame.

- **T_**: A member variable to stores the time window frame value.

- **maxp**: A member varible to store the maximum network average load recorded in the last time window frame.

5.4.2.2 Exponential Averaging Estimator Code Description

This estimation mechanism is implemented as a C++ class with the name *ExpAvg_Est*. The class declaration and definitions of its member functions and member variables are found in the C++ file *ns-2/adc/expavg-est.cc*. It is derived from the *Estimator* base class and composed of two member functions and one member variable. The importance of these components are the following:

- **ExpAvg_Est()**: This is the class constructor function. It only binds the member variable *w_* so that it can be configured from OTcl space.

- **estimate()**: This function implements the estimation process. It invokes the measurement object to measure the current network average load every specified interval of time. This value is then used in an impulse response function together with the *w_* member variable and the existing average network load to estimate the actual average network laod. It does not return a value but continually actualizes the average network load.
- **w_**: A member variable that represents the averaging weight in the impulse response function.

5.4.2.3 Point Sample Estimation Code Description

The point sample estimation mechanism is realized as a C++ class with the name *PointSample_Est*. It is a child class derived from the *Estimator* base class. The class declaration and definitions of its member functions are found in *ns-2/adc/pointsample-est.cc*. This class defines only two member functions and their importance are described as follows:

- **PointSample_Est()**: This function is the class constructor. It is empty and does only the default work of allowing access to the class.
- **estimate()** This member function implements the estimation process. It invokes the measurement object to measure the current network average load every specified interval of time. It does not return a value but continually actualizes the average network load.

5.5 Summary

The new multiservice framework is composed of four components, which together realises the funtionalities of the framework. The primary aim of this framework is to simulate the effects of simultaneous transmission of three traffic classes, each with different quality of service requirement over a packet network. The components of this framework are the queue scheduler, the signalling mechanism, the enhanced link, and the admission control, which includes the estimation mechanism. The components are either implemented in C++ or in OTcl language both using the object-oriented mechanism. The classes of these components are presented in UML diagrams showing the class name, some of the class member functions and some of the class member variables for a quick overview of the class. Furthermore, some of the class member functions and member variables are explained in details.

6 Simulation Scenarios and Results

This chapter presents the simulation results of the new multiservice framework designed and implemented in this master thesis. The simulations serve as a testing methodology to validate the implemented functionalities of this framework. The framework is designed for differentiated handling of the three traffic classes simultaneously traversing over a packet network and thereby ensuring their different QoS requirements.

The multiservice framework is designed with static and dynamic bandwidth allocation mechanisms. These two mechanisms are simulated using the two-node and eight-node network topologies defined in this chapter. There are two simulation scenarios designed to fully cover and evaluate the performances of different aspects of the multiservice framework.

In all the simulations, the multiservice link is set up with a 10 Mbit/s bandwidth capacity and 1 ms propagation delay. The simulation runs for 3000 seconds. The performance for each of the four admission control algorithms introduced in chapter 3, section 3.3.2.4 is calculated by measuring the average packet delay, actual link utilization and the drop rate. These numbers are measured starting after an intial warmup period of 1500 seconds and stored in files for postprocessing at the end of the simulation. In addition gnuplot is used to plot snap shots of estimated and actual bandwidth utilized in the period between 2000 and 2200 seconds of the simulation time, and the delays experienced by the traffic packets between 2800 and 3000 seconds of the simulation time. Note that in all simulations, the measured sum algorithm is configured with a 95 percent utilization factor, which means that the algorithm cannot achieve a total network utilization greater than this factor.

This chapter is divided into four sections, the first section covers the simulation of the static bandwidth allocated version of the multiservice framework. It simulates the two scenarios using each of the network topology type for the four measurement-based admission control algorithms. The second section simulates the dynamic bandwidth allocated version of this framework. It also simulates two scenarios using each of the network topology types for the MBAC algorithms. The achieved performance results are also compared. The third section presents some performance comparisons between the static and dynamic bandwidth allocated version of the multiservice framework. The last section summarizes the chapter.

6.1 Static Bandwidth Allocation Mechanism

The multiservice framework supports the transmission of three different traffic classes (VoIP, video, and best effort). In order to achieve different quality of service demands of these traffic classes, the total network bandwidth is shared among the traffic classes according to the class priorities. In the static version of the multiservice framework, the bandwidth sharing is static. i.e., it is done once and each class operates only with its allocated class bandwidth. When there is little or no traffic flows for the best effort class, its class bandwidth is wasted and can't be borrowed by the other higher priority classes that might require of extra bandwidth.

The simulation of this static bandwidth allocated version is carried out for two types of network topologies using two simulation scenarios for each as described in the subsections below.

6.1.1 Simulation with the Two-Node Network Topology

The first network topology type is made up of two nodes and the multiservice bottle-neck link connecting them. The traffic agents and the traffic application source generators are connected directly to the first node (*Node 0*), and the receiver agents and the traffic application sinks are directly connected to the second node (*Node 1*) as shown in Figure 6.1.

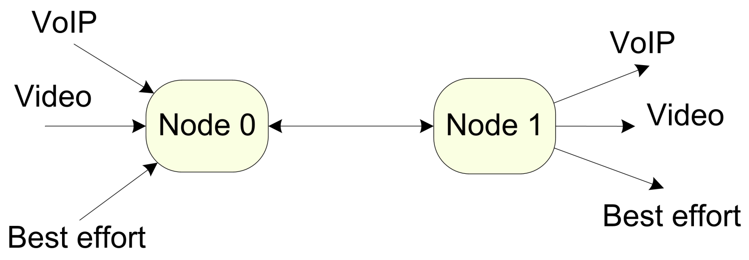


Figure 6.1: Two-node network topology

The VoIP, video and best effort application traffic sources generate data packets, which are sent through node one over the 10 Mbit/s bottleneck link to their respective sink applications. The bottleneck link is where the quality of service is ensured by executing different operations like data packets classification, network load estimations, delay measurements, and packet lost measurements.

To simulate the traffic of the three applications types in ns-2, the different application traffic generators defined in ns-2 (see ns manual [20]) are used. The VoIP application traffic is represented by the constant bit rate source traffic type, the video application traffic is represented by video trace source traffic type, and the best effort application traffic is represented by the exponential traffic source type.

In order to fully test this static bandwidth allocated version, two simulation scenarios are created using the above mentioned source traffic types and their different configuration parameters. They are presented together with the achieved performance results of the MBAC algorithms in the two subsections below.

6.1.1.1 Simulation Scenario One

The first simulation scenario for the static bandwidth allocated version using the two-node network topology is illustrated in Table 6.1.

CBR Traffic		**Video Trace**	**Exponential Traffic**			
Packet size (byte)	Rate (kbit/s)	Target rate (kbit/s)	Packet size (byte)	Burst time (ms)	Idle time (ms)	Rate (kbit/s)
125	200	256	125	313	325	64

Table 6.1: Simulation scenario one

This simulation scenario represents a standard situation where the different class bandwidths are good utilized by their respective application traffic. In Table 6.1, *packet size* means the traffic packet size, *rate* means the traffic sending rate, *target rate* means the target rate for the video traces, *burst time* means the *On* time of the exponential traffic, and *idle time* means the *Off* time of the exponential traffic.

The results of the first simulation scenario using the two-node network topology are presented for the four MBAC algorithms as follows:

Measured Sum
The performance results achieved by the measured sum algorithm are displayed in Table 6.2 and in Figure 6.2.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14509310	1812059980	62460	9.347	0.332	93.03
Video	282837	1176755809	0	5.781		
Best effort	4036428	502916760	0	1.845		

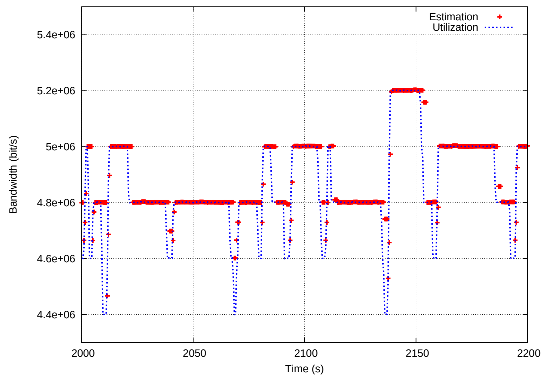
Table 6.2: Measured sum performance results

Table 6.2, illustrates the simulation results obtained by the three different traffic classes. The column headers in the table are described as follows:

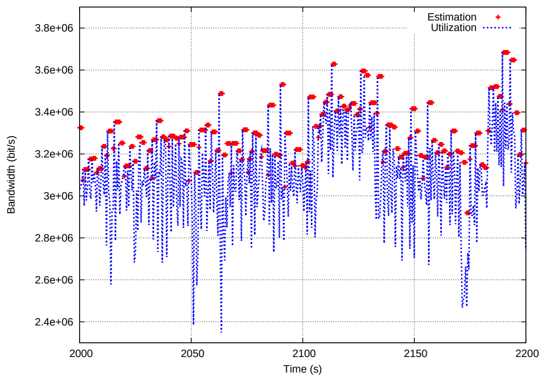
- *Received packets*: This is the number of packets received for each traffic class.
- *Received packets size*: This is the size in byte of the total received packets for each traffic class.
- *Lost packets*: Shows the number of packets lost by each traffic class.
- *Average packet delay*: This represents the total delay suffered by a packet on the outgoing link due to queueing and transmission including the processing time at the node, and the propagation delay of the link averaged over a number of measurements.
- *Total Drops*: This represents the percentage of the total packets dropped in the bottleneck link.
- *Total utilization*: Defined as a fraction of the time during which the outgoing link queue was not empty. It represents the percentage of the total network utilizations achieved by the traffic classes.

The received packet column in Table 6.2 shows which traffic occupies larger portions of the network link. The VoIP traffic has highest priority and the largest bandwidth share followed by the video traffic, and the best effort traffic has the least priority and smallest bandwidth share. The results also show that the algorithm achieved high total network utilization and low packet drop rate.

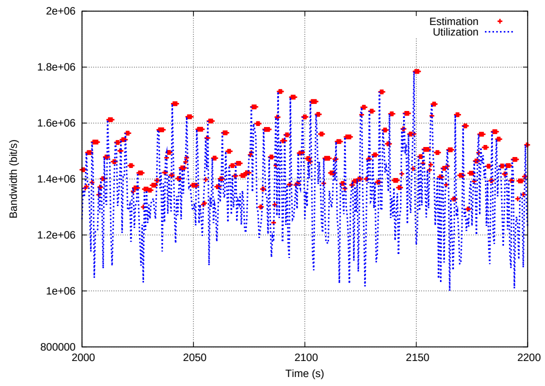
Figure 6.2 displays the estimated and actual utilized average network load for each of the traffic classes and the delay experienced by the traffic flows.
Figures 6.2(a), 6.2(b), and 6.2(c) illustrate how good the time-window estimator could estimate the actual network utilization for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the graphics, it is noticable that there is a better estimation and actual utilization correspondances for VoIP traffic flows as compared with that of video traffic flows. The figures also show the degree of utilization of the respective allocated class bandwidths. Figure 6.2(d) shows the delays experienced by VoIP, video, and best effort traffic flows. It can be observed from the graphic that the traffic classes show variable delay behaviours but their highest delays are still below the maximum acceptable delay thresholds for their respective applications.



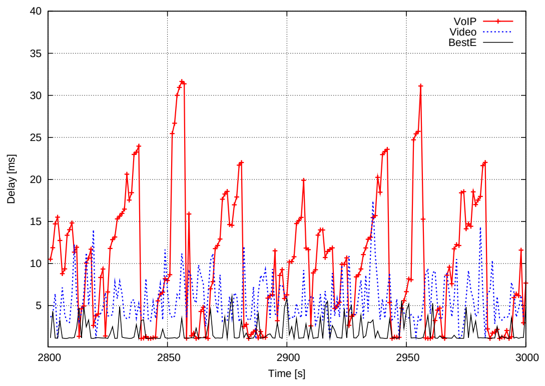
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video, and best effort packets delay behaviours

Figure 6.2: Actual and estimated utilization: MS algorithm; two-node topology; scenario one

Hoeffding Bounds

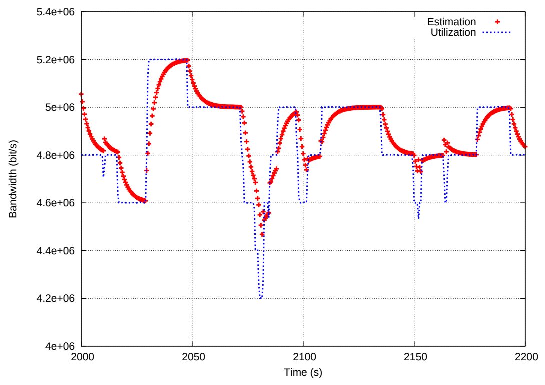
The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.3 and in Figure 6.3.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14113472	1762579810	14397	6.773	0.0816	88.54
Video	267972	1101984148	0	5.617		
Best effort	3269366	407041360	0	1.804		

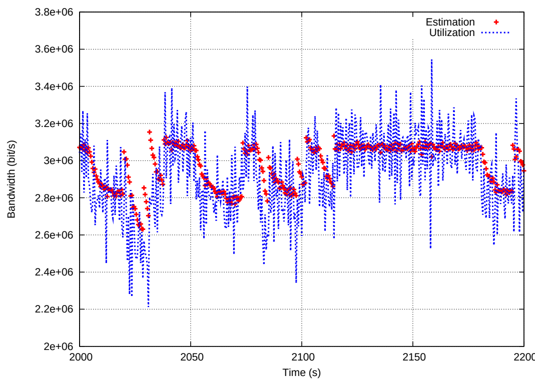
Table 6.3: Hoeffding bounds performance results

The algorithm achieves high network utilisation performance and low drop rates. The most drops packets are from VoIP class which occupies the largest portion of the network. Figure 6.3 depicts the estimation behaviours of the exponential averaging estimator and the admittance behaviours of the hoeffding bounds admission control algorithm.

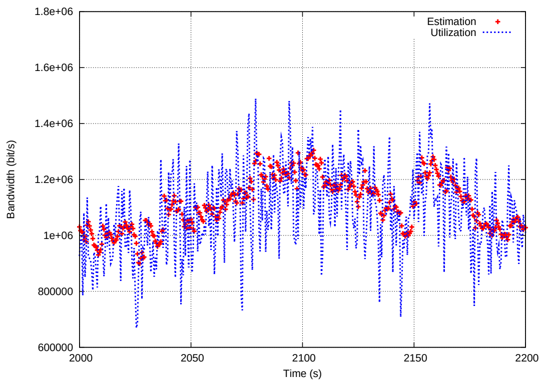
Figures 6.3(a), 6.3(b), and 6.3(c) illustrate how good the exponential averaging estimator could estimate the actual network utilization for the hoeffding bounds



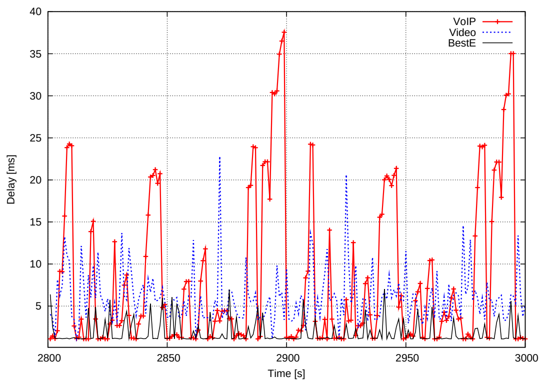
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.3: Actual and estimated utilization: HB algorithm; two-node topology; scenario one

algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the graphics, the exponential averaging achieves better estimations of the actual utilizations for VoIP traffic flows as for video and best effort traffic flows. They also show the degree of utilization of the respective allocated class bandwidths. Figure 6.3(d) shows the delay behaviours of VoIP, video and best effort traffic flows. The delay behaviours varies for each of the traffic classes but they are below the maximum delay tolerable by their respective applications.

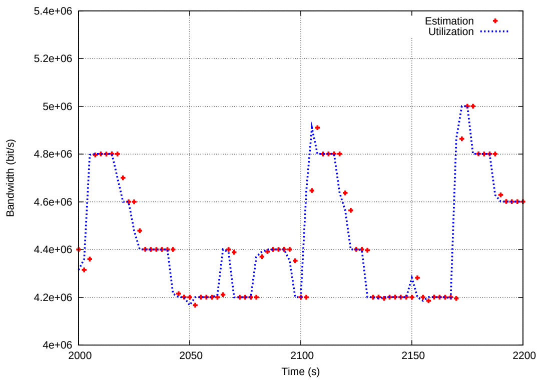
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.4 and in Figure 6.4.

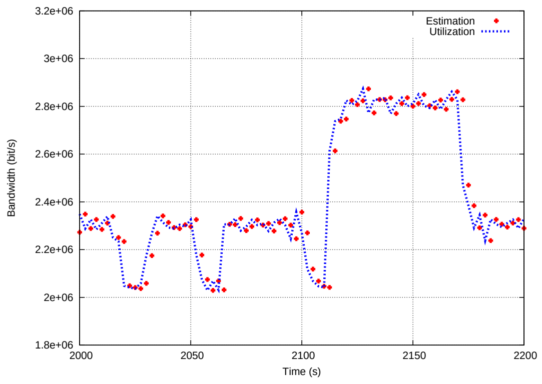
The algorithm obtains good total utilization performance while maintaining low drop rate. From the results, it is the most strict algorithm in VoIP packet admission as compared to the other algorithms. This behaviour is obvious with the number of packets it accepts for this class. Figure 6.4 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	13562102	1693660030	66759	6.076	0.3835	83.55
Video	237437	984322558	0	5.209		
Best effort	3606468	449179215	0	1.759		

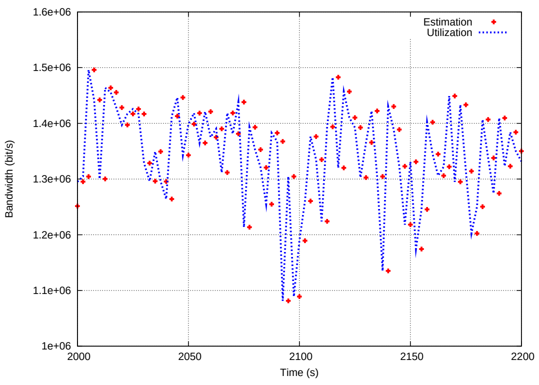
Table 6.4: Acceptance region tangent at origin performance results



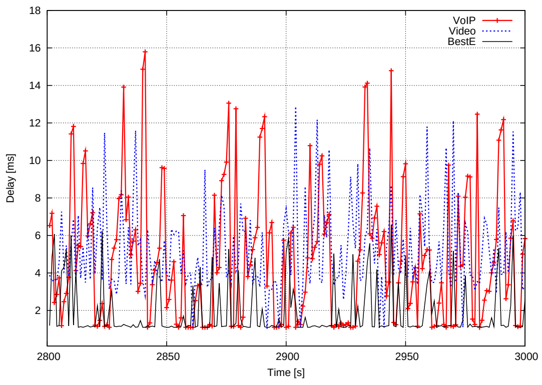
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.4: Actual and estimated utilization: ACTO algorithm; two-node topology; scenario one

Figures 6.4(a), 6.4(b), and 6.4(c) illustrate how good the point sample estimator could estimate the actual network utilization for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As shown in the graphics, the VoIP and video classes show good correspondances of the estimation and actual utilizations as compared to the best effort traffic class. They also show the degree of utilization of the respective allocated class bandwidths. Figure 6.4(d) displays the delay behaviours of VoIP, video and best effort traffic flows. The delay behaviours are not constant but are still below the maximum tolerable delay thresholds for their applications.

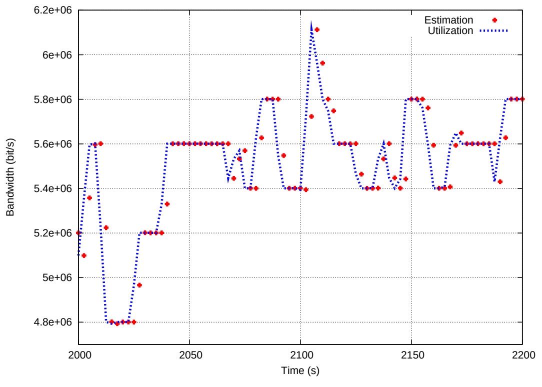
Acceptance Region Tangent at Peak

The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.5 and in Figure 6.5.

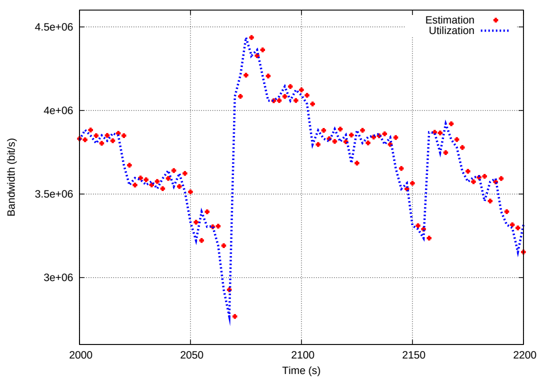
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	16347350	2041807420	1374799	22.074	6.7398	99.13
Video	321800	1359630466	0	5.448		
Best effort	3729278	464524900	0	1.939		

Table 6.5: Acceptance region tangent at peak performance results

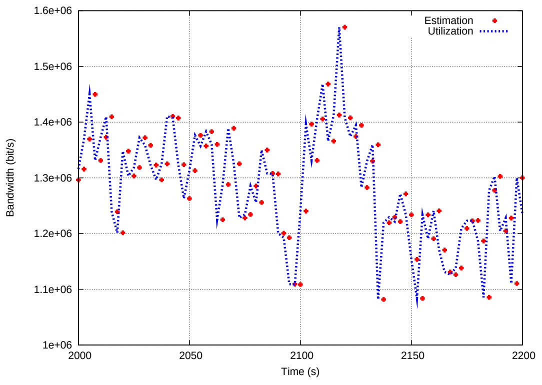
This algorithm achieved the highest total network utilization as compared to the others for this scenario. But it also suffers the highest delays for VoIP traffic flows. Figure 6.5 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm.



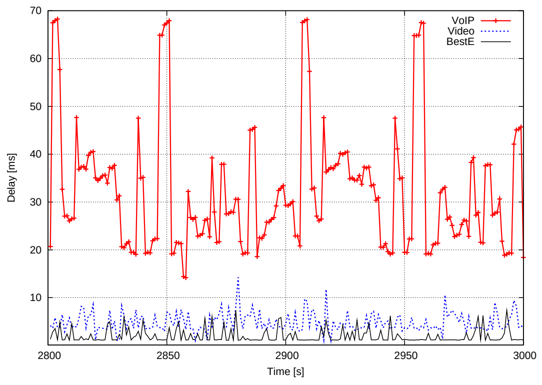
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.5: Actual and estimated utilization: ACTP algorithm; two-node topology; scenario one

Figures 6.5(a), 6.5(b), and 6.5(c) illustrate how good the point sample estimator could estimate the actual network utilization for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets in a packet network. According to the graphics, it is noticable that the estimator does good estimations of the actual utilizations for the traffic classes. The graphics also show the degree of utilization of the respective class bandwidth. Figure 6.5(d) illustrates the delay behaviours of VoIP, video and best effort traffic flows. It is very obvious from the figure that the VoIP class exhibits the highest delay. This is caused by the excess accaptance of VoIP traffic by the algorithm. But the delays are still below the maximum tolerable delay thresholds for their applications.

6.1.1.2 Simulation Scenario Two

The second simulation scenario is presented in Table 6.6. In this scenario, the best effort applications are purposely set to transmit packets at a low sending rate to investigate the effects of underutilized class bandwidth in the total network utilization performance.

CBR Traffic		**Video Trace**	**Exponential Traffic**			
Packet size (byte)	Rate (kbit/s)	Target rate (kbit/s)	Packet size (byte)	Burst time (ms)	Idle time (ms)	Rate (kbit/s)
125	200	256	125	313	325	6

Table 6.6: Simulation scenario two

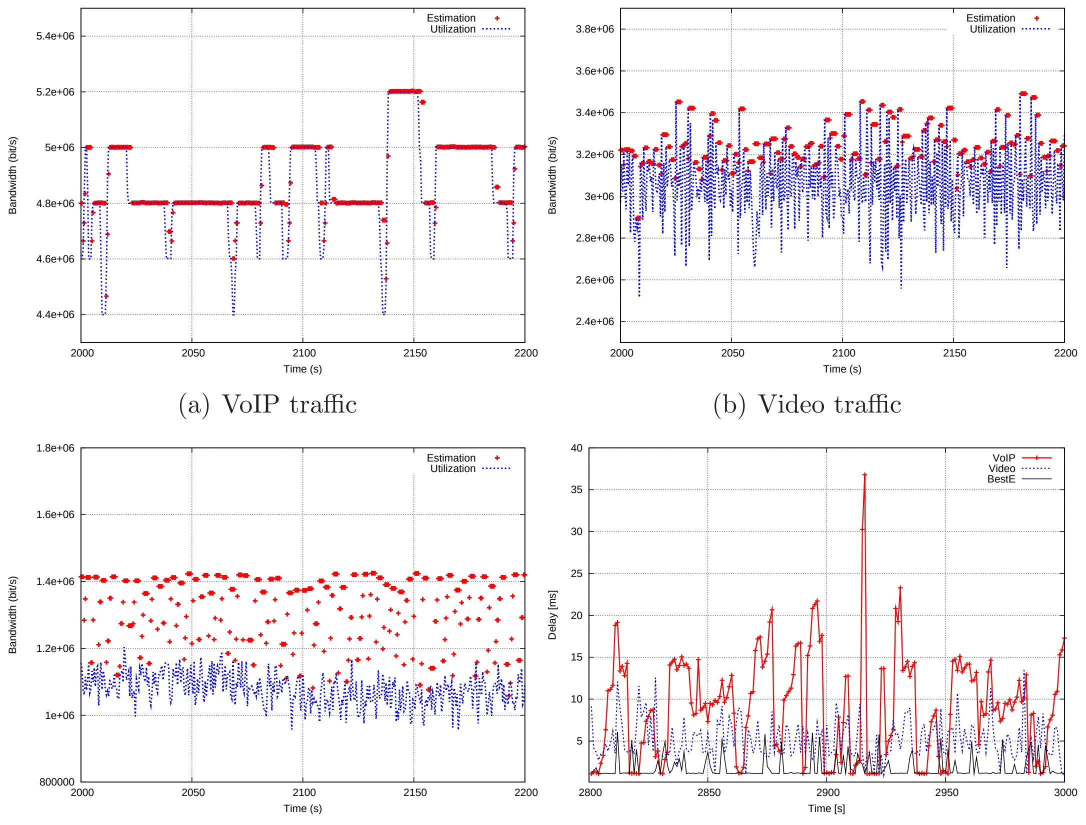
The reason for setting only the best effort applications to send at low rate is based on the assumption that the higher priority VoIP and video applications are of higher demand and more important to the network, i.e., both applications make up more than 80% of the network traffic. The low sending rate of best effort application causes under utilization of its class bandwidth. Thus, this scenario shows how a class bandwidth that could be used by other traffic classes, is wasted in a network where bandwidth is statically shared among traffic classes. This consequently leads to a poor total network utilization level.

Below are the simulation results achieved by the four MBAC algorithms for the three traffic classes.

Measured Sum
The performance results achieved by the measured sum for this scenario are displayed in Table 6.7 and in Figure 6.6.
Inspite of the underutilized best effort class bandwidth, the algorithm achieves high total network utilization level and lower drop rates as compared to scenario one. Figure 6.6 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this scenario.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14523736	1813863335	28462	7.662	0.158	90.74
Video	283501	1172753596	0	5.227		
Best effort	3180903	395665860	0	1.818		

Table 6.7: Measured sum performance results



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.6: Actual and estimated utilization: MS algorithm; two-node topology; scenario two

Figures 6.6(a), 6.6(b), and 6.6(c) illustrate how good the time-window estimator could estimate the actual network utilization for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the figures, it is clearly recogniseable that the best effort class bandwidth is underutilized following from the large difference between the estimations and the actual network utilizations. Figure 6.6(d) displays the packet delay behaviours of VoIP, video and best effort traffic. From the figure, the traffic delay behaviours varies but they are still below the maximum tolerable delay thresholds for their applications.

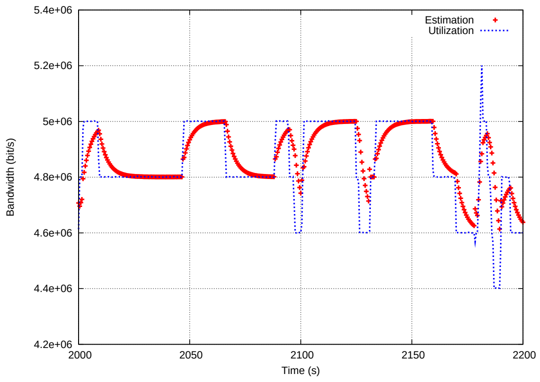
Hoeffding Bounds

The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.8 and in Figure 6.7.

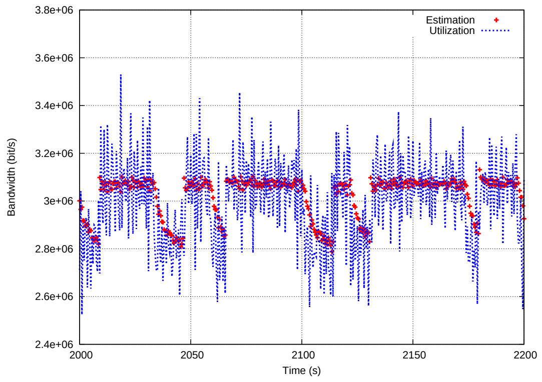
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14343716	1791360835	2236	4.456	0.0148	80.55
Video	267302	1108477294	0	5.115		
Best effort	546807	66699015	0	1.724		

Table 6.8: Hoeffding bounds performance results

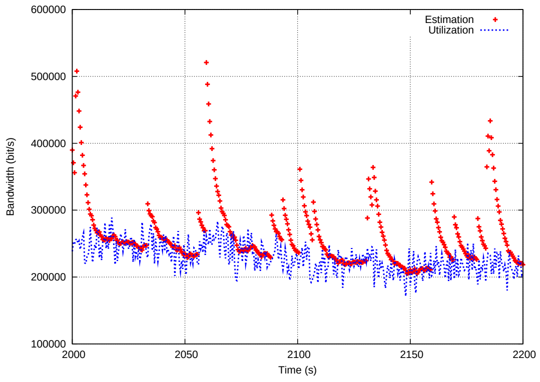
The algorithm achieves good total network utilization and low drop rate for this scenario. Although the influence of the low utilized best effort class bandwidth is seen at the total utilization obtained as compared to the first scenario. Figure 6.7 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this scenario.



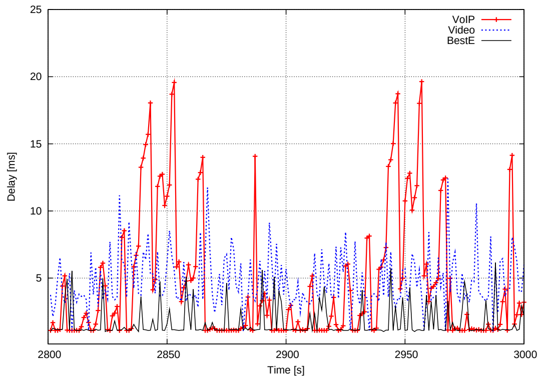
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.7: Actual and estimated utilization: HB algorithm; two-node topology; scenario two

Figures 6.7(a), 6.7(b), and 6.7(c) illustrate how good the exponential averaging estimator could estimate the actual network utilization loads for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the figures, it is observable that the estimator does better estimations for VoIP traffic flows as for video traffic flows. The graphic also shows the underutilization of the best effort class bandwidth. Figure 6.7(d) displays the delay behaviours of VoIP, video and best effort traffic flows. The delay behaviours shown in the figure vary for each of the traffic classes but they fall below the maximum tolerable delay thresholds required by their applications.

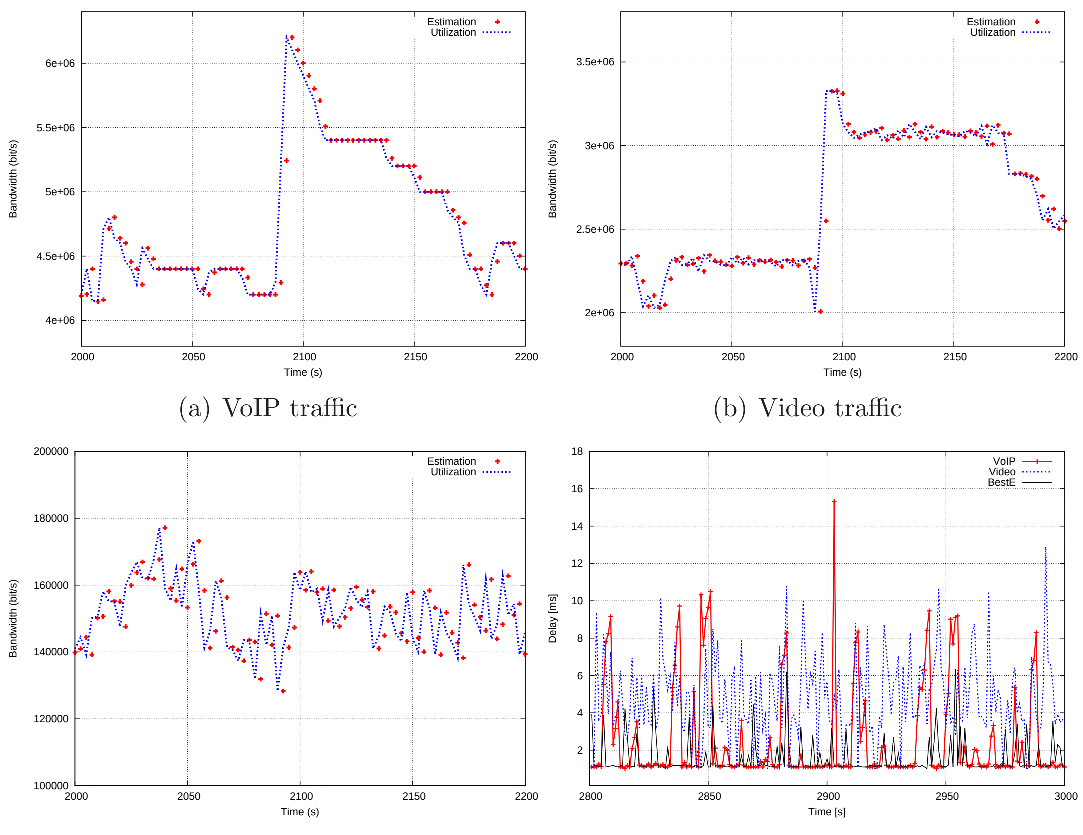
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.9 and in Figure 6.8.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	13548345	1691942190	4875	3.366	0.0343	72.85
Video	240608	995685638	0	5.011		
Best effort	420073	50871125	0	1.610		

Table 6.9: Acceptance region tangent at origin performance results

The algorithm achieves acceptable total utilization and better drop rates as compared to scenario one. Figure 6.8 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm during the simulation of the second scenario.
Figures 6.8(a), 6.8(b), and 6.8(c) illustrate how good the point sample estimator could estimate the actual network utilization loads for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the figures, it is observable that the estimator does good estimations of the actual network utilizations for the traffic classes. It is also oberservable that the best effort class bandwidth is clearly under utilized, the application can only make use of about 180 kbit/s bandwidth while the class has a 1.5 Mbit/s allocated bandwidth. Figure 6.8(d) displays the delay behaviours of VoIP, video and best effort traffic flows. The variance in delays for all traffic classes are noticeable from the figure, but the highest delay observed is within range of tolerable delay thresholds for their applications.



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.8: Actual and estimated utilization: ACTO algorithm; two-node topology; scenario two

Acceptance Region Tangent at Peak

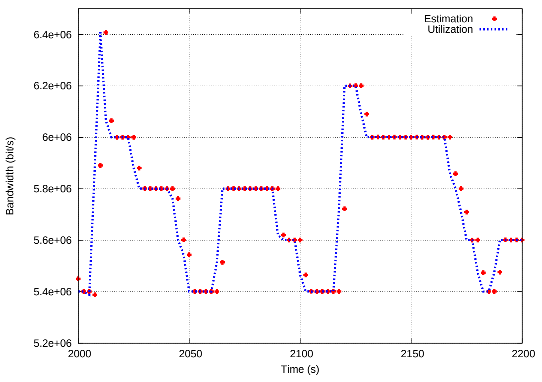
The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.10 and in Figure 6.9.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	16834594	2102714390	213454	11.947	1.1976	95.16
Video	322294	1357666381	0	5.465		
Best effort	666200	81612220	0	1.889		

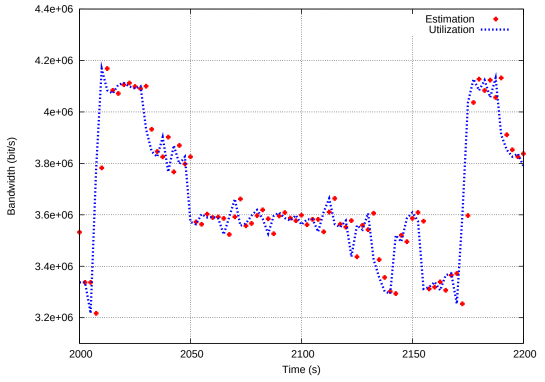
Table 6.10: Acceptance region tangent at peak performance results

The algorithm achieves much lower drop rates and still high total network utilization level as compared to the first simulation scenario. Figure 6.9 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.

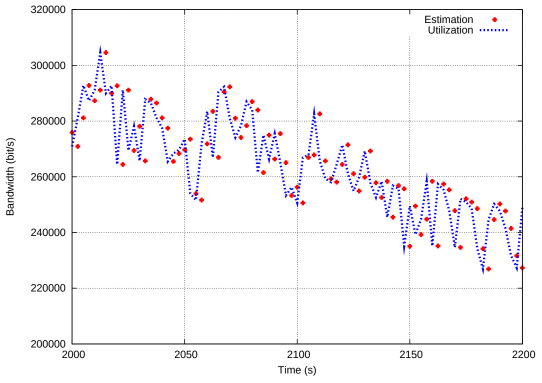
Figures 6.9(a), 6.9(b), and 6.9(c) illustrate how good the point sample estimator could estimate the actual network utilization loads for the acceptance



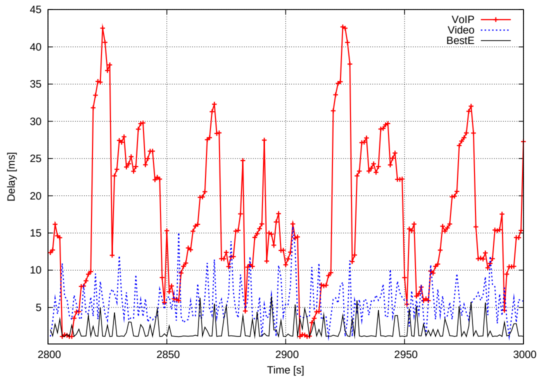
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.9: Actual and estimated utilization: ACTP algorithm; two-node topology; scenario two

region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets in a packet network. It is noticeable from the figures that the best effort class bandwidth is poorly utilized. It could only make use of about 300 kbit/s bandwidth out of its class for which 1.5 Mbit/s bandwidth capacity is allocated. They also show the degree of utilization of the other class bandwidth. Figure 6.9(d) shows the delay behaviours of VoIP, video and best effort traffic. From the figure, the delays experienced by the traffic classes varies but they are still below tolerable boundaries for such applications. It is also interesting to notice that the average delay suffered by VoIP traffic is 50 % lower as in scenario one.

6.1.2 Simulation with the Eight-Node Network Topology

The second network topology type is made up of eight nodes connected as shown in Figure 6.10. The traffic agents for VoIP, video and best effort and their

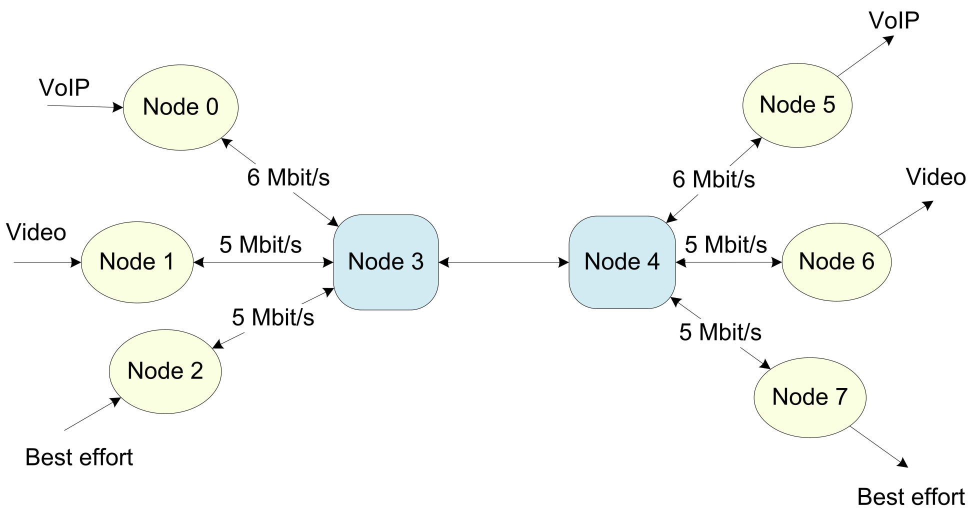


Figure 6.10: The eight-node network topology

application traffic generators are each attached to a different node. The three source nodes send the application traffic through node 3 which acts as a router, to their respective sink agents on the other end, which are also attached to different destination nodes. The three source nodes are each connected to the router node with a *duplex-link*. The link connecting VoIP traffic to the router node is made up of 6 Mbit/s bandwidth capacity, 1 ms propagation delay, and *DropTail* queueing mechanism, while the links connecting video and best effort traffic are each made up of 5 Mbit/s bandwidth capacity, 1 ms propagation delay, and also *DropTail* queueing mechanism. These link bandwidths capacities are set bigger than the biggest class bandwidth share in the multiservice link, thereby ensuring insignificant packet lost between the source node and the router node. At the other end, *duplex-links* with the same parameters are also used to connect router node 4 with the respective destination nodes.

Similarly as for the two-node network topology in subsection 6.1.1, two simulation scenarios are performed for the eight-node topology too. The simulation scenarios are designed in this way to create a common basis for comparing the performances of the two network topologies.

6.1.2.1 Simulation Scenario One

The first simulation scenario is exactly the same as the scenario presented in subsection 6.1.1.1. That means the same application traffic source generators and their defined parameters are used for this simulation too. This simulation scenario is used to simulate the four measurement-based admission control algorithms and their achieved performance results are each presented below.

Measured Sum

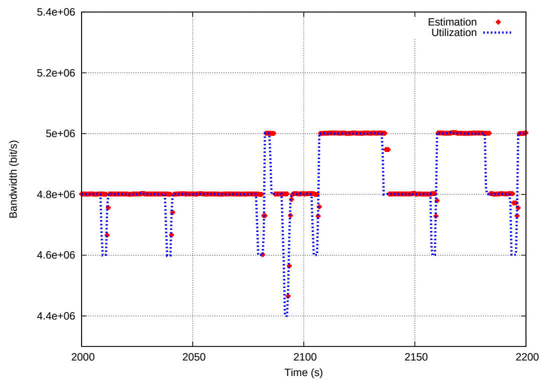
The performance results achieved by the measured sum for this scenario are displayed in Table 6.11 and in Figure 6.11.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14512053	1812445905	66410	9.867	0.3526	93.14
Video	281277	1173995547	0	5.670		
Best effort	4041399	503535930	0	1.876		

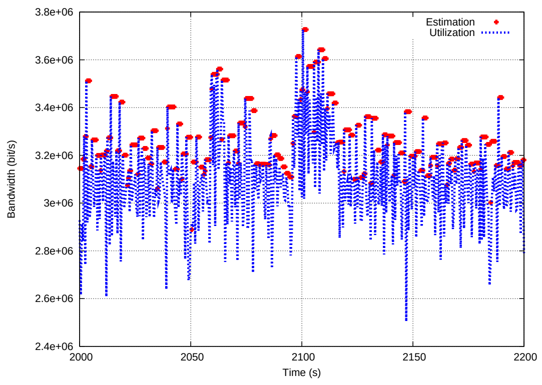
Table 6.11: Measured sum performance results

The algorithm achieved a very good total network utilization level, considering the fact that the utilization factor is configured for 95 percent utilization. Figure 6.11 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario using the eight-node network topology.

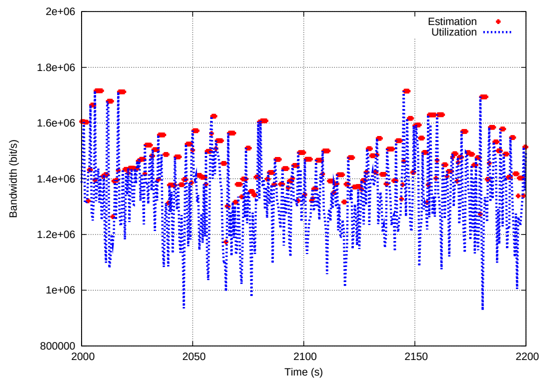
Figures 6.11(a), 6.11(b), and 6.11(c) illustrate how good the time-window estimator could estimate the actual network utilization for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. The figures show better estimations of actual utilization for the VoIP traffic as for video and best effort. They also show the degree of utilization of the respective class bandwidth. Figure 6.11(d) depicts the delays behaviours experienced by VoIP, video, and best effort traffic packets. As shown on the figure, the VoIP and video delays vary much more as compared to the best effort delays. But all the delays are below the maximum tolerable boundaries for their respective applications.



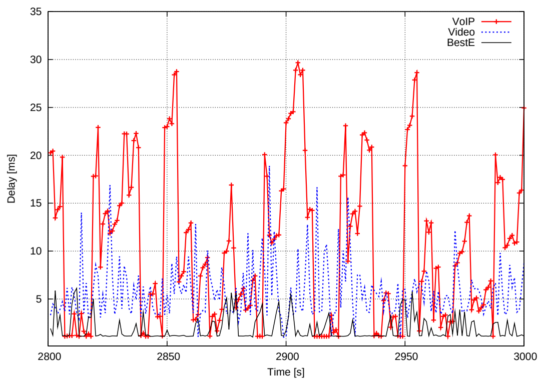
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.11: Actual and estimated utilization: MS algorithm; eight-node topology; scenario one

Hoeffding Bounds

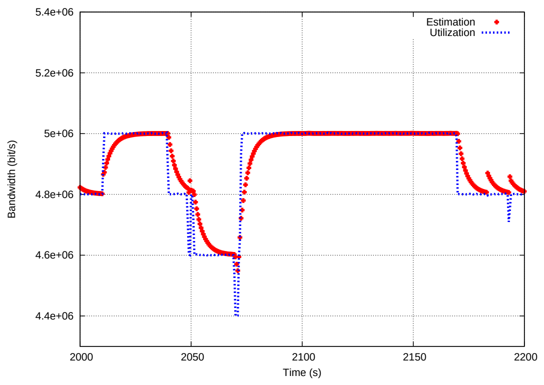
The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.12 and in Figure 6.12.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14051663	1754878255	15226	6.623	0.0865	88.84
Video	265638	1110417684	0	5.309		
Best effort	3285906	409108860	0	1.806		

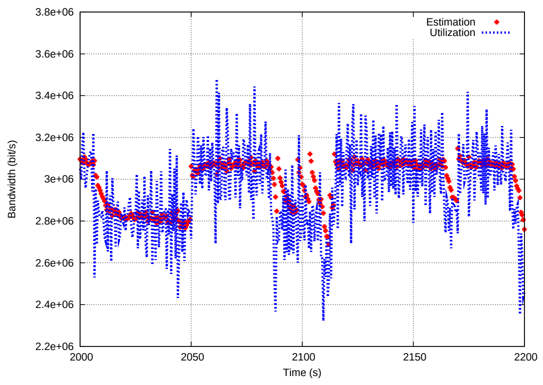
Table 6.12: Hoeffding bounds performance results

The algorithm achieved high network utilization level while maintaining low drop rate. Figure 6.12 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.

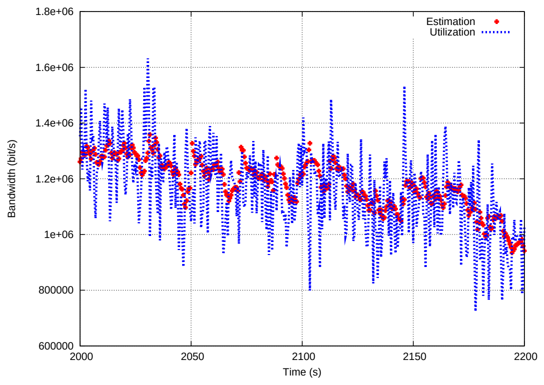
Figures 6.12(a), 6.12(b), and 6.12(c) illustrate how good the exponential averaging estimator could estimate the actual network utilization for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort application



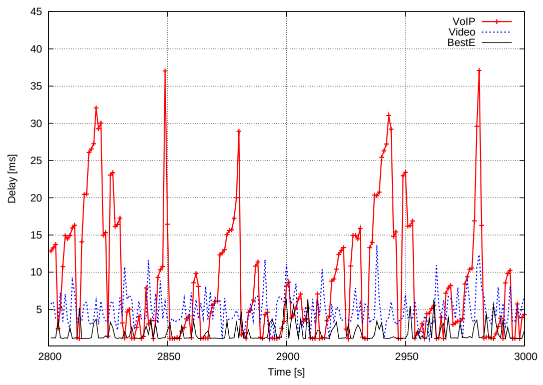
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.12: Actual and estimated utilization: HB algorithm; eight-node topology; scenario one

packets in a packet network. As shown in the figure, the estimator does better estimations of VoIP traffic actual utilizations as for video and best effort traffic estimations. The figures also show the degree of utilization of the respective class bandwidths. Figure 6.12(d) displays the delay behaviours of VoIP, video and best effort traffic flows. From the figure, the delay behaviours of the traffic classes are not constant but the highest delay behaviours of each class are below the tolerable boundaries for their respective applications

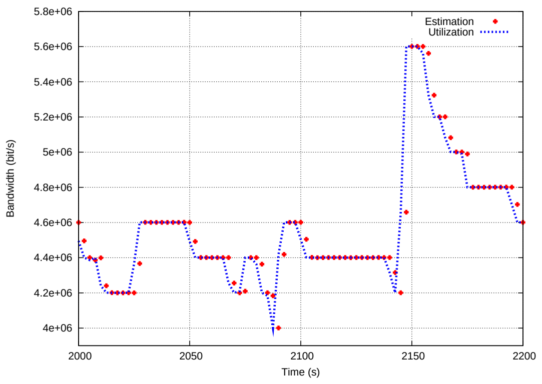
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.13 and in Figure 6.13.

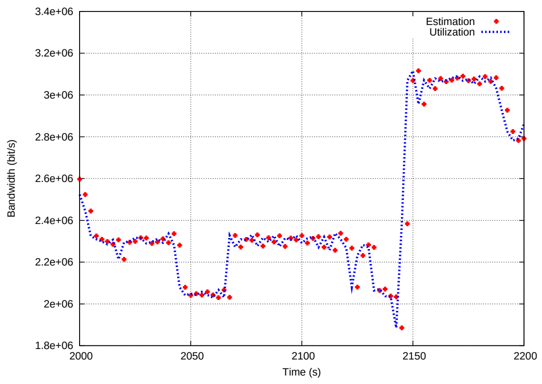
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	13510478	1687227400	87462	3.523	0.5053	83.08
Video	244863	1020296399	0	4.846		
Best effort	3553177	442515845	0	1.659		

Table 6.13: Acceptance region tangent at origin performance results

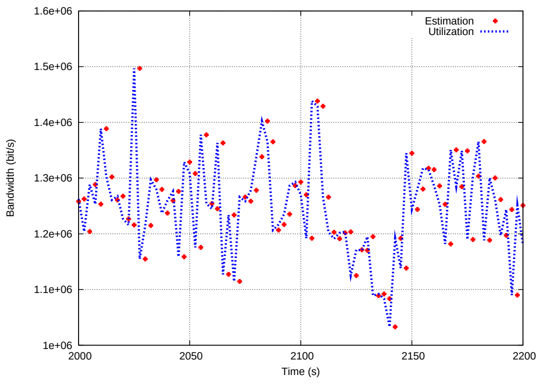
According to Table 6.13, the algorithm obtaines good network utilization level while maintaining low drop rates. Figure 6.13 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm for this simulation scenario.



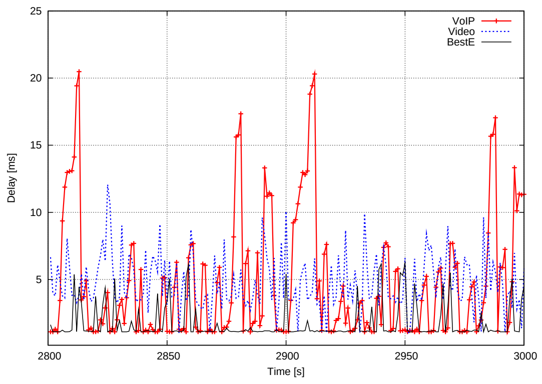
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.13: Actual and estimated utilization: ACTO algorithm; eight-node topology; scenario one

Figures 6.13(a), 6.13(b), and 6.13(c) illustrate how good the point sample estimator could estimate the actual network utilization levels for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets in a packet network. From the figures, the estimator does better estimations of actual utilizations for VoIP and video traffic flows as for best effort traffic flows. The figures also show the degree of utilization of the respective class bandwidths. Figure 6.13(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown in the figure, the traffic flows exhibits variable delay behaviours but their highest delays are still below the tolerable boundaries for such applications.

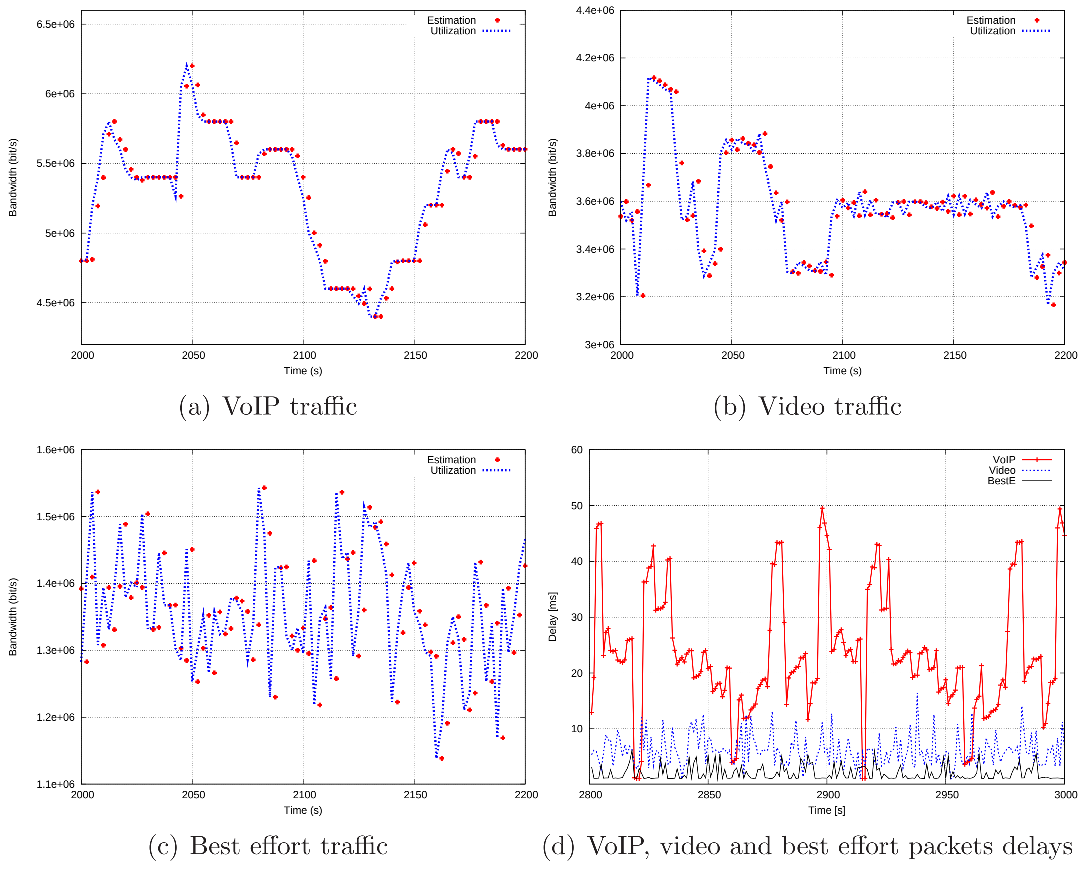
Acceptance Region Tangent at Peak

The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.14 and in Figure 6.14.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	16296295	2035526345	1294828	22.846	6.3625	99.01
Video	320631	1348950565	0	5.719		
Best effort	3734140	465130760	0	1.982		

Table 6.14: Acceptance region tangent at peak performance results

As shown in Table 6.14, this algorithm achieves the highest network utilization level while suffering the highest drop rates as compared to the other algorithms for this scenario. Figure 6.14 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.



(a) VoIP traffic (b) Video traffic (c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.14: Actual and estimated utilization: ACTP algorithm; eight-node topology; scenario one

Figures 6.14(a), 6.14(b), and 6.14(c) illustrate how good the point sample estimator could estimate the actual network utilization levels for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets in a packet network. The figures show better estimations for VoIP and video traffic flows as for best effort traffic flows. They also show the degree of utilization of the respective class bandwidths. Figure 6.14(d) illustrates the delay behaviours of VoIP, video and best effort traffic flows. The figure depicts variable delays for the traffic flows. The VoIP traffic shows the highest delay behaviours but all the traffic flow delays are still below the tolerable boundaries for such applications.

6.1.2.2 Simulation Scenario Two

The second simulation scenario using the eight-node network topology is defined with the same application source traffic generators and their parameters as the simulation scenario presented in Table 6.6, subsection 6.1.1.2. This simulation scenario is used to simulate the four MBAC algorithms and their obtained performance results are presented below.

Measured Sum

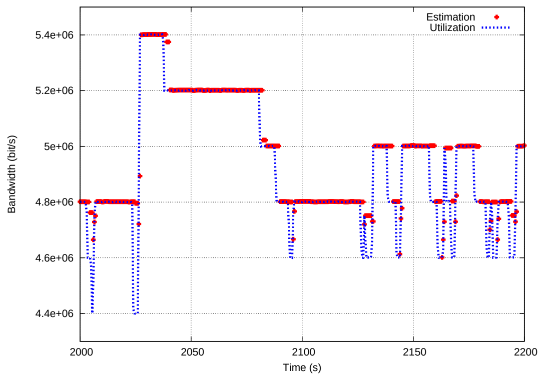
The performance results achieved by the measured sum for this scenario are displayed in Table 6.15 and in Figure 6.15.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14517540	1813128945	31041	7.627	0.1727	90.68
Video	281654	1173803884	0	5.072		
Best effort	3171237	394455930	0	1.837		

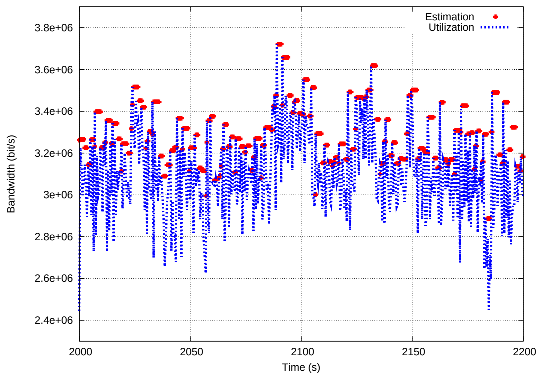
Table 6.15: Measured sum performance results

As shown in Table 6.15, the algorithm obtained high total utilization level with low drop rate inspite of the poor utilization of the best effort class bandwidth. Figure 6.15 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario using the eight-node network topology.

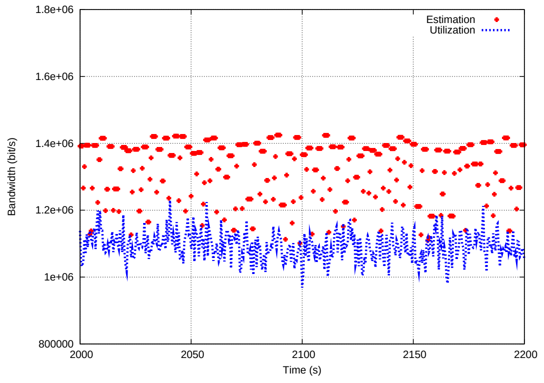
Figures 6.15(a), 6.15(b), and 6.15(c) illustrate how good the time-window estimator could estimate the actual network utilization for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As can be observed from the figures, the estimator does good estimations of actual utilizations for VoIP traffic flows as for video and best effort traffic flows. It is also clearly noticeable that the best effort class bandwidth is poorly utilized. Figure 6.15(d) shows the delay behaviours experienced by VoIP, video, and best effort traffic packets. As shown in the figure, the delays display high variances among the measurements, but the highest delay behaviours of



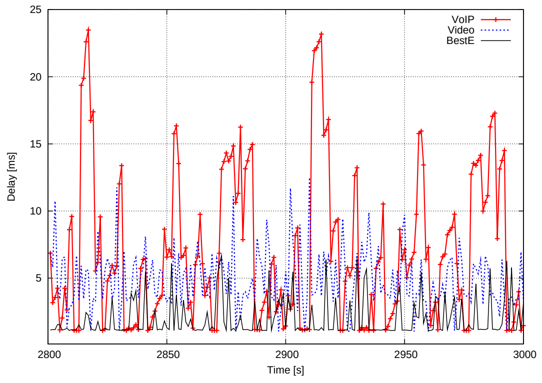
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.15: Actual and estimated utilization: MS algorithm using eight-node topology for scenario two

each traffic class is below the maximum tolerable delay thresholds defined for their applications.

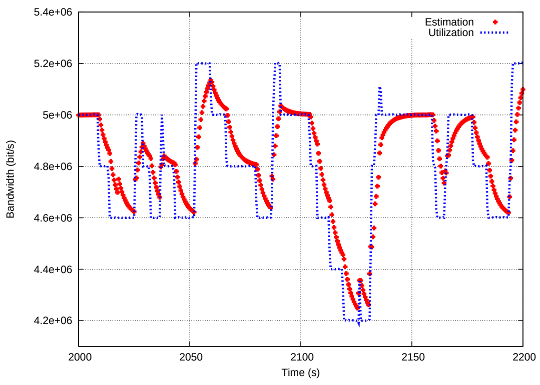
Hoeffding Bounds

The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.16 and in Figure 6.16.

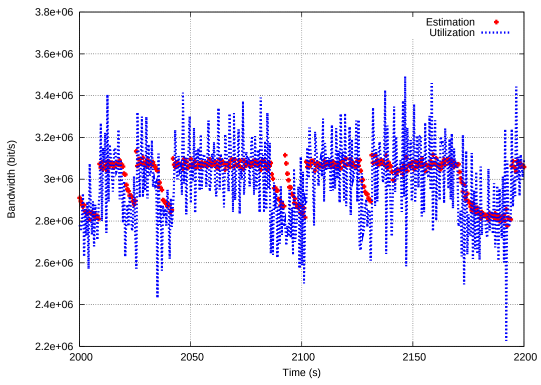
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	14254447	1780221635	1898	4.747	0.0126	80.28
Video	264183	1111463333	0	5.638		
Best effort	567712	69308465	0	1.729		

Table 6.16: Hoeffding bounds performance results

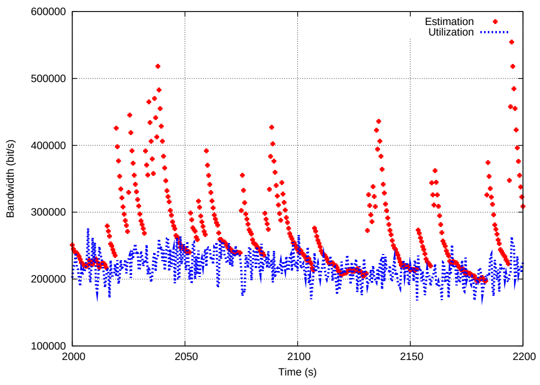
As presented in Table 6.16, the algorithm obtaines good total utilization level while keeping low drop rates inspite of the poor utilized best effort class bandwidth. Figure 6.16 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.



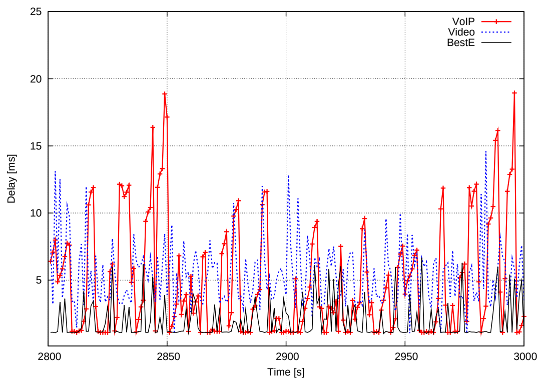
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.16: Actual and estimated utilization: HB algorithm; eight-node topology; scenario two

Figures 6.16(a), 6.16(b), and 6.16(c) illustrate how good the exponential averaging estimator could estimate the actual network utilization levels for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As displayed in the figures, the estimator does better estimations of actual utilizations for VoIP traffic flows as for video and best effort traffic flows. It is also clearly observable from the figures that the best effort class bandwidth is poorly utilized. The best effort actual utilization is below 300 kbit/s while the class has allocated bandwidth of 1.5 Mbit/s capacity. Figure 6.16(d) displays the packet delay behaviours of VoIP, video and best effort traffic. According to the figure, the delay behaviours are not constant but the highest delays are still below the tolerable boundaries for their respective applications.

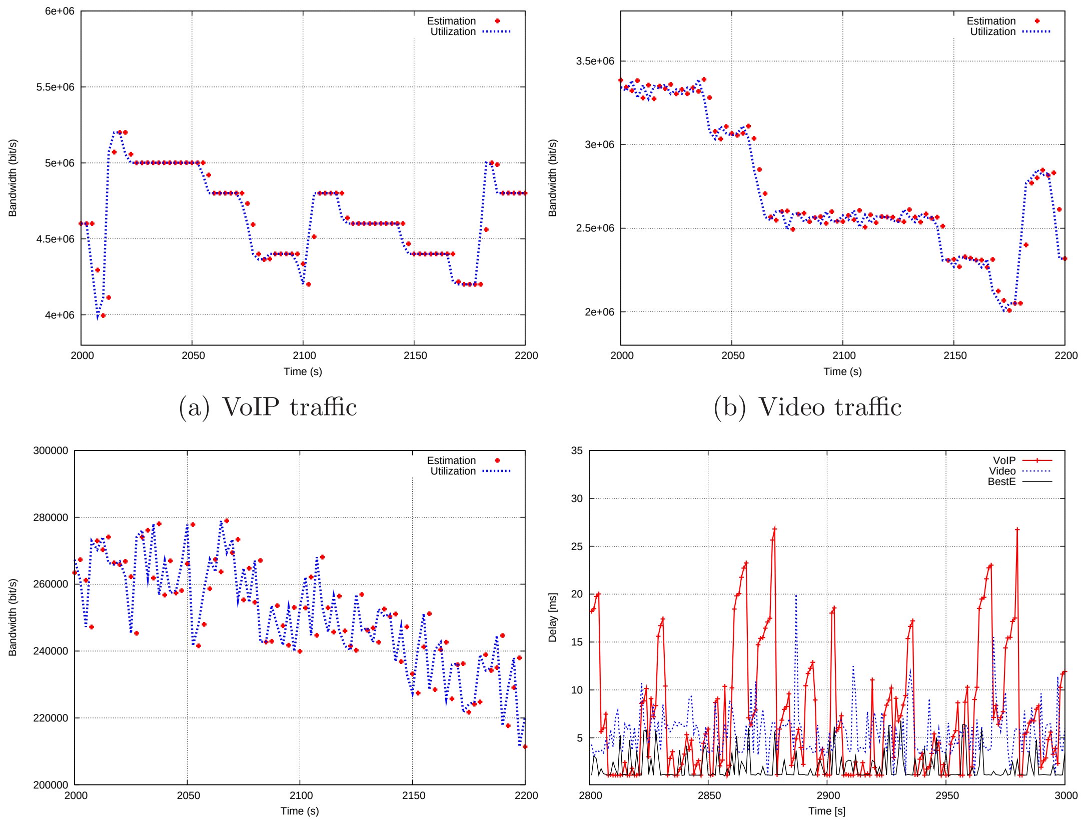
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.17 and in Figure 6.17.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	13598159	1698183955	19197	5.063	0.1328	73.94
Video	251227	1043418543	0	5.118		
Best effort	601993	73593695	0	1.771		

Table 6.17: Acceptance region tangent at origin performancs results

According to Table 6.17, the algorithm achieved acceptable network utilization level while keeping low drop rate regardless of the poor utilized best effort class bandwidth. Figure 6.17 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm for this simulation scenario.



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.17: Actual and estimated utilization: ACTO algorithm; eight-node topology; scenario two

Figures 6.17(a), 6.17(b), and 6.17(c) illustrate how good the point sample estimator could estimate the actual network utilization levels for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets in a packet network. The figures show relatively good estimations of the actual utilizations for the three traffic classes. The low utilization of the best effort class bandwidth is also observable in the figure. Figure 6.17(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown in this figure, the traffic classes experiences variable delay behaviours but their highest delays are still below the maximum tolerable delay boundaries for their applications.

Acceptance Region Tangent at Peak

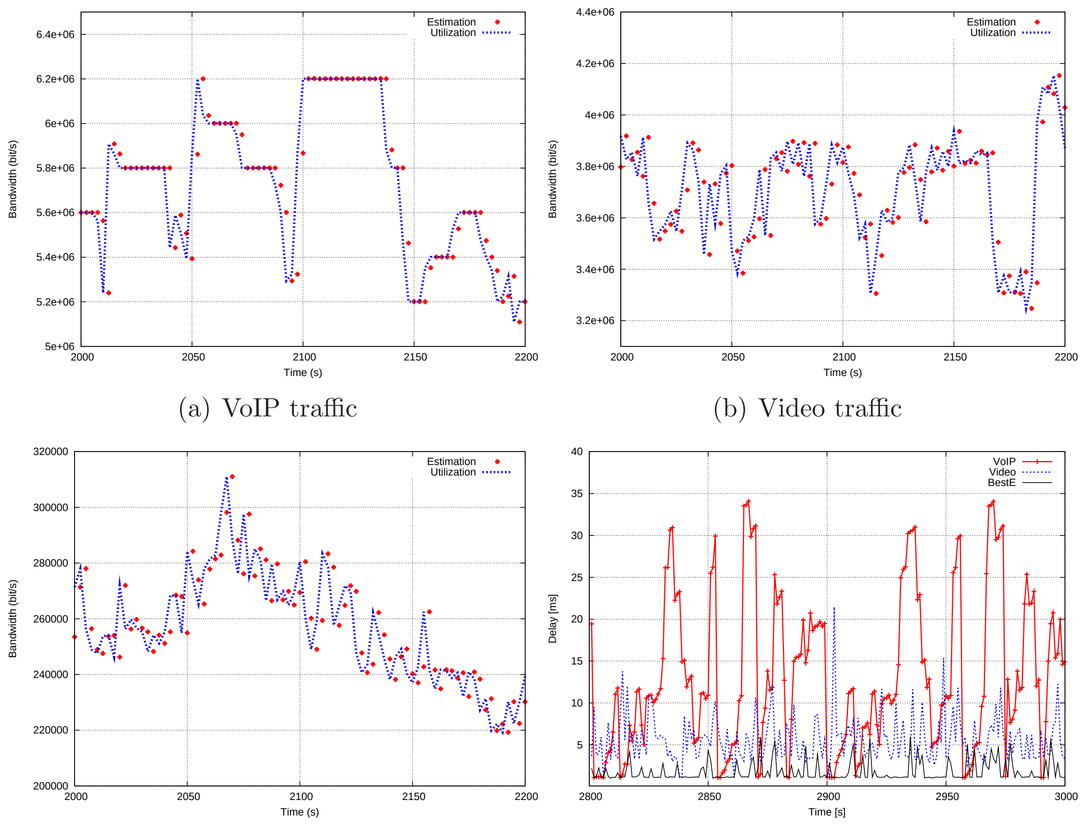
The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.18 and in Figure 6.18.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilizations** (%)
VoIP	16774938	2095392315	231074	10.231	1.2997	95.07
Video	325460	1371553432	0	6.008		
Best effort	679340	83250835	0	1.887		

Table 6.18: Acceptance region tangent at peak performance results

As presented in Table 6.18, this algorithm achieved the highest network utilization level while suffering the largest drop rate as compared to the other algorithms for this scenario. Figure 6.18 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.

Figures 6.18(a), 6.18(b), and 6.18(c) illustrate how good the point sample estimator could estimate the actual network utilization levels for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As shown in the figures, the estimator does better estimations of actual network utilization for VoIP traffic flows as for video and best effort traffic flows. It can also be noticed that the best effort class bandwidth is under utilized due the fact that its highest actual utilization is below 320 kbit/s bandwidth whereas its class bandwidth is 1.5 Mbit/s. Figure 6.18(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown in the figure, the traffic classes exhibit variable delay behaviours where the VoIP class shows the highest delays. But the delays are within the tolerable boundaries for their respective applications.



(a) VoIP traffic

(b) Video traffic

(c) Best effort traffic

(d) VoIP, video and best effort packets delays

Figure 6.18: Actual and estimated utilization: ACTP algorithm; eight-node topology; scenario two

6.1.3 Performance Comparison of the Two Network Topologies

In this section, the simulation results obtained for the two network topologies with the static bandwidth allocation mechanism are compared. The essence of this comparison is not to declare a winner but to create basis and motivation for further investigation of the effects that network topologies could have on overall network performances.

Two simulation scenarios with each of the network topology were investigated. Therefore, two result groups are compared against each other for the two topologies. These results comparison are based on the achieved network performances of the four measurement-based admission control algorithms with the two network topologies. Table 6.19 presents an overview of the results.

The results in Table 6.19 illustrate the achieved utilizations and the drop rates of the four MBAC algorithms for the two simulation scenarios using the two network topologies. It is interesting to notice the difference in performance of the two network topologies for the same simulation scenarios. For example, the

Two-node network topology			**Eight-node network topology**		
Simulation scenario one			Simulation scenario one		
Admission algorithm	Total drops (%)	Total util (%)	Admission algorithm	Total drops (%)	Total util (%)
MS	0.3317	93.03	MS	0.3526	93.14
HB	0.0816	88.54	HB	0.0865	88.84
ACTO	0.3835	83.55	ACTO	0.5053	83.08
ACTP	6.7398	99.13	ACTP	6.3625	99.01
Simulation scenario two			Simulation scenario two		
Admission algorithm	Total drops (%)	Total util (%)	Admission algorithm	Total drops (%)	Total util (%)
MS	0.1582	90.74	MS	0.1727	90.68
HB	0.0148	80.55	HB	0.0126	80.26
ACTO	0.0343	72.85	ACTO	0.1328	73.94
ACTP	1.1976	95.16	ACTP	1.2997	95.07

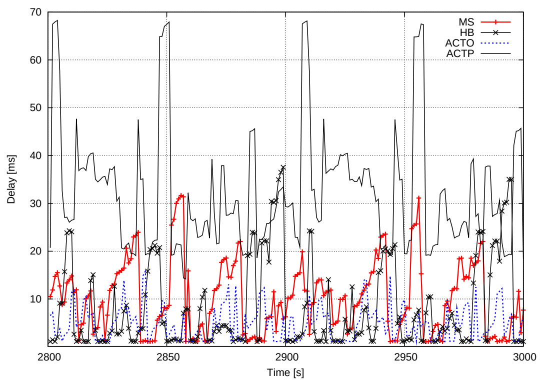
Table 6.19: Performance comparison of the network topologies using static allocated bandwidth

acceptance region tangent at origin admission control algorithm obtained a lower drop rate with the two-node network topology for simulation scenario two as compared to its drop rate achieved with the eight-node network topology for simulation scenario two. On the other hand the same algorithm achieved better network utilization with the eight-node network topology.

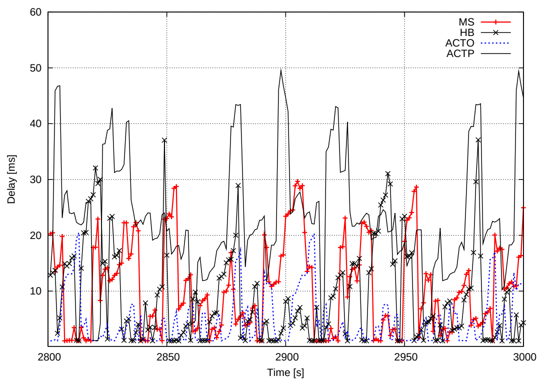
The delay behaviours are compared here only for the VoIP traffic, since it hast the most strict delay requirements from a network. Figure 6.19 depicts the flow delays revealed by the four MBAC algorithms using the two network topologies for simulation scenario one.
Figure 6.19(a) presents the flow delays experienced by the four MBAC algorithms using the two-node network topology and Figure 6.19(b) illustrates the packet delays sensed by the four algorithms using the eight-node nework topology. The figures depict the delays experienced by the VoIP traffic classes flows while traversing the multiservice bottleneck link. Taking some closer look at the figures one could notice the different delays exhibitted by the same algorithms between the two network topologies. For example, the ACTO algorithm shows higher delays in two-node network topology as in eight-node network topology.

Figure 6.20 illustrates the traffic flow delay behaviours sensed by the four MBAC algorithms using the two network topologies for simulation scenario two.

Figure 6.20(a) presents the flow delays experienced by the four algorithms using

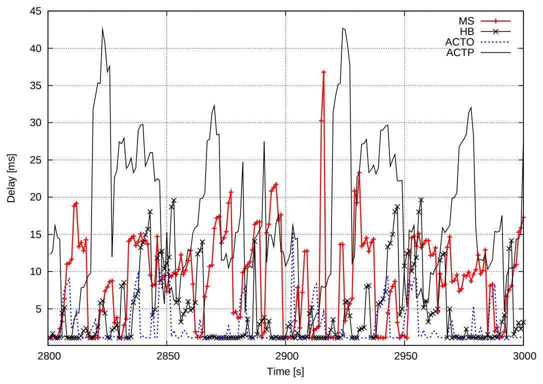


(a) With two-node topology

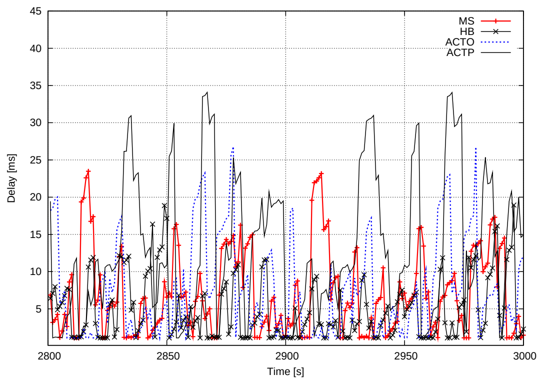


(b) With eight-node topology

Figure 6.19: Delays experienced by VoIP traffic for scenario one



(a) With two-node topology



(b) With eight-node topology

Figure 6.20: Delays experienced by VoIP traffic for scenario two

the two-node network topology and Figure 6.20(b) illustrates the flow delays in case of the eight-node nework topology for the same simulation scenario. The figures show the delays experienced by the VoIP traffic classes while sending their flows over the multiservice bottle-neck link. It is noticed from the figures that the same MBAC algorithm displays different delay behaviours in the two network topologies. For example, the ACTO algorithm displays lower delays for the same simulation scenario in two-node network topology as in eight-node network topology. The reason for such behaviours are subject to further research.

Although, the numerical results in Table 6.19, and the traffic flow delay behaviours shown in Figure 6.19 and 6.20 does not show magnificant differences, they do demonstrate the effects of the two network topologies in simulating simultaneous multi-class traffic traversing over a packet network controlled by the MBAC algorithms.

6.2 Dynamic Bandwidth Allocation Mechanism

In this section, the dynamic bandwidth allocation mechanism integrated into the multiservice framework is simulated to evaluate its effects for the drop rate, packet delays, and network utilization performances. The static bandwidth allocation mechanism simulated in the previous section (6.1) shares the network bandwidth among the traffic classes according to the class priorities. Each class then operates only with the allocated bandwidth share. Consider the situation where the low priority best effort applications, which requires loose QoS is sending their packets at low rate thereby not utilizing their class bandwidth, and the higher priority VoIP and video classes, which are of high demands and require more strict QoS are short of bandwidths. In this situation, using the static bandwidth allocation mechanism causes waste of the unutilized best effort bandwidth as it was the case in the second simulation scenario of the previous section.

In an effort to allivate this problem of bandwidth wastage and strive to improve the total network performance, the dynamic bandwidth allocation mechanism is devised. This mechanism makes it possible for the VoIP and video classes to borrow the unused best effort bandwidth when they ran short of their own class bandwidths and the best effort bandwidth is underutilized.

There are two simulation scenarios in this section designed to test the multiservice framework using the two network topologies. The achieved performance results of the four measurement-based admission control algroithms using each of the network topology are presented and compared at the end of the section.

6.2.1 Simulation with the Two-Node Network Topology

The two-node network topology used in this section to evaluate the multiservice framework with dynamic bandwidth allocation mechanism has the same structure and parameters as the one introduced in subsection 6.1.1. It is made up of two nodes. The source agents and application traffic generators are attached to the sending node on one side and on the other side the receiver agents and the application traffic sinks are attached to the receiving node.

This network topology is used to carry out the two simulations scenarios for the multiservice framework in this subsection. The scenarios are presented in the subsequent subsections.

6.2.1.1 Simulation Scenario One

The first simulation scenario is designed to evaluate the effects of transmitting three traffic classes (VoIP, video, and best effort) packets over a packet network using the dynamic bandwidth allocation mechanism. The four MBAC algorithms control the admission of new traffic flows into the network. The scenario stimulates the standard situation, where the three traffic classes are sending at their full rates and striving to fully utilize their class bandwidths.

This simulation scenario uses the same application source traffic generators and their parameters for generating and sending traffic of the three traffic classes as the ones described in Table 6.1, in subsection 6.1.1.1. As mentioned earlier the CBR traffic generator is used to generate VoIP packets, video trace file is used to generate video packets, and exponential traffic generator is used for generating best effort traffic.

The performance results achieved by each of the four measurement-based admission control algorithms are presented below.

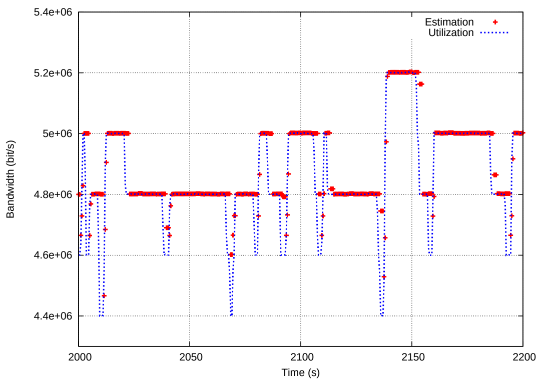
Measured Sum
The performance results achieved by the measured sum for this scenario are displayed in Table 6.20 and in Figure 6.21.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	14523740	1813863835	73056	9.594	0.3961	93.57
Video	294916	1235298236	0	5.403		
Best effort	3627192	451765200	0	1.887		

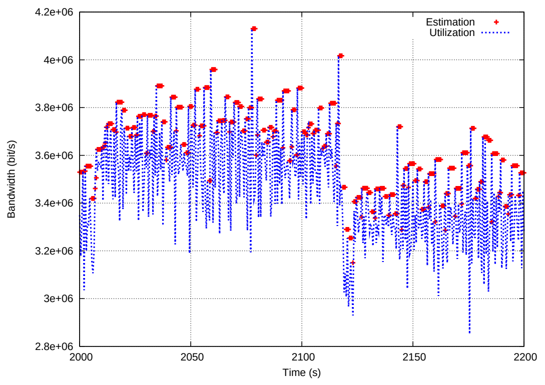
Table 6.20: Measured sum performance results

As shown in Table 6.20, the algorithm obtained high network utilization level while suffering low packet drop rate, considering the fact that the utilization factor is configured for 95 percent, i.e., the algorithm cannot achieve a utilization level higher than 95 percent. Figure 6.21 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario.

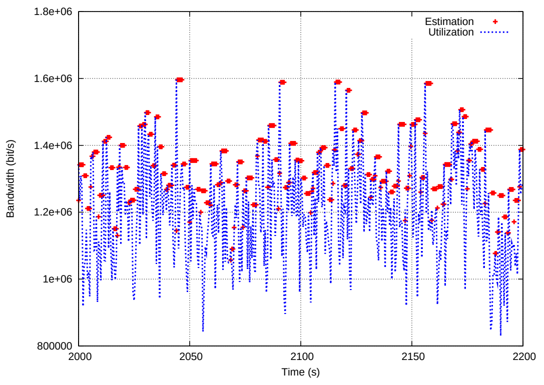
Figures 6.21(a), 6.21(b), and 6.21(c) illustrate how good the time-window estimator could estimate the actual network utilization levels for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. According to the graphics, the estimator does better estimation of the actual utilization for VoIP traffic flows as for video and best effort traffic flows. They also illustrate the degree of utilization of the respective class bandwidths. Figure 6.21(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown in the figure, the delay behaviours of the traffic classes are not constant but they are still below the tolerable boundaries of their respective applications.



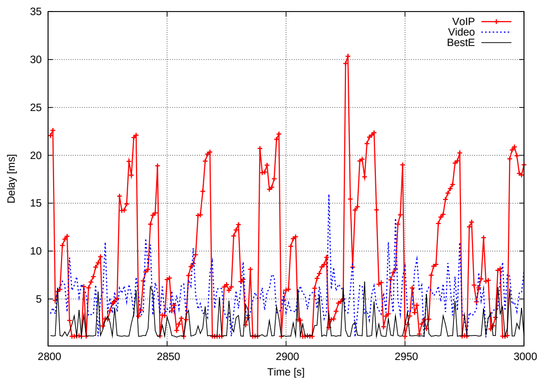
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.21: Actual and estimated utilization: MS algorithm; two-node topology; scenario one

Hoeffding Bounds

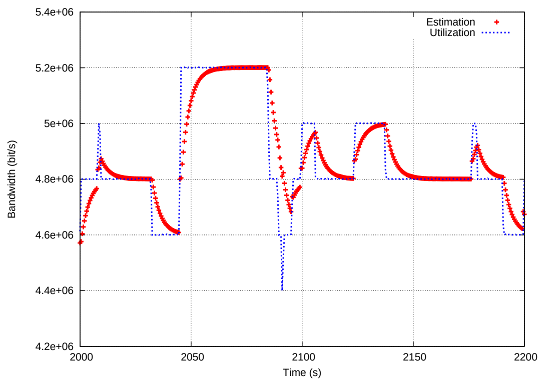
The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.21 and in Figure 6.22.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	13939180	1740795935	12717	5.964	0.0741	86.44
Video	267787	1118597571	0	5.153		
Best effort	2961891	368611710	0	1.785		

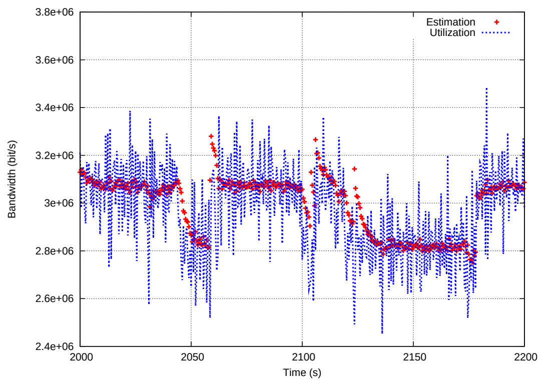
Table 6.21: Hoeffding bounds performance results

As illustrated in Table 6.21, the algorithm achieves high total utilization level while suffering low packet drop rate. Figure 6.22 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.

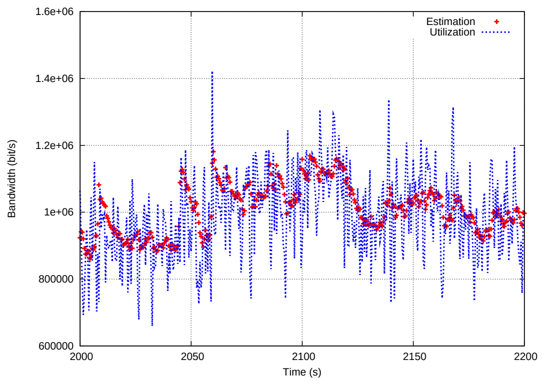
Figures 6.22(a), 6.22(b), and 6.22(c) illustrate how good the exponential averaging estimator could estimate the actual network utilization levels for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort ap-



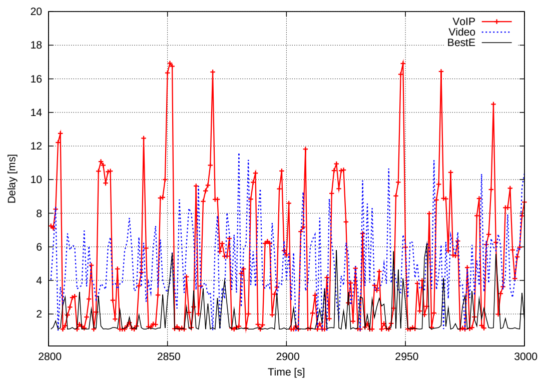
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.22: Actual and estimated utilization: HB algorithm; two-node topology; scenario one

plication packets in a packet network. As it can be seen from the graphics, the estimator performs better for VoIP traffic flows as for video and best effort traffic flows. The graphics also depicts the degree of utilization of the respective class bandwidths. Figure 6.22(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be observed from the graphic that the traffic classes show variable delay behaviours but their highest delays are below the maximum tolerable delay thresholds for their applications.

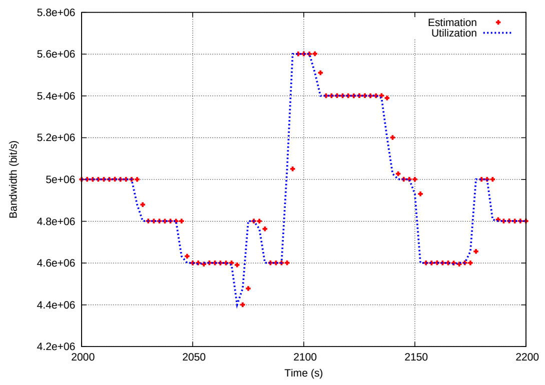
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.22 and in Figure 6.23.

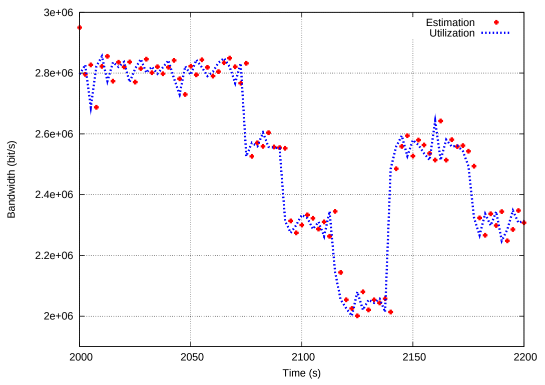
Traffic class	**Received packets** (pkt)	**Received packets size** (pkt)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	14090656	1759729175	86516	5.265	0.5261	83.35
Video	268542	1112276408	0	5.274		
Best effort	2085562	259078775	0	1.751		

Table 6.22: Acceptance region tangent at origin performance results

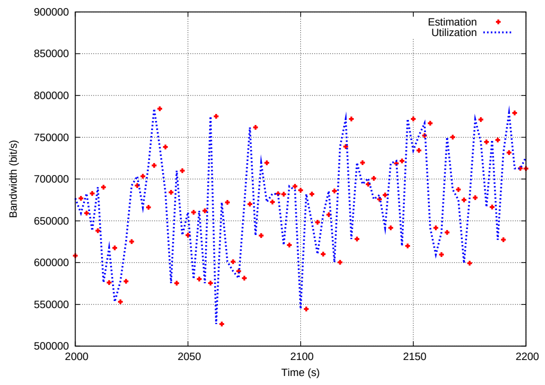
It is observable from Table 6.22 that the algorithm achieved high utilization level and low drop rate. Figure 6.23 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm for this simulation scenario.



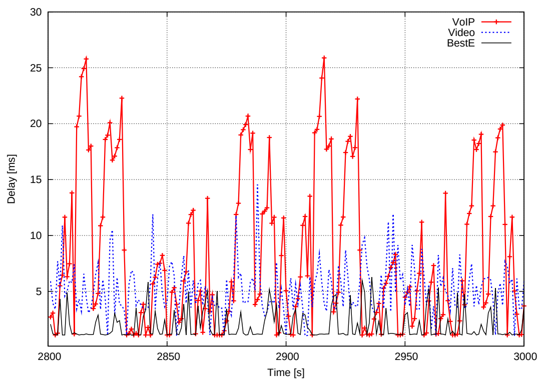
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.23: Actual and estimated utilization: ACTO algorithm; two-node topology; scenario one

Figures 6.23(a), 6.23(b), and 6.23(c) illustrate how good the point sample estimator could estimate the actual network utilization for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets. As it can be noticed in the graphics, the estimator does good estimations of the actual utilizations for the three traffic classes. The graphics also show the degree of utilization of the respective class bandwidths. Figure 6.23(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be observed from the graphic that the traffic classes exhibit variable delay behaviours but their highest delays are still below the tolerable delay thresholds for their respective applications.

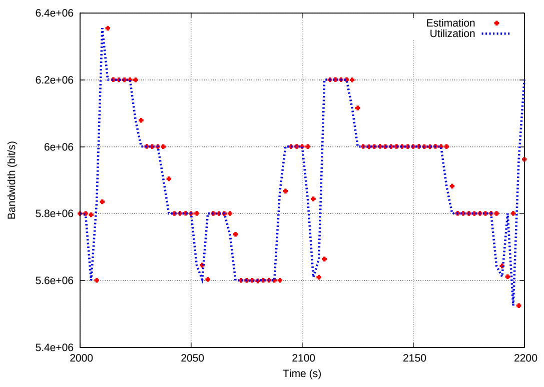
Acceptance Region Tangent at Peak

The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.23 and in Figure 6.24.

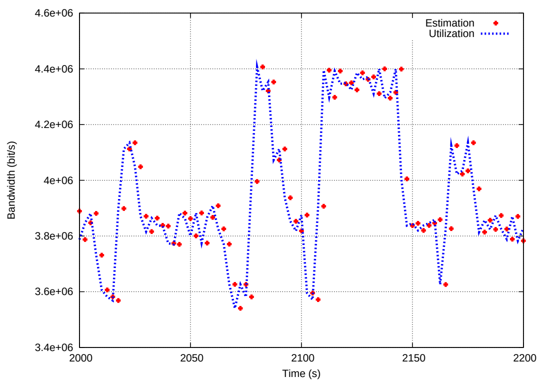
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	17028934	2127007625	1622781	19.600	8.1650	99.28
Video	346025	1459232821	0	5.713		
Best effort	2500008	310880640	0	2.002		

Table 6.23: Acceptance region tangent at peak performance results

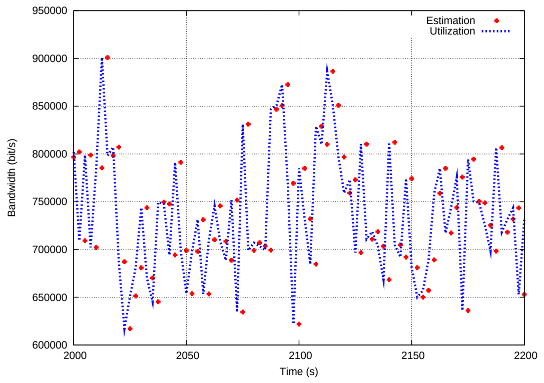
As shown in Table 6.23, the algorithm achieved high network utilization level with a higher packet drop rate. For this scenario, this algorithm achieved the highest utilization level but with highest packet drop rate. Figure 6.24 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.



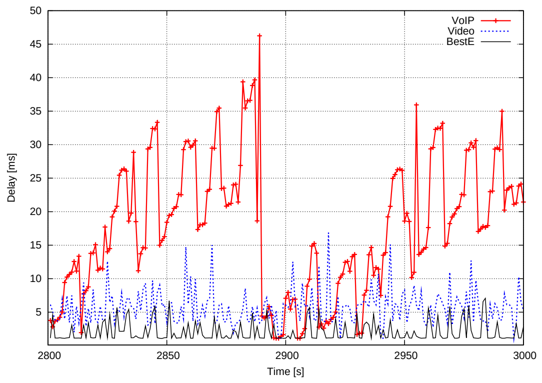
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.24: Actual and estimated utilization: ACTP algorithm; two-node topology; scenario one

Figures 6.24(a), 6.24(b), and 6.24(c) illustrate how good the point sample estimator could estimate the actual network utilizations for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets. As it can be observed from the graphics, the estimator performs well for all three traffic flows. The grahics also show the degree of utilization of the respective class bandwidths. Figure 6.24(d) depicts the delay behaviours of VoIP, video and best effort traffic flows. It can be noticed from the graphic that the traffic classes show variable delay behaviours and the VoIP class shows the highest delays, but the delays are still below the maximum tolerable delay boundaries for their respective applications.

6.2.1.2 Simulation Scenario Two

The second simulation scenario using the two-node network topology demonstrates a situation where the best effort application traffic flows can't fully utilize their class bandwidth. The sending rate of the best effort applications is purposely reduced so that they only send their traffic packets at a low rate thereby causing a low utilization of their class bandwidth. The idea behind this scenario is to create a situation where the dynamic bandwidth allocation mechanism can be effectively evaluated.

This simulation scenario is configured with the same application source traffic generators and their parameters as presented in Table 6.6, subsection 6.1.1.2. The four measurement-based admission control algorithms are used in controlling the admission of new traffic flows into the network. The obtained performance results are described as shown below.

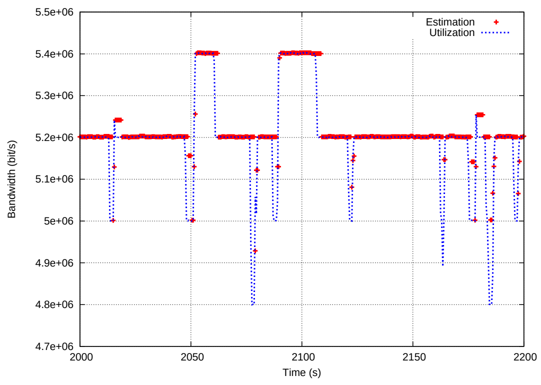
Measured Sum
The performance results achieved by the measured sum for this simulation scenario are displayed in Table 6.24 and in Figure 6.25.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15728663	1964475430	66210	8.214	0.3742	91.98
Video	305587	1279337427	0	5.401		
Best effort	1659330	205643850	0	1.850		

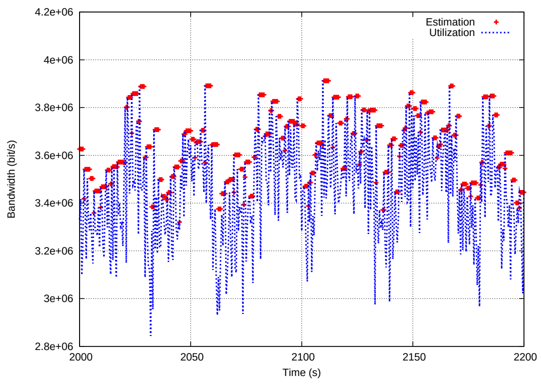
Table 6.24: Measured sum performance results

As shown in Table 6.24, the algorithm achieved high utilization level while suffering low packet drop rates. This results reflects the better usage of the unused best effort class bandwidth. Figure 6.25 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario.

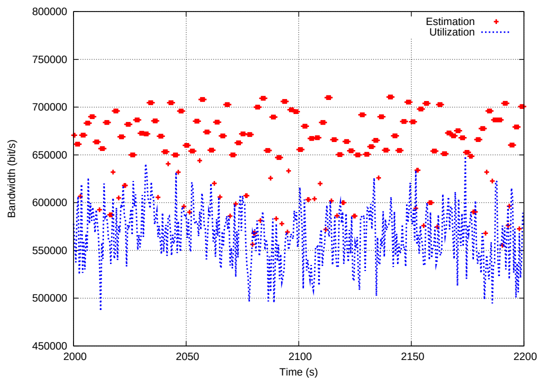
Figures 6.25(a), 6.25(b), and 6.25(c) illustrate how good the time-window estimator could estimate the actual network utilizations for the measured sum algorithm while transmitting VoIP, video, and best effort application packets. As



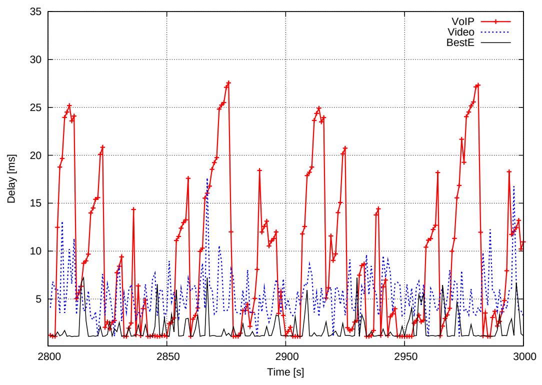
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.25: Actual and estimated utilization: MS algorithm; two-node topology; scenario two

it can be seen from the graphics, the estimator does better estimations of the actual utilizations for VoIP traffic flows as for video and best effort traffic flows. It can also be observed from the grahics that the best effort class bandwith is not optimally utilized. Figure 6.25(d) illustrates the delay behaviours of VoIP, video and best effort traffic flows. It is noticed from the graphic that the traffic classes exhibits variable delay behaviours but their highest delays are still below tolerable delay threshold for their respective applications.

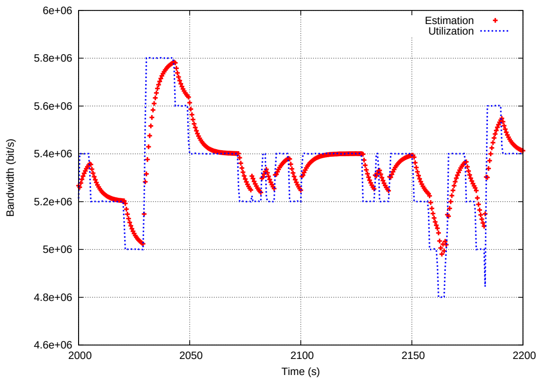
Hoeffding Bounds

The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.25 and in Figure 6.26.

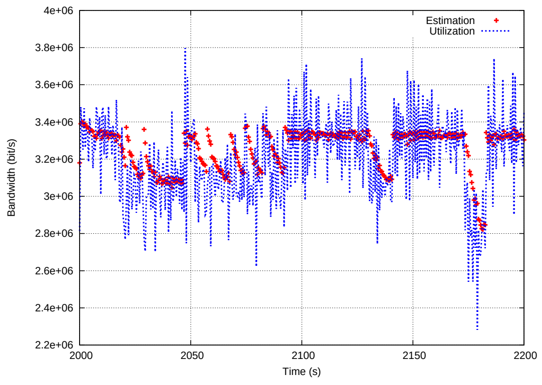
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15482510	1933708615	15912	5.874	0.0975	87.01
Video	289176	1205962436	0	5.093		
Best effort	544252	66381635	0	1.791		

Table 6.25: Hoeffding bounds performance results

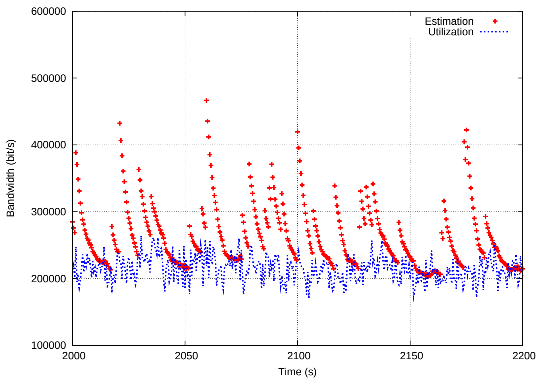
As presented in Table 6.25, the algorithm achieves high network utilization level while suffering low packet drop rates regardless of the low sending rate of the best effort traffic. This shows the efficincy of the dynamic bandwidth allocation mechanism. Figure 6.26 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.



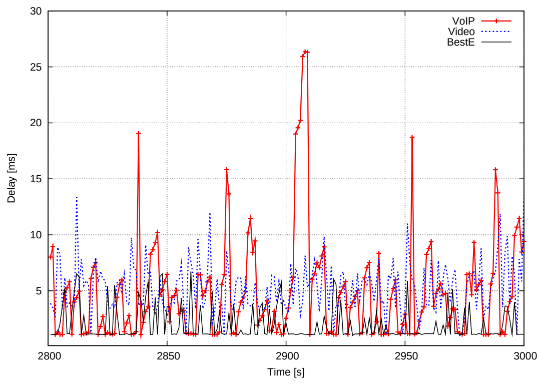
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.26: Actual and estimated utilization: HB algorithm; two-node topology; scenario two

Figures 6.26(a), 6.26(b), and 6.26(c) illustrate how good the exponential averaging estimator could estimate the actual network utilizations for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort application packets. As it can be observed from the graphics, the estimator performs better for VoIP traffic flows as for video and best effort traffic flows. The graphics also illustrate the degree of utilization of the respective class bandwidths. Figure 6.26(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be observed from the graphic that the traffic classes show variable delay behaviours, but their highest delays are still below the maximum tolerable delay thresholds for their respective applications.

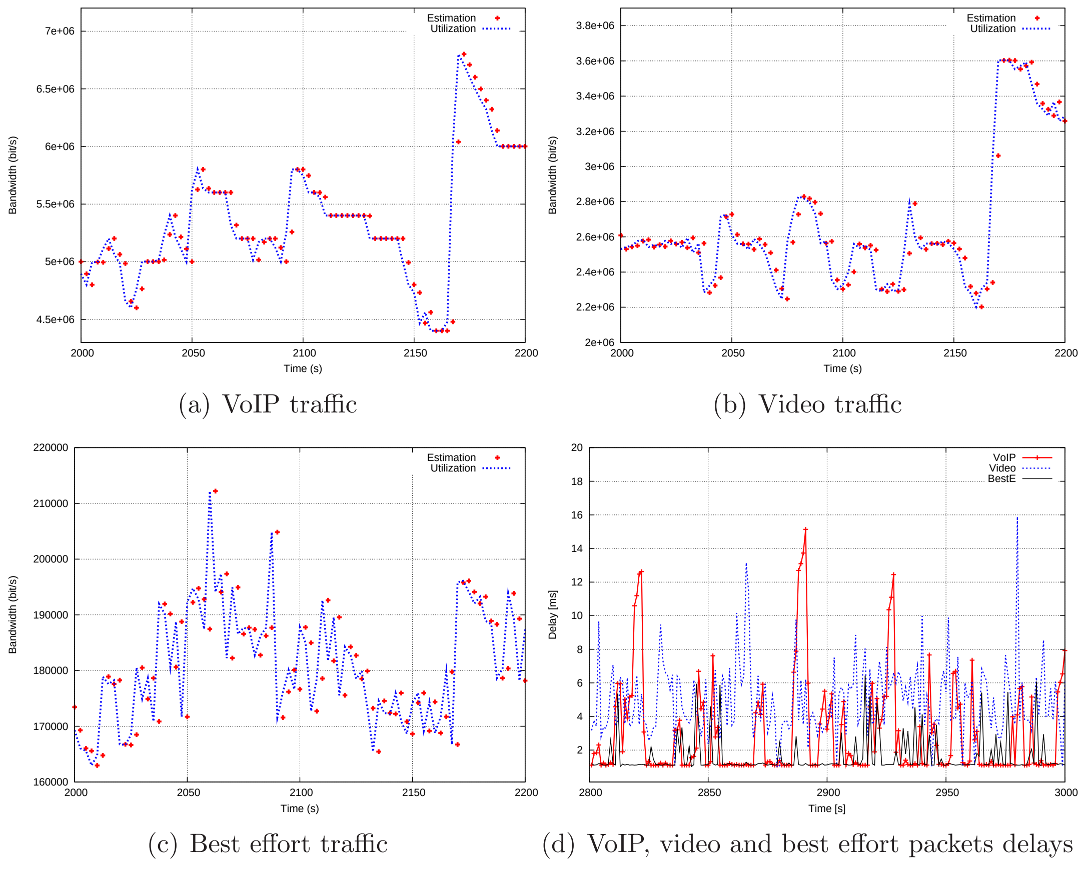
Acceptance Region Tangent at Origin

The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.26 and in Figure 6.27.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15835990	1977892460	31165	4.208	0.1879	83.63
Video	259391	1083880737	0	4.974		
Best effort	494085	60116745	0	1.691		

Table 6.26: Acceptance region tangent at origin performance results

As presented in Table 6.26, the algorithm obtainea a high network utilization level while suffering low packet drop rate inspite of the low sending rate of the best effort traffic flows. The results confirms the efficiency of the dynamic bandwidth allocation mechanism for the algorithm. Figure 6.27 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm for this simulation scenario.



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.27: Actual and estimated utilization: ACTO algorithm; two-node topology; scenario two

Figures 6.27(a), 6.27(b), and 6.27(c) illustrate how good the point sample estimator could estimate the actual network utilizations for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort traffic. As it can be observed from the graphics, the estimator does good estimations of the actual utilizations for the three traffic classes. The grahics also show the degree of utilization of the respective class bandwidths. In the case of best effort, it can be seen that the class bandwidth utilization is in kbit/s range instead of Mbit/s. Figure 6.27(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It is noticable from the graphic that the traffic classes exhibit variable delay behaviours but their highest delays are still below the maximum tolerable delay thresholds for their respective applications.

Acceptance Region Tangent at Peak

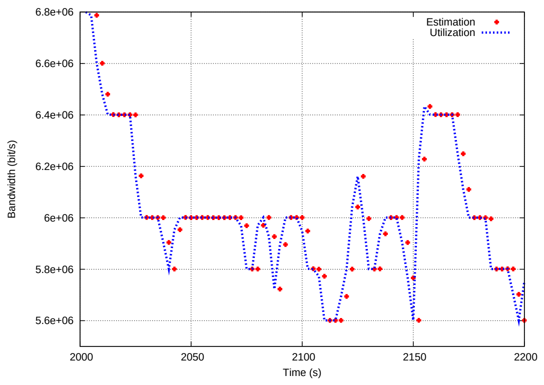
The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.27 and in Figure 6.28.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	17937099	2240524365	847206	17.048	4.4837	98.44
Video	341535	1449645871	0	5.380		
Best effort	616552	75410210	0	1.939		

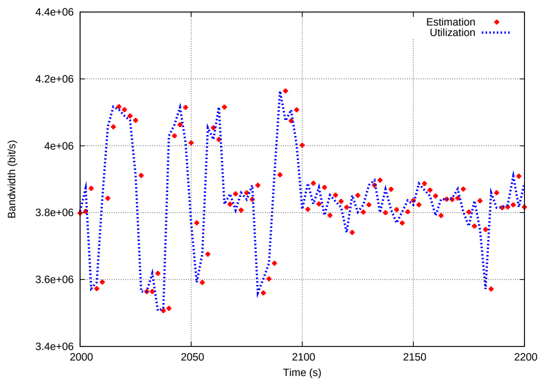
Table 6.27: Acceptance region tangent at peak performance results

As presented in Table 6.27, the algorithm achieved high network utilization level while suffering a little higher packet drop rate, regardless of the low sending rate of the best effort application traffic. These results show the efficiency of the dynamic bandwidth allocation mechanism for the algorithm. Figure 6.28 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.

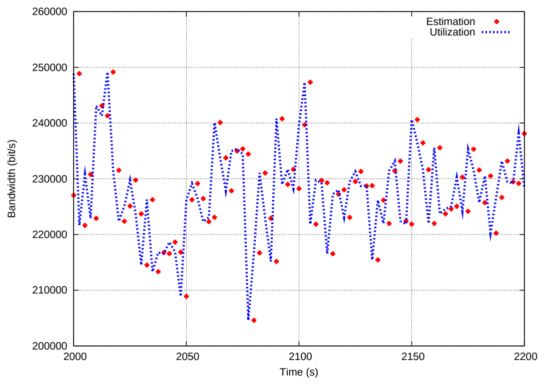
Figures 6.28(a), 6.28(b), and 6.28(c) illustrate how good the point sample estimator could estimate the actual network utilizations for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets. As it can be observed from the graphics, the estimator performs good estimations of the actual utilizations for all three traffic classes. The graphics also show the degree of utilization for the respective class bandwidths. Figure 6.28(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be noticed from the graphic that the traffic classes show variable delay behaviours but their highest delays are still below the maximum acceptable delay for their respective applications.



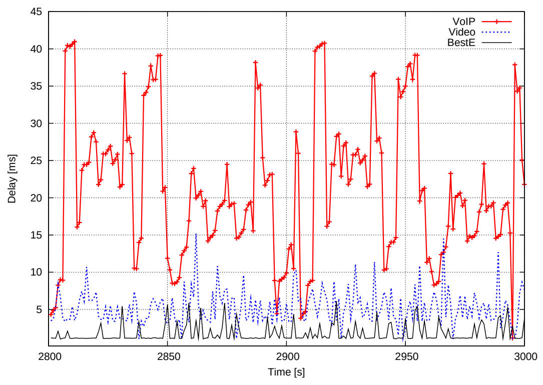
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.28: Actual and estimated utilization: ACTP algorithm; two-node topology; scenario two

6.2.2 Simulation with the Eight-Node Network Topology

The eight-node network topology used for simulation in this section is the same as the one presented in Figure 6.10, subsection 6.1.2. As already described, the source agents and the application source traffic generators for the three traffic classes are attached to separate nodes and they send their traffic through a router node on one side and on the other side the receiving agents and the application sink agents are attached to separate nodes and they receive traffic through a router node.

This network topology is used for two simulation scenarios in this section. The simulation scenarios are described in the following two subsections.

6.2.2.1 Simulation Scenario One

The first simulation scenario using the eight-node network topology and the dynamic bandwidth allocation mechanism is designed to provide a standard situation in a packet network where the three traffic classes are fully utilizing their class bandwidths.

The scenario uses the same application traffic source generators and their parameters as those specified in Table 6.1, subsection 6.1.1.1. The four MBAC algorithms are used to control the admission of new traffic flows into the network. The performance results obtained by each of the four MBAC algorithms are presented below.

Measured Sum

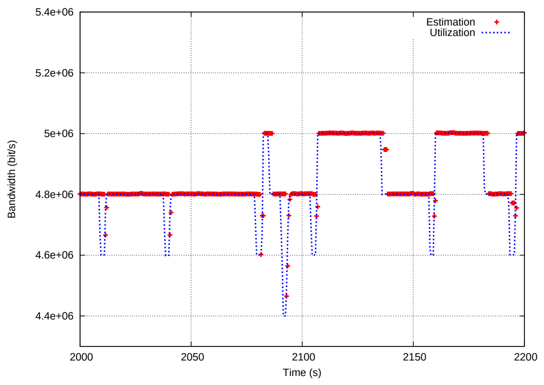
The performance results achieved by the measured sum for this scenario are displayed in Table 6.28 and in Figure 6.29.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	14512053	1812445905	66410	9.867	0.3526	93.14
Video	281277	1173995547	0	5.670		
Best effort	4041399	503535930	0	1.876		

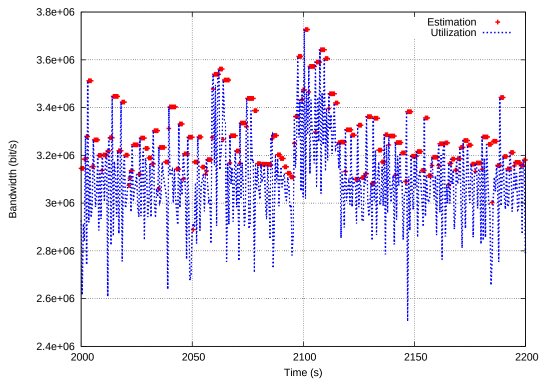
Table 6.28: Measured sum performance result

As depicted in Table 6.28, the algorithm achieves high network utilizations while maintaining low packet drop rate. Figure 6.29 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario.

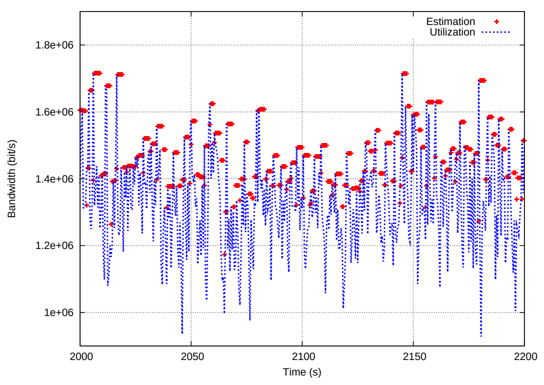
Figures 6.29(a), 6.29(b), and 6.29(c) illustrate how good the time-window estimator could estimate the actual network utilizations for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As it can be observed from the graphics, the estimator performs better for VoIP traffic flows as for video and best effort traffic flows. The graphics also show the degree of utilization of the respective class bandwidths. Figure 6.29(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be seen from the graphic that the traffic classes exhibits variable delay behaviours but their highest delays are still below the maximum acceptable delay thresholds for their respective applications.



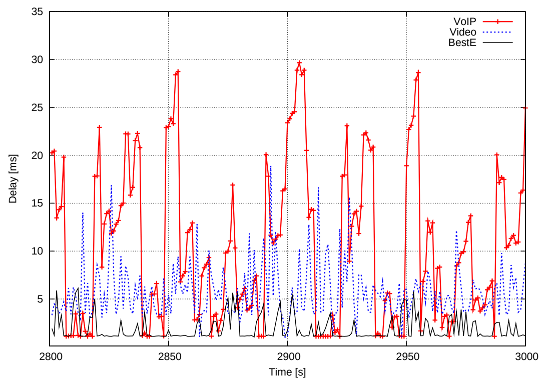
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.29: Actual and estimated utilization: MS algorithm; eight-node topology; scenario one

Hoeffding Bounds

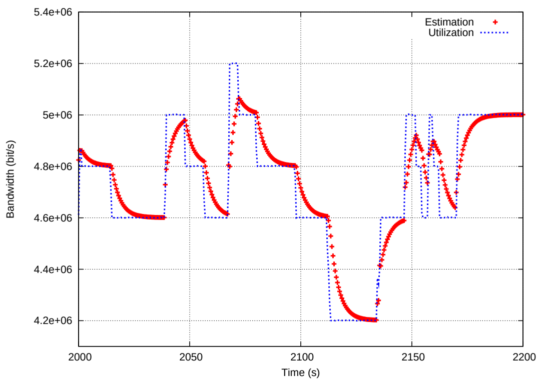
The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.29 and in Figure 6.30.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	13966262	1744206175	13147	6.877	0.0765	87.60
Video	270628	1123911437	0	5.092		
Best effort	2956413	367928745	0	1.791		

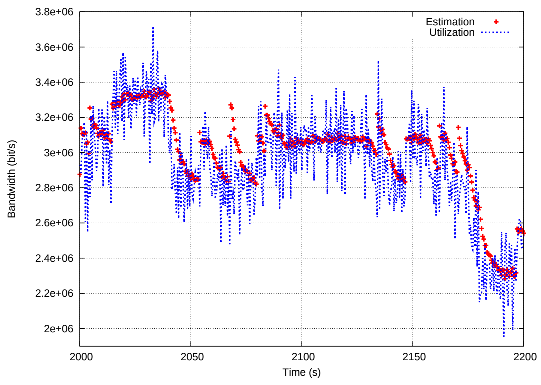
Table 6.29: Hoeffding bounds performance results

As presented in Table 6.29, the algorithm obtaines good total network utilization level while keeping low packet drop rate. Figure 6.30 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.

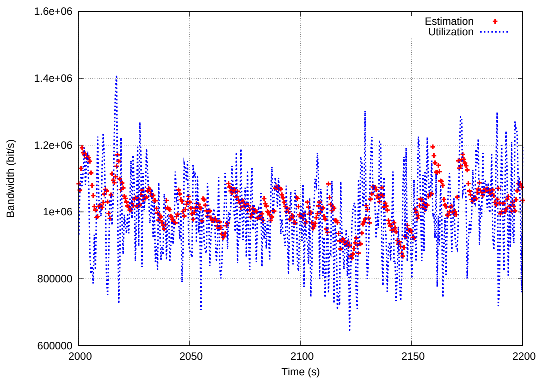
Figures 6.30(a), 6.30(b), and 6.30(c) illustrate how good the exponential averaging estimator could estimate the actual network utilizations for the hoeffding



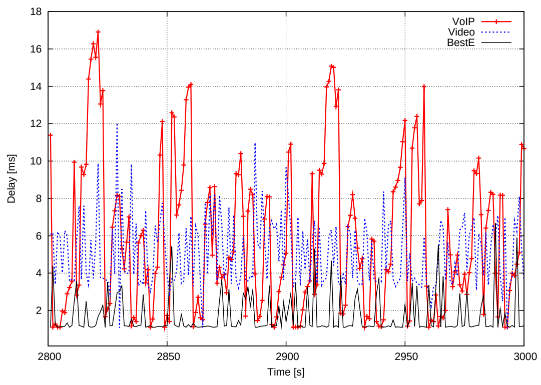
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.30: Actual and estimated utilization: HB algorithm; eight-node topology; scenario one

bounds algorithm while transmitting VoIP, video, and best effort application packets. As it can be seen from the grahics, the estimator does better estimations of actual utilizations for VoIP traffic flows as for video and best effort traffic flows. The graphics also depict the degree of utilization of the respective class bandwidths. Figure 6.30(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown on the graphic, the three traffic classes show variable delay behaviours, but their highest delays are still below the maximum tolerable delay boundaries for their respective applications.

Acceptance Region Tangent at Origin

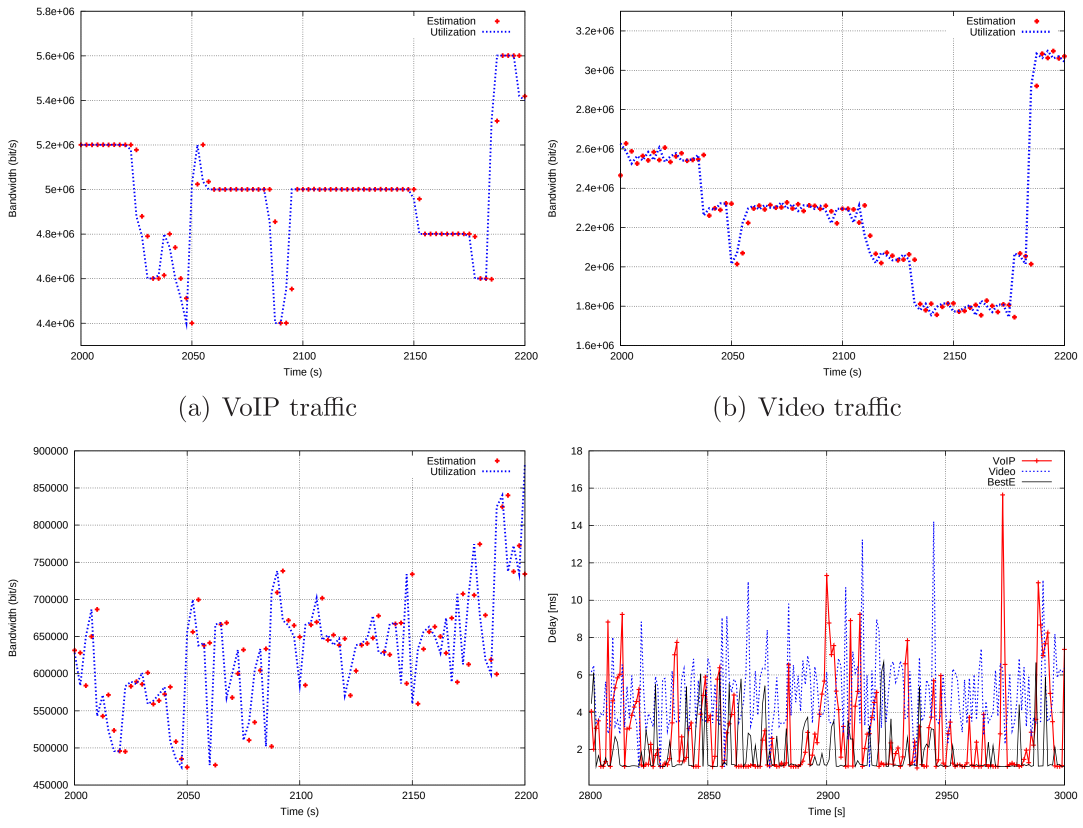
The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.30 and in Figure 6.31.

As presented in Table 6.30, the algorithm achieves a good total network utilization while maintaining a low packet drop rate. Figure 6.31 depicts the behaviour of the point sample estimator and the acceptance region tangent at origin admission control algorithm for this simulation scenario.

Figures 6.31(a), 6.31(b), and 6.31(c) illustrate how good the point sample es-

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	14346259	1791702335	12123	4.086	0.0727	80.80
Video	239251	986021143	0	4.834		
Best effort	2081926	258623015	0	1.673		

Table 6.30: Acceptance region tangent at origin performance results



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.31: ACTO algorithm using eight-node topology for scenario one

timator could estimate the actual network utilization for the acceptance region tangent at origin algorithm while transmitting VoIP, video, and best effort application packets. It is observable from the graphics that the estimator performs good estimations for all three traffic classes. The graphics also show the degree of utilization of their respective class bandwidths. Figure 6.31(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As it can be seen on the graphic, the traffic classes exhibit variable delay behaviours but their highest delays are still below the maximum acceptable delay threshold for their respective applications.

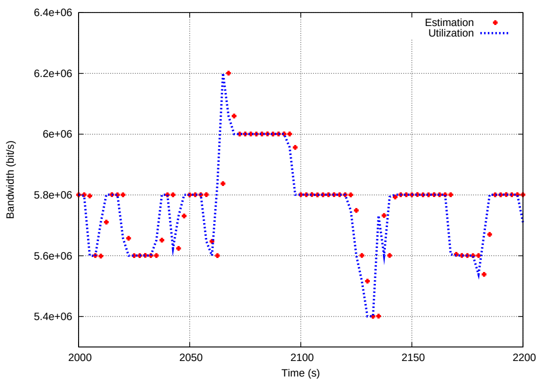
Acceptance Region Tangent at Peak

The results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.31 and in Figure 6.32.

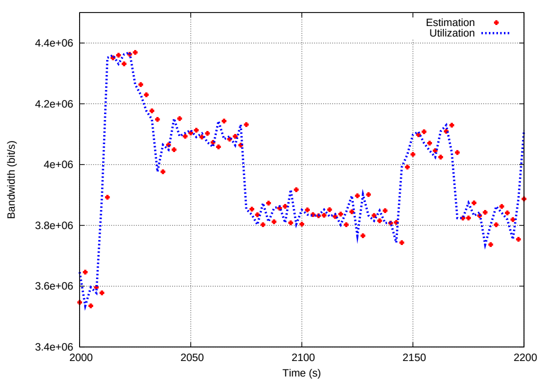
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	16632595	2077596500	1357079	19.652	6.9463	99.37
Video	343052	1452878192	0	5.820		
Best effort	2560763	318474595	0	1.993		

Table 6.31: Acceptance region tangent at peak performance results

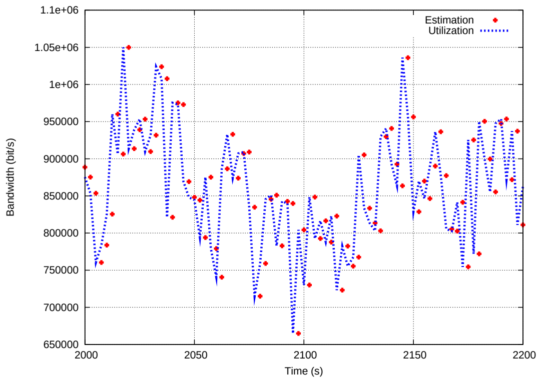
As shown in Table 6.31, the algorithm achieves high total network utilization level while suffering a relatively high packet drop rate. The algorithm achieved the highest total network utilization level among the other algorithms for this scenario. Figure 6.32 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.



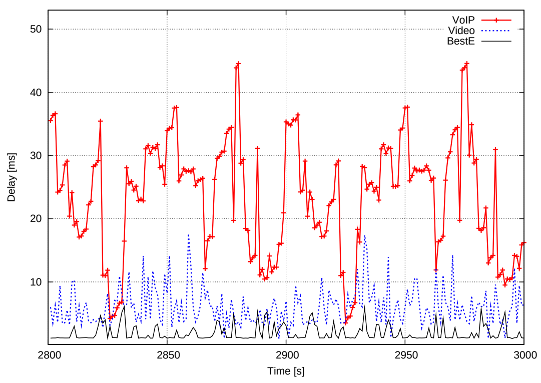
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.32: Actual and estimated utilization: ACTP algorithm; eight-node topology; scenario one

Figures 6.32(a), 6.32(b), and 6.32(c) illustrate how good the point sample estimator could estimate the actual network utilization for the acceptance region tangent at peak algorithm while transmitting VoIP, video, and best effort application packets. As can be seen on the graphics, the estimator performs good estimations for all three traffic classes. The graphics also illustrate the degree of utilization of their respective class bandwidths. Figure 6.32(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown on the graphic, the traffic classes exhibit variable delay behaviours but their highest delays are still below the maximum tolerable delay thresholds for their respective applications.

6.2.2.2 Simulation Scenario Two

The second simulation scenario using eight-node network topology and the dynamic bandwidth allocation mechanism is created to demonstrate a situation in a packet network where the best effort applications are sending their traffic packets at a low rate, thereby not being able to fully utilize their allocated class bandwidth.

The best effort applications are purposely configured to send their packets at low rate so as to evaluate the behaviour of the dynamic bandwidth allocation mechanism, which should then allocate the unused best effort class bandwidth to the higher priority VoIP and video classes.

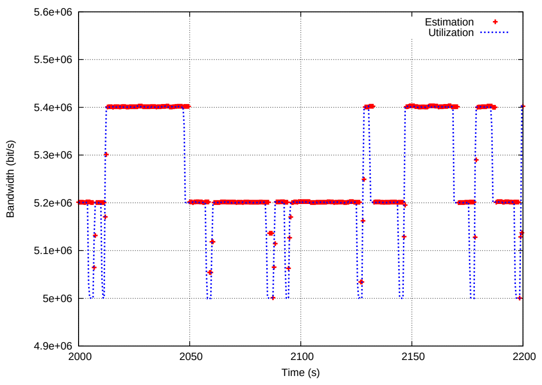
This simulation scenario is set up with the same application traffic source generators and their parameters like the ones presented in Table 6.6, in subsection 6.1.1.2. The four measurement-based admission control algorithms are used to regulate the acceptance of new traffic flows into the network. The performance results obtained by these algorithms are described below.

Measured Sum
The performance results achieved by the measured sum for this scenario are illustrated in Table 6.32 and in Figure 6.33.

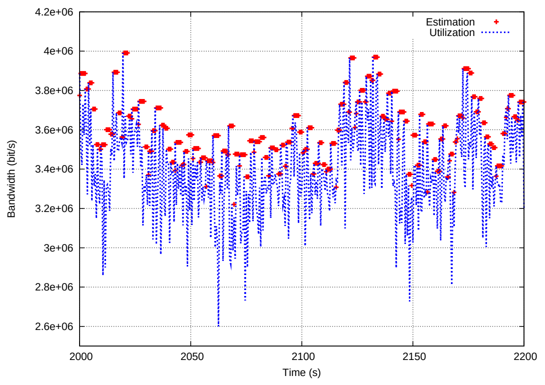
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15667109	1956860455	60913	8.678	0.3454	92.30
Video	305920	1283982609	0	5.308		
Best effort	1664347	206267825	0	1.870		

Table 6.32: Measured sum performance results

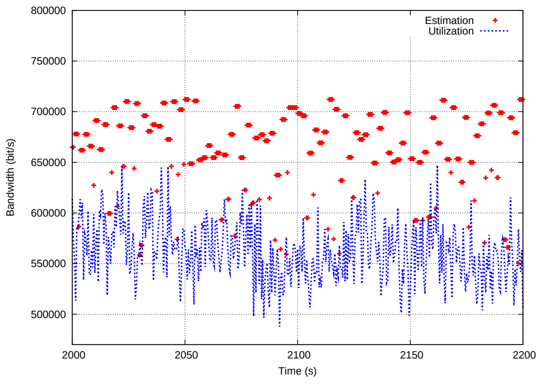
As shown in Table 6.32, the algorithm achieves a high total network utilization level while maintaining low packet drop rate, inspite of the fact that the utilization factor is set for 95 percent and the best effort applications are sending at a low rate. This result confirms the efficiency of the dynamic bandwidth allocation mechanism for this algorithm. Figure 6.33 displays the behaviour of the time-window estimator and the measured sum admission control algorithm for this simulation scenario.



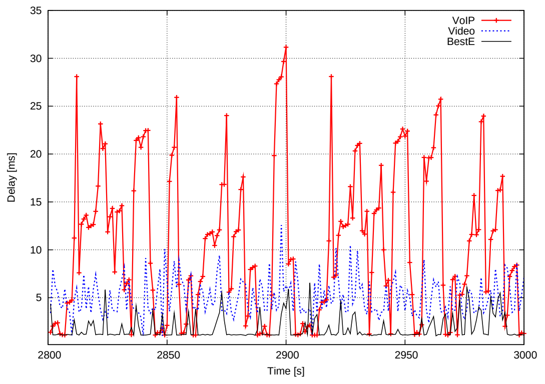
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.33: Actual and estimated utilization: MS algorithm; eight-node topology; scenario two

Figures 6.33(a), 6.33(b), and 6.33(c) illustrate how good the time-window estimator could estimate the actual network utilizations for the measured sum algorithm while transmitting VoIP, video, and best effort application packets in a packet network. As it can be seen on the graphics, the estimator performs better estimations for VoIP traffic as for video and best effort traffic flows. The graphics also show the degree of utilization of their respective class bandwidths. In the case of best effort traffic class, it is clearly observable that the class bandwidth is not optimally utilized. Figure 6.33(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be noticed from the graphic that the traffic classes exhibit variable delay behaviours but their highest delays are still below the maximum tolerable delay thresholds for their respective applications.

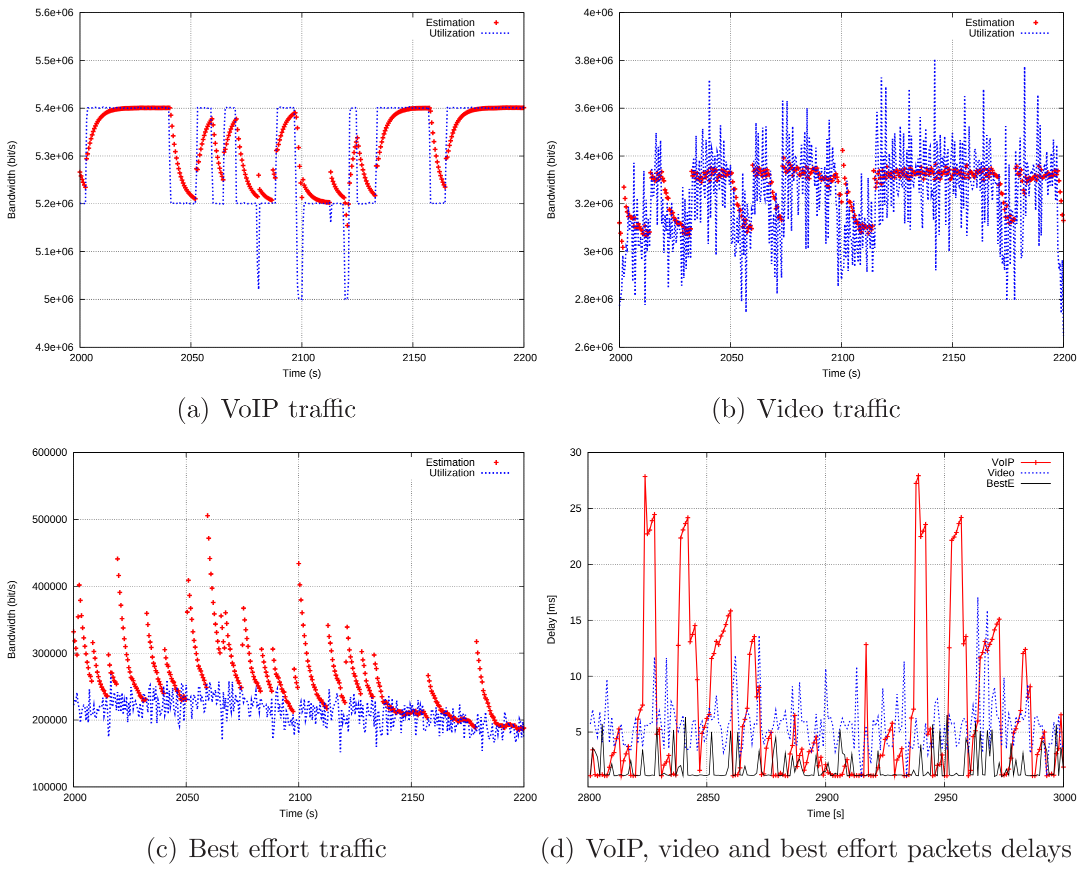
Hoeffding Bounds

The performance results achieved by the equivalent bandwidth based on hoeffding bounds algorithm are shown in Table 6.33 and in Figure 6.34.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15488303	1934485975	17507	6.362	0.1072	86.83
Video	286090	1200911655	0	5.494		
Best effort	554886	67710885	0	1.814		

Table 6.33: Hoeffding bounds performance results

As shown in Table 6.33, the algorithm obtaines good total network utilization level while maintinaing low drop rate regardless of the fact that the best effort traffic is sending at low rate. This result shows the efficiency of the dynamic bandwidth allocation mechanism for this algorithm. Figure 6.34 depicts the behaviour of the exponential averaging estimator and the hoeffding bounds admission control algorithm for this simulation scenario.



(a) VoIP traffic (b) Video traffic

(c) Best effort traffic (d) VoIP, video and best effort packets delays

Figure 6.34: Actual and estimated utilization: HB algorithm; eight-node topology; scenario two

Figures 6.34(a), 6.34(b), and 6.34(c) illustrate how good the exponential averaging estimator could estimate the actual network utilizations for the hoeffding bounds algorithm while transmitting VoIP, video, and best effort application packets. As it can be seen from the graphics, the estimator performs better estimations for VoIP traffic flows as for video and best effort traffic flows. The graphics also show the degree of utilization of their respective class bandwidths. In the case of best effort, it can be seen that only a small portion of the bandwidth is utilized by its applications. Figure 6.34(d) displays the delay behaviours of VoIP, video and best effort traffic flows. It can be noticed from the graphic that the traffic classes show variable delay behaviours, but their highest delays are still below the maximum tolerable delay thresholds for their applications.

Acceptance Region Tangent at Origin

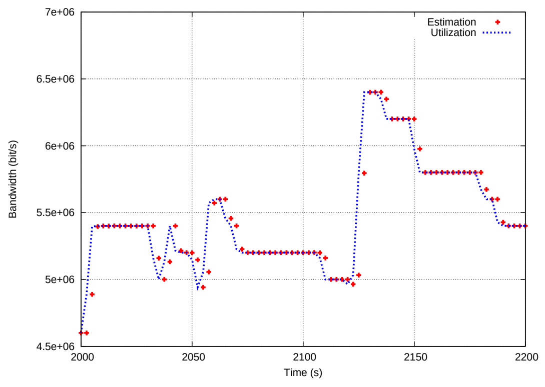
The performance results achieved by the acceptance region tangent at origin algorithm are illustrated in Table 6.34 and in Figure 6.35.

Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	15791637	1972395375	92061	3.959	0.5539	80.96
Video	255508	1074959017	0	4.820		
Best effort	573116	69988585	0	1.639		

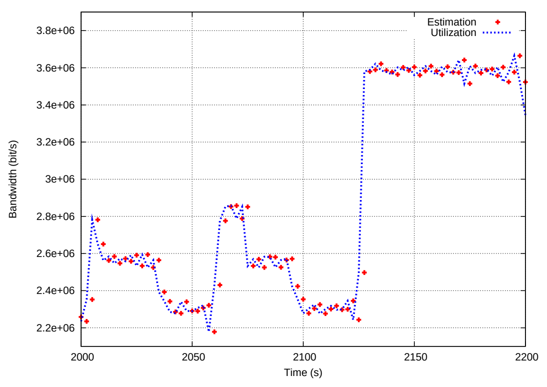
Table 6.34: Acceptance region tangent at origin performance results

From Table 6.34, it can be seen that the algorithm achieves an acceptable total network utilization while keeping low packet drop rate, inspite of the fact that the best effort applications are sending at low rate. This result shows the efficiency of the dynamic allocation mechanism for this algorithm.

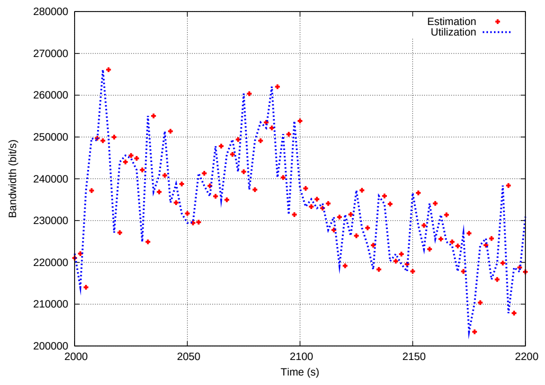
Figures 6.35(a), 6.35(b), and 6.35(c) illustrate how good the point sample estimator could estimate the actual network utilizations for the acceptance region tangent at origin algorithm while transmitting different traffic. It can be observed from the graphics that the estimator performs well for all the flows of the three traffic classes. The grahpics also depict the degree of utilization of their respective class bandwidths. It can be noticed that only small portion of the best effort class bandwidth is actually utilized. Figure 6.35(d) shows the delay behaviours of VoIP, video and best effort traffic flows. As displayed on the graphic, the traffic classes show variable delay behaviours but their highest delays are still below the maximum acceptable delay boundaries for their respective applications.



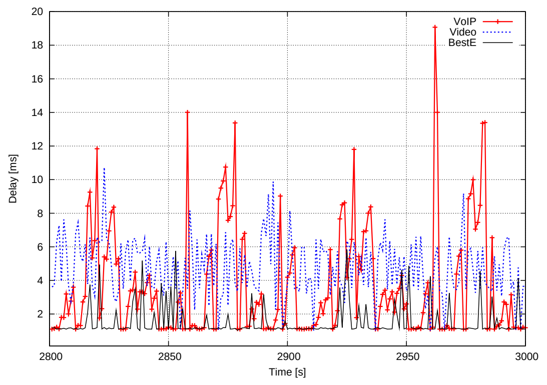
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.35: Actual and estimated utilization: ACTO algorithm; eight-node topology; scenario two

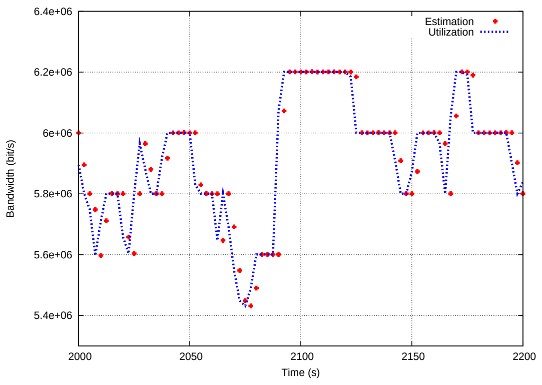
Acceptance Region Tangent at Peak

The performance results achieved by the acceptance region tangent at peak algorithm are illustrated in Table 6.35 and in Figure 6.36.

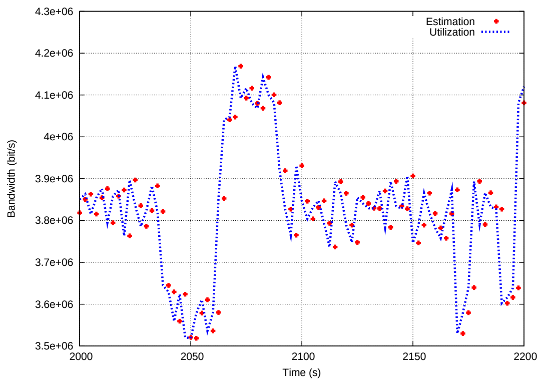
Traffic class	**Received packets** (pkt)	**Received packets size** (byte)	**Lost packets** (pkt)	**Average packets delay** (ms)	**Total drops** (%)	**Total utilization** (%)
VoIP	17748730	2217161150	676000	15.476	3.6055	98.13
Video	341567	1438923892	0	5.961		
Best effort	658876	80694620	0	1.908		

Table 6.35: Acceptance region tangent at peak performance results

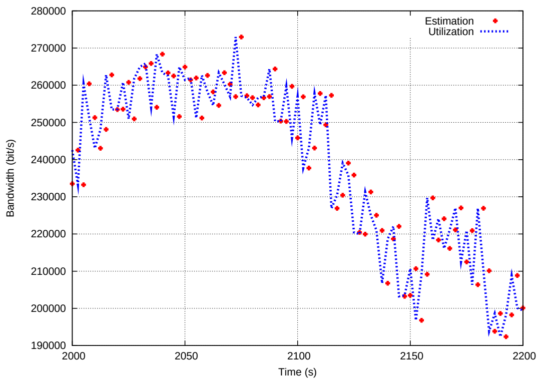
As shown in Table 6.35, the algorithm obtaines high total network utilization level while suffering lower packet drop rate inspite of the fact that the best effort applications are sending their traffic at low rate. This algorithm achieves the highest total network utilization level among the other algorithms for this simulation scenario. Figure 6.36 depicts the behaviour of the point sample estimator and the acceptance region tangent at peak admission control algorithm for this simulation scenario.



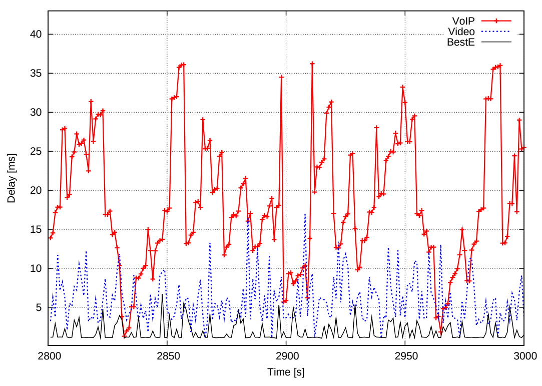
(a) VoIP traffic



(b) Video traffic



(c) Best effort traffic



(d) VoIP, video and best effort packets delays

Figure 6.36: Actual and estimated utilization: ACTP algorithm; eight-node topology; scenario two

Figures 6.36(a), 6.36(b), and 6.36(c) illustrate the performance of the ACTP policy algorithm and the point sample estimator. It can be seen from the graphic that the estimator does good estimations of actual utilizations for the flows of the three traffic classes. The graphics also show the degree of utilization of their respective class bandwidths. It is noticable that only a small portion of the best effort class bandwidth is utilized. Figure 6.36(d) displays the delay behaviours of VoIP, video and best effort traffic flows. As shown on the graphic, the traffic classes exhibit variable delay behaviours, but their highest delays are still below the maximum acceptable delay boundaries for their respective applications.

6.2.3 Performance Comparison of the Two Network Topologies

In this section, the results achieved by the two-node and the eight-node network topologies using the dynamic bandwidth allocation mechanism and the two simulation scenarios are compared. The essence of this comparison is to highlight the effects of these network topologies in the performances obtained by the four measurement-based admission control algorithms and to create a basis for making possible assumptions and further research in this area.

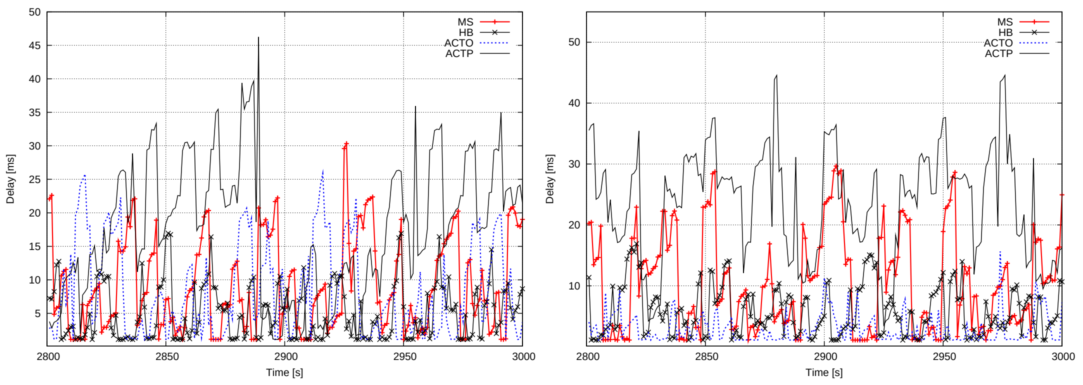
There are totally four groups of simulations to be compared. Two results were achieved using each of the two network topologies. The comparison is based on the achieved network utilizations, drop rates and packet delays experienced by the traffic classes using the four measurement-based admission control algorithms. Table 6.36 displays an overview of the achieved results.

Two-node network topology			**Eight-node network topology**		
Simulation scenario one			Simulation scenario one		
Admission algorithm	Total drops (%)	Total util (%)	Admission algorithm	Total drops (%)	Total util (%)
MS	0.3960	93.57	MS	0.3526	93.14
HB	0.0741	86.44	HB	0.0765	87.60
ACTO	0.5261	83.35	ACTO	0.0727	80.80
ACTP	8.1650	99.28	ACTP	6.9464	99.38
Simulation scenario two			Simulation scenario two		
Admission algorithm	Total drops (%)	Total util (%)	Admission algorithm	Total drops (%)	Total util (%)
MS	0.3742	91.98	MS	0.3454	92.30
HB	0.0975	87.01	HB	0.1072	86.83
ACTO	0.1879	83.63	ACTO	0.5539	80.96
ACTP	4.4837	98.44	ACTP	3.6055	98.13

Table 6.36: Performance comparison of the two network topologies using dynamic allocated bandwidth

The performance results in Table 6.36 illustrate the achieved utilization and the drop rates of the four MBAC algorithms simulated. It is interesting to notice the difference in performance between the two network topologies. For example, the equivalent bandwidth based on hoeffding bound admission control algorithm obtained both a lower drop rate and a higher utilization level for simulation scenario two using the two-node network topology as compared to its achievement for the same scenario using the eight-node network topology. On the other side, the measured sum algorithm achieved better drop rate and higher network utilization for simulation scenario two using the eight-node topology as compared to its achievement for the same scenario using the two-node network topology.

The delay behaviours are here compared only for the VoIP traffic, since it has the most strict delay requirements from a network. Figure 6.37 depicts the delay experienced by traffic flows that are controlled by the four MBAC algorithms for simulation scenario one using both network topologies.



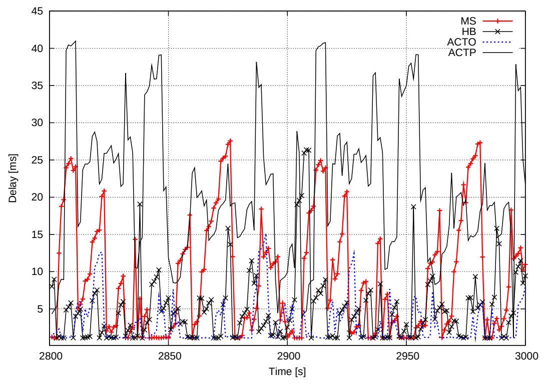
(a) With two-node topology (b) With eight-node topology

Figure 6.37: Delays experienced by VoIP traffic for scenario one

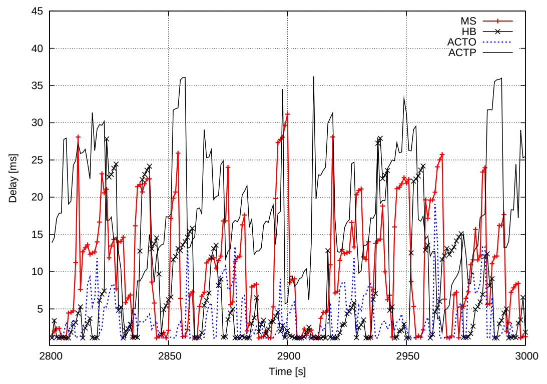
Figure 6.37(a) presents the packet delays experienced by the traffic flows with the four MBAC algorithms using the two-node network topology and Figure 6.37(b) illustrated the packet delays sensed by the traffic flows with the four MBAC algorithms using the eight-node nework topology. The figures depict the delays sensed by the VoIP traffic class packets while traversing the multiservice bottleneck link. Taking some closer look at the graphics one could notice the different delays exhibitted by the same algorithms between the two network topologies. For example, the acceptance region tangent at origin shows lower delay behaviours with the eight-node network topology as compared to its delays with the two-node network topology.

The delay behaviours shown by the VoIP traffic flows with the admission control algorithms for simulation scenario two using both of the network topologies are presented in Figure 6.38.

Figure 6.38(a) presents the packet delays experienced by the VoIP traffic flow with the four MBAC algorithms using the two-node network topology and Figure 6.38(b) illustrates the packet delays revealed by the VoIP traffic flows with the four MBAC algorithms using the eight-node nework topology. The graphics show different delay behaviours for the same algorithm type while using the two network topologies. Observing the graphics carefully one notices that the measured sum algorithm exhibits lower delay behaviours with the two-node network



(a) With two-node topology



(b) With eight-node topology

Figure 6.38: Delays experienced by VoIP traffic for scenario two

topology as compared to its delays with the eight-node network topology.

6.3 Performance Comparison: Static vs. Dynamic Bandwidth Allocation Mechanism

This section presents the comparison of the performance results achieved by the four measurement-based admission control algorithm for the static and dynamic bandwidth allocation mechanisms using the two network topologies and the two simulation scenarios.

The comparison offers the opportunity to study the importance of these mechanisms for the proposed multiservice framework. It also creates ideas for possible further enhancement of the framework and more research in this area. Before going into the performance results comparison, some general comments about the results should be noted.

6.3.1 General Comments About Achieved Results

In the above evaluated tables showing the achieved performance results, it is noticable that the number of received packets from best effort application traffic flows is bigger than from video traffic flows inspite of the fact that video traffic class has larger bandwidth share than best effort traffic class. This situation is caused by the variable large sizes of the video packets. Taking a closer look at the size of the received packets in byte, one can then prove that the received video traffic flows are actually bigger than the received best effort traffic flows. This situation is noticed in all simulations in this thesis where video trace is used.

From the achieved average delay and the plotted delay behaviours, it can be noticed that the VoIP traffic, which has the highest priority and is most sensitive to delay shows in the most cases[1] the largest delays behaviours. This is due to the fact that the VoIP traffic class has the highest number of flows in the network and in the multiservice link each traffic class is served with a seperate queue. Therefore, the VoIP queue is steadily filled and due to that reason, the packets suffers larger delays. Regardless of this situation, the experienced delay is below the maximium defined tolerable delay threshold of 150-200 ms for VoIP applications [19].

6.3.2 Selected Results and Comparison

There are many results achieved in this thesis but for this comparison, only selected results are evaluated. Based on these results, the effects of the static and dynamic bandwidth allocation mechanism can be clearly observed. The comparison is carried out with the performance results obtained for the second simulation using the two network topologies in each bandwidth allocation mechanism.

Table 6.37 presents the performance results selected for the comparison.

Static bandwidth allocation mechanism						**Dynamic bandwidth allocation mechanism**					
Two-node network topology						**Two-node network topology**					
Simulation scenario two						Simulation scenario two					
AC	Total drops (%)	Total util (%)	VoIP avg delay (ms)	Video avg delay (ms)	bestE avg delay (ms)	AC	Total drops (%)	Total util (%)	VoIP avg delay (ms)	Video avg delay (ms)	bestE avg delay (ms)
MS	0.1582	90.74	7.662	5.227	1.818	MS	0.3742	91.98	8.214	5.401	1.850
HB	0.0148	80.55	4.456	5.115	1.724	HB	0.0975	87.01	5.874	5.093	1.791
ACTO	0.0343	72.85	3.366	5.011	1.610	ACTO	0.1877	83.63	4.208	4.974	1.691
ACTP	1.1976	95.16	11.947	5.465	1.889	ACTP	4.4837	98.44	17.048	5.380	1.939
Eight-node network topology						**Eight-node network topology**					
Simulation scenario two						Simulation scenario two					
AC	Total drops (%)	Total util (%)	VoIP avg delay (ms)	Video avg delay (ms)	bestE avg delay (ms)	AC	Total drops (%)	Total util (%)	VoIP avg delay (ms)	Video avg delay (ms)	bestE avg delay (ms)
MS	0.1727	90.68	7.267	5.072	1.837	MS	0.3454	92.30	8.678	5.308	1.870
HB	0.0126	80.28	4.747	5.638	1.729	HB	0.1072	86.83	6.362	5.494	1.814
ACTO	0.1328	73.94	5.063	5.118	1.771	ACTO	0.5539	80.96	3.959	4.820	1.639
ACTP	1.2997	95.07	10.231	6.008	1.887	ACTP	3.6055	98.13	15.476	5.961	1.908

Table 6.37: Performance comparison of the static and dynamic bandwidth allocation mechanism

As it can be observed in Table 6.37, the dynamic bandwidth allocation mechanism achieved up to 11 % total network utilization difference in comparison to the static bandwidth allocation mechanism. It can be noticed that the dynamic bandwidth allocation mechanism shows different efficiency levels for the four measurement-based admission control algorithms. For example the acceptance region tangent

[1]The ACTO algorithm achieved lower average delay for VoIP as for video in all the cases except for simulation scenario one using static allocated bandwidth and two-node network topology

at origin admission control algorithm achieved about 11 % difference using the two-node network topology whereas the measured sum algorithm achieved about 1.4 % differece using the same network topology.

Furthermore, it is noticed that generally the drop rates are lower for the static bandwidth allocation mechanism. This can be attributed to the lower number of traffic flows traversed in the network while using this mechanism for simulation scenario two.

6.4 Summary

The proposed multiservice framework designed for simulating the performance effects of simultaneous transmission of multi-class traffic flows over a packet network is thoroughly evaluated in this chapter. There are three traffic classes (VoIP, video, and best effort) defind with different class priorities for this framework. The VoIP class has the highest priority, followed by the video class and the best effort has the lowest priority. The framework includes two powerful bandwidth allocation mechanism, which are responsible for differentiated resource allocation to these traffic classes according to their priorities.

Two network topologies and two simulation scenarios were defined for this evaluation. The first simulation scenario demonstrates a standard situation in a packet network where all the traffic classes are striving to make full use of their class bandwidth. This scenario evaluates the performance of the multiservice framework in such situations. The second simulation scenario depicts a situation where the best effort class bandwidth is underutilized. The essence of this scenario is to show and evaluate the efficiency of the dynamic bandwidth allocation mechanism with the four measurement-based admission control algorithms. For the evaluation, the simulation scenarios are ran for the static bandwidth allocation mehanism using each of the two network topologies. The same is also done for the dynamic bandwidth allocation mechanism. The obtained performances of the four MBAC algorithms are measured.

The measurement parameters include the number of packets recieved for each traffic class, the size of the packets in byte, the number of lost packets, the average delay experienced by the packets, the total network drop rate, and the total network utilization level achieved. The achieved performances using each of the network topologies for each of the bandwith allocation mechanism are compared to investigate the effects of the network topologies in the total network performance. At the end the results achieved for the two bandwidth allocation mechanisms are compared to show the advantages of the dynamic bandwidth allocation mechanism.

7 Conclusions

IP networks are originally designed for best effort services. Due to the introduction of real–time applications, which are sensitive to resources availability and place performance requirements on the underlying networks, the original IP networks could no longer satisfy the new requirements of these applications. Hence, there arises a need for improvement in the IP architecture to support quality of service. The notion of service level agreement laids the basis for specifying and agreeing on certain QoS for applications.

Some QoS technologies evolved over the past decade. The integrated service model together with resource reservation protocol is one of the first QoS mechanism introduced in the IP architecture. The IntServ had the assumption that resources must be explicitly managed by applications in order to meet their QoS requirements. IntServ with RSVP provided a genuine QoS architecture but however, had scalability and operational complexity problems. To solve these problems the IETF introduced the DiffServ model, which is scalable and does not require signalling protocol. Later, MPLS was introduced by IETF as connection–oriented approach to connectionless IP–based networks, and it supports traffic engineering.

Some designs of the integrated service networks use admission control (AC) mechanism to provide quality communication by ensuring resources availability for customer traffic flows. There are different approaches to admission control mechanism. The parameter-based approach uses prespecified traffic characteristics to compute the network load and thus make its admission decision whether to accept or reject a new traffic flow. It has the disadvantage of not utilising the network resources well, thereby causing poor network utilization. The measurement-based approach provided an alternative solution by making an on-line measurement of current network load and based on this measured load, makes its admission decision upon the arrival of a new flow. Four examples of the MBAC algorithms are implemented in the ns-2 tool for network simulations.

The ns-2 tool is an object-oriented discrete event driven simulator targeted for network research. It is widely used today in many institutions and research centers for teaching and carrying out research in internetworking area. It covers a large set of networking protocols, which includes wired networks, wireless networks, and satellite networks. The tool is an open source software package, which is free to download and it is possible to make changes to the source code. The network simulator is written in two programming languages (C++ and OTcl), with the primary aim of supporting detailed simulations and fast testing of different parameter configurations. Simulation scenarios are designed and written only in OTcl.

A single service framework based on integrated service is embedded in ns-2 for evaluating quality of service of a single controlled-load service in a packet network. It is made up of two basic components known as end-to-end signalling mechanism and enhanced link. The enhanced link contains the four measurement-based admission control algorithms. Based on this existing framework, a new multiservice framework is designed and implemented within this thesis for simulating service quality of multi-class traffic flows simultaneously traversing a packet network. The new framework contains two bandwidth allocation mechanisms for sharing the total network resources among the traffic classes according to their priorities.

The new framework is thoroughly investigated by numerous network simulations to evaluate its performance and correctness. Two simulation scenarios and two network topologies are used for this purpose. The achieved performance results proved the capability and efficiency of the framework.

Bibliography

[1] G. Almes, S. Kalidindi, and M. Zekauskas. A One–way Delay Metric for (IPPM). *IETF RFC 2679*, September 1999.

[2] E. Altman and T. Jimenez. *NS Simulator for Beginners*. University de Los Andes, Merida, Venezuela, and ESSI, Sophia-Antipolis, France, December 2003.

[3] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirement for Traffic Engineering Over MPLS. *IETF RFC 2702*, September 1999.

[4] A. Bak, W. Burakowski, F. Ricciato, S. Salsano, and H. Tarasiuk. Traffic Handling in AQUILA QoS IP Network. In *Lecture Notes in Computer Science*, volume 2156, pages 243–260, January 2001.

[5] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A Framework for Integrated Services Operation over DiffServ Networks. *IETF RFC 2998*, November 2000.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *IETF RFC 2475*, December 1998.

[7] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. *IETF RFC 1633*, June 1994.

[8] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). *IETF RFC 2205*, September 1997.

[9] L. Breslau, S. Jamin, and S. Shenker. Measurement-Based Admission Control: What is the Research Agenda? In *Seventh International Workshop on Quality of Service*, pages 3–5, London, UK, June 1999.

[10] L. Breslau, S. Jamin, and S. Shenker. Comments on the Performance of Measurement-Based Admission Control Algorithms. In *Proceeding of the*

Nineteenth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM 2000), volume 3, pages 1233–1242, Tel Aviv, Israel, March 2000.

[11] C. Casetti, J. F. Kurose, and D. F. Towsley. An Adaptive Algorithm for Measurement-based Admission Control in Integrated Services Packet Networks. In *Computer Communications*, volume 23, pages 1363–1376, 2000.

[12] J. Chung and M. Claypool. Ns By Example Tutorial. http://nile.wpi.edu/NS/.

[13] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network; Architecture and Mechanism. In *ACM SIGCOMM '92*, volume 22, pages 14–26, August 1992.

[14] M. Conti, M. Kumar, S.K. Das, and B.A. Shirazi. Quality of Service Issues in Internet Web Services. In *Transactions on computers*, volume 51, pages 593–594. IEEE, June 2002.

[15] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick. A Framework for QoS–based Routing in the Internet. *IETF RFC 2386*, August 1998.

[16] B. Davie, A. Charny, J.C.R Bennett, K. Benson, J.Y Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). *IETF RFC 3246*, March 2002.

[17] A. Demers, S. Shenker, and S. Keshav. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM Symposium proceedings on Communication Architecture & Protocols*, pages 1–12, Austin-Texas, USA., 1989.

[18] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performence Metrics (IPPM). *IETF RFC 3393*, November 2002.

[19] J. Evans and C. Filsfils. *Deploying IP and MPLS QoS for Multiservice Networks*. Morgen Kaufmann, San Francisco, USA, 2007.

[20] K. Fall and K. Varadhan. The NS Manuel (Documentation), May 2008. http://www.isi.edu/nsnam/ns/ns-documentation.html.

[21] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. In *JSAC*, volume 8:3, pages 368–379, April 1990.

[22] S. Floyd. Comments on measurement-based admissions control for controlled-load services. *Technical Report*, July 1996.

[23] L. Georgiadis, R. Gurin, V. Peris, and R. Rajan. Efficient Support of Delay and Rate Guarantees in an Internet. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 106–116, October 1996.

[24] R. J. Gibbens and F. P. Kelly. Measurement-based Connection Admission Control. In *15th International Teletraffic Congress Proceeings*, pages 879–888, Amsterdam, June 1997.

[25] Richard J. Gibbens, Frank P. Kelly, and Peter B. Key. A Decision-Theoretic Approach to Call Admission Control in ATM Networks. In *IEEE Journal on Selected Areas in Communications*, volume 13, pages 1102–1114, August 1995.

[26] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz. Quality of Service Terminology in IP Networks. In *Communication Magazine*, volume 41, pages 153 – 159. IEEE, March 2003.

[27] M. Greis. Marc Greis Tutorial. http://www.isi.edu/nsnam/ns/tutorial/index.html.

[28] M. Grossglauser and D. N. C. Tse. A Framework for Robust Measurement-Based Admission Control. In *IEEE/ACM Transactions on Networking*, volume 7, pages 293 – 309, June 1999.

[29] M. Grossglauser and D. N. C. Tse. A Time-Scale Decomposition Approach to Measurement-based Admission Control. In *IEEE/ACM Transactions on Networking*, volume 11, pages 550 – 563, August 2003.

[30] R. Guerin, H. Ahmadi, and M. Naghshineh. Equivalent Capacity and its application to bandwidth allocation in high-speed networks. In *IEEE Journal on selected areas in communications*, volume 9, pages 968 – 981, September 1991.

[31] AWK User Guide. http://www.gnu.org/manual/gawk/html_node/index.html.

[32] P. Gupta and N. McKeown. Algorithm for Packet Classification. In *IEEE Networks*, volume 15, pages 24–32, March/April 2001.

[33] A. Halimi. *Quality of Service Networking with MPLS*. PhD thesis, Vienna University of Technology, November 2004.

[34] W. C. Hardy. *QoS Measurement and Evaluation of Telecommunication Quality of Service*. John Wiley & Sons Ltd., August 2001.

[35] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. *IETF RFC 2597*, June 1999.

[36] P. Huang. Internet Simulations with the NS Simulator. In *2nd European ns-2 Workshop*, Vienna, Austria, April 2001.

[37] J. Huttunen. Measurement on Differentiation of Internet Traffic. Master's thesis, Helsinki University of Technology, January 2005.

[38] ITU-T Recommendation E.800. *Terms and Definitions related to Quality of Service and Network Performance including Dependability*, August 1994.

[39] ITU-T Recommendation E.860. *Framework of a Service Level Agreement*, June 2002.

[40] ITU-T Recommendation G.1000. *Communications Quality of Service: A Framework and Definitions*, November 2001.

[41] ITU-T Recommendation I.350. *General Aspect of Quality of Service and Network Performance in Digital Networks, Including ISDNs*, March 1993.

[42] ITU-T Recommendation Y.1241. *Support of IP-based Service using IP Transfer Capabilities*, March 2001.

[43] ITU-T Recommendation Y.1540. *Internet Protocol Data Communication Service - IP Packet Transfer and Availability Performance Parameters*, December 2002.

[44] ITU-T Recommendation Y.1541. *Network Performance Objectives for IP-based Services*, May 2002.

[45] J. Glasmann and M. Czermin and A. Riedl. Estimation of Token Bucket Parameters for Videoconferencing Systems in Corporate Networks. *SoftCOM*, pages 10–14, September 2000.

[46] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang. A Measurement-based Admission Control Algorithm for Integrated Service Packet Networks. In *IEEE/ACM Transactions on Networking*, volume 5, pages 56–70, February 1997.

[47] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang. A Measurement-based Admission Control Algorithm for Integrated Service Packet Networks (Extended Version). *IEEE/ACM Transactions on Networking*, February 1997.

[48] S. Jamin and S. J. Shenker. Measurement-based Admission Control Algorithms for Controlled-load Service: A Structural Examination. Technical Report CSE-TR-333-97, Computer Science and Engineering, University of Michigan, April 1997.

[49] S. Jamin, S. J. Shenker, and P. B. Danzig. Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service. In *Proceeding of the sixteenth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM '97)*, volume 3, pages 973 – 980, Kobe, Japan, April 1997.

[50] F. Kelly. Notes on Effective Bandwidths. In *In Stochastic Networks: Theory and Applications*, pages 141–168. Royal Statistical Society Lecture Notes Series, February 1996.

[51] E. W. Knightly. On the Accuracy of Admission Control Tests. In *Proceeding of the International Conference on Network Protocols*, pages 125–133, Atlanta, GA USA., October 1997. IEEE.

[52] Y. Lai and S. Tsai. Unfairness of Measurement-Based Admission Controls in a Heterogeneous Environment. In *Proceedings of Eighth International Conference on Parallel and Distributed Systems*, pages 667 – 674, Kyongju City, South Korea, June 2001.

[53] S. R. Lima, P. Carvalho, and V. Freitas. Distributed Admission Control in Multiservice IP Networks: Concurrency Issues. In *Journal of Communications (JCM)*, volume 1, pages 1–9, June 2006.

[54] S. R. Lima, P. Carvalho, and V. Freitas. Admission Control in Multiservice IP Networks: Architectural Issues and Trends. In *IEEE Communications Magazine*, volume 45, pages 114–121, April 2007.

[55] E. Marilly, O. Martinot, S. Betg-Brezetz, and G. Delgue. Requirements for Service Level Agreement Management. In *Workshop on IP operations and management*, volume 51, pages 57–62. IEEE, June 2002.

[56] B. Mayayo. Call Admission Control for IP Networks. Master's thesis, Vienna University of Technology, August 2004.

[57] A. W. Moore. *Measurement-Based Management of Network Resources*. PhD thesis, Computer Laboratory, University of Cambridge, April 2002.

[58] A. W. Moore. An Implementation-based Comparison of Measurement-Based Admission Control algorithms. In *J. High Speed Networks*, volume 13, pages 87–102, 2004.

[59] J. Moy. OSPF Version 2. *IETF RFC 2178*, July 1997.

[60] M. K. Naveen. Internet Protocol Design & Testing: Network Simulator 2. Technical report, Electrical Communication Engineering, Indian Institute of Science, 2006.

[61] NS2 Download Page. http://www.isi.edu/nsnam/ns/ns-build.html.

[62] NS2 Home Page. http://www.isi.edu/nsnam/ns/.

[63] Object-Oriented Tcl Tutorial. http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl/tutorial.html, September 1995.

[64] H. Ohnishi, T. Okada, and K. Noguchi. Flow Control Schemes and Delay-Loss Tradeoff in ATM Networks. In *IEEE Journal. Selected Areas in Communications (Special Issue: Broadband Packet Communications)*, volume 6, pages 1609–1616, December 1988.

[65] Network Animator (Nam) Manual Page. http://www.isi.edu/nsnam/ns/tutorial/nam.txt.

[66] A. K. Parakh and G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single–Node Case. In *IEEE/ACM Transactions on Networking*, volume 1, pages 344 – 357, June 1993.

[67] A.K. Parakh and G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple–Node Case. In *IEEE/ACM Transactions on Networking*, volume 2, pages 137 – 150, April 1994.

[68] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. *IETF RFC 2330*, May 1998.

[69] E.C.K. Poh and H.T. Ewe. IPv6 Packet Classification based on Flow Label, Source and Destination Adresses. In *IEEE Proceeding of the Third International Conference on Information Technology and Application*, volume 2, pages 659–664, Sydney, Australia, July 2005.

[70] J. Qiu. Measurement-Based Admission Control in Integrated-Service Networks. Master's thesis, Rice University, Houston Texas, April 1998.

[71] J. Qiu and E. W. Knightly. QoS Control via Robust Envelope-Based MBAC. In *Sixth International Workshop on Quality of Service*, pages 62–64, Napa, CA, USA, May 1998.

[72] E. Rosen and Y. Rekhter. BGP/MPLS VPNs. *IETF RFC 2547*, March 1999.

[73] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. *IETF RFC 3031*, January 2001.

[74] Running Ns under Window Using Cygwin. http://nsnam.isi.edu/nsnam/index.php/ Running_Ns_and_Nam_Under_Windows_9x/2000/XP_Using_Cygwin.

[75] H. Saito and K. Shiomoto. Dynamic Call Admission Control in ATM Networks. In *IEEE Journal on Selected Areas in Communications*, volume 9, pages 982–989, September 1991.

[76] S. Shenker, C. Partridge, and R. Guerin. Specification for Guarateed Quality of Service. *IETF RFC 2212*, September 1997.

[77] B. Statovci-halimi and A. Halimi. QoS Management through Service Level Agreements: A Short Overview. *e&i*, (6):243–246, June 2004.

[78] G. Stylianos, T. Panos, and P. George. Joint Measurement- and Traffic Descriptor-based Admission Control at real-time Traffic Aggregation Points. In *IEEE International Conference on Communication*, volume 4, Paris, France, June 2004.

[79] Tcl Tutorial. http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html, September 2006.

[80] D. N. C. Tse and M. Grossglauser. Measurement-Based Call Admission Control: Analysis and Simulation. In *Proceedings of the sixteenth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM '97)*, pages 981 – 989, Kobe, Japan, April 1997.

[81] Z. Turányi, A. Veres, and A. Oláh. A Family of Measurement-based Admission Control Algorithms. In *IFIP TC6/WG6.3 Seventh International Conference on Performance of Information and Communication Systems*, volume 127, Lund, Schweden, May 1998. Chapman & Hall.

[82] S. Wright. Admission Control in Multi-Service IP Networks: A Tutorial. In *IEEE Communications Surveys & Tutorials*, volume 9, pages 72–87, 2nd Quarter 2007.

[83] J. Wroclawski. Specification of the Controlled-Load Network Element Service. *IETF RFC 2211*, September 1997.

[84] X. Xiao, B. Bailey, A. Hannan, and L.M. Ni. Traffic Engineering with MPLS in the Internet. In *IEEE Networks*, volume 14, pages 28–33, March/April 2000.

[85] X. Xiao and L. M. Ni. Internet QoS: A Big Picture. In *IEEE Networks*, volume 13, pages 8–18, March/April 1999.

Glossary

AC	**(Admission Control)** A mechanism to control the network load by admitting a new flow into a network, if there is enough resource for the new flow and the QoS commitment of the existing flows will not be violated.
ASP	**(Application Service Provider)** A business that provide computer–based service to customer over a network.
ATM	**(Asynchronous Transfer Mode)** A cell relay packet switching network and data link layer protocol which encodes data traffic flows into a small fixed–sized cell.
BGP	**(Border Gateway Protocol)** A core routing protocol of the Internet. Which can be used in MPLS for label distribution
CBR	**(Constant Bit Rate)** A data traffic source that sends packets at constant bit rates.
CLS	**(Controlled-Load Service)** A QoS mechanism which does not assure strict bounds on delay and packet loss. It provide a service closely equivalent to that provided to uncontrolled best effort traffic under lightly loaded conditions.
Compiled Hierarchy	A set of classes written in C++ which has dependancy relationships
CoS	**(Class of Service)** A service classification mechanism in IP networks. That is used to group packet flows into service classes.

CR–LDP	**(Constraint–based Routing Label Distribution Protocol)** An extension of LDP for capabilities such as setup path beyond what is available for the routing protocol.
DiffServ	**(Differential Service)** A computer–networking architecture that specify a simple, scalable, and coarse–grained for classifying, managing network traffic flows and providing quality of service on modern IP network
DSCP	**(Differentiated Service Code Point)** is a field in the header of IP packets for packet classification purpose.
EA	**(Exponential Averaging)** A measurement mechanism best used to estimate the network load for the HB algorithm.
ETSI	**(European Telecommunication Standard Institute)** An Independent non-for-profit standardization organisation of the telecommunication industry in Europe with worldwide projection.
FEC	**(Forward Equivalence Class)** A term used in MPLS to describe a set of packets with similar and or identical characteristics which maybe forwarded the same way; that is, they maybe bounded to the same label.
FIFO	**(First In, First Out)** The most basic queue scheduling discipline. In FIFO, all packets are treated equally by placing them into a single queue, and then servicing them in the same order that they were placed into the queue.
FQ	**(Fair Queueing)** A scheduling Algorithm used in computer and telecommunication networks to allow multiply packet flow to fairly share the link capacity.
FTP	**(File Transfer Protocol)** is a network protocol used to transfer data from one computer to another through a network, such as the Internet. In ns-2, it is a simulated application traffic source.

GS	**(Guaranteed Service)** A service which provides assured level of bandwidth and strict bounds on end-to-end delay and packet loss for conforming flows.
HB	**(Hoeffding Bounds)** An MBAC algorithm which computes the equivalent bandwidth for a set of flows using the Hoeffding bounds.
IETF	**(Internet Engineering Task Force)** An open standard organisation with no formal membership. It deals with standard for TCP/IP and Internet protocol suite.
Interpreted Hierarchy	A set of classes written in OTcl which has dependancy relationships
IntServ	**(Integrated Service)** A networking architecture that specifies the elements to guarantee quality of service on IP networks.
IP	**(Internet Protocol)** A conectionless transport protocol
IPPM	**(IP Performance Metrics)** An IETF working group developing a set of metrics that can be applied to the quality, performance and reliability of Internet data delivery service.
ISI	**(Information Science Institute)** An institute of the university of southern California that manages ns-2
ISP	**(Internet Service Provider)** An organisation that offer users access to the Internet and related services.
ITU	**(International Telecommunication Union)** An international organisation established to regulate radio and telecommunication standards
ITU–T	**(ITU Telecommunication)** The ITU telecommunication standardization sector.
LDP	**(Label Distribution Protocol)** A protocol defined by the IETF for the purpose of distributing labels in an MPLS environment.

LSP	**(Label Switched Path)** A path through an MPLS network set up by a signalling protocol such as RSVP–TE.
LSR	**(Label Switch Router)** A type of router located at the middle of a MPLS network. It is responsible for switching the labels used to route packets
MBAC	**(Measurement-Based Admission Control)** An AC approach that uses on-line measurement to measure the network current load.
MPLS	**(Multiprotocol Label Switching)** In computer networking and telecommunication, it is a data–carrying mechanism that belongs to the family of packet switched networks
MS	**(Measured Sum)** An MBAC algorithm which uses measurement to estimate the load caused by the existing network traffic flows.
MTU	**(Maximum Transmission Unit)** The size of the largest packet that a network protocol can transmit.
Nam	**(Network Animator)** A tool used to visualize simulation results gathered in a special trace format.
ns-2	**(Network Simulator version 2)** A network simulation tool used to verify the performance of the algorithm implemented in this thesis.
OSPF	**(Open Shortest Path First)** A dynamic routing protocol for use in Internet protocol networks to navigate data packets from source to destination.
OTcl	**(Object-oriented Tcl)** Used for configuration and writing simulation scripts in ns-2.
PBAC	**(Parameter-Based Admission Control)** An AC approach which uses prespecified traffic parameter to compute the network load.
PHB	**(Per–Hop Behaviour)** Defines the policy and priority applied to a packet when traversing a hop in a DiffServ network.

PQ	**(Priority Queueing)** The arrangement of jobs to be carried out in a list according to their relative importance, with the most important first.
PS	**(Point Sample)** A measurement mechanism best used for both TP and TO algorithm to estimate the network load.
QoS	**(Quality of Service)** A collective effects of service performance which determines the degree of satisfaction of a user on the service.
RIP	**(Routing Information Protocol)** A dynamic routing protocol used in local area network to create paths for delivering data packets from source to destination.
RR	**(Round Robin)** A scheduling algorithm which repeatedly runs through a list of users, giving each user opportunity to service its request in succession.
RSVP	**(Resource ReSerVation Protocol)** A transport layer protocol designed to reserve resources across a network for an integrated service Internet.
RSVP–TE	**(Resource ReSerVation Protocol–Traffic Engineering)** An extension of the RSVP protocol for traffic engineering. Which can serve the purpose of distributine label in MPLS.
SAP	**(Service Access Point)** An identifying label for network endpoints used in open source interconnection (OSI) networking
SLA	**(Service Level Agreement)** A formal agreement between two or more entities that is reached after a negotiating activity with the scope to assess service characteristics, responsibilities and priorities of every party
SLS	**(Service Level Specification)** Represents the technical part of an SLA
Tcl	**(Tool Command Language)** A scripting language used for scripted applications.

TCP	**(Transmission Control Protocol)** A transport layer four protocol, that is one of the core protocols of the Internet protocol suite. In ns-2, it is implemented as an agent.
TELNET	**(Terminal Emulation)** Generally a program for remote network accessing. In ns-2, it is a simulated application traffic source.
TO	**(Tangent at Origin)** An MBAC algorithm which is based on the tangent to the equivalent bandwidth at the origin.
Token Bucket	A common algorithm used to control the amount of data that is injected into a network, allowing for bursts of data to be sent.
TP	**(Tangent at Peak)** An MBAC algorithm which is based on the tangent at the peak of the equivalent bandwidth curve computed from the Chernoff Bounds.
TTL	**(Time To Live)** A limit of the period of time or number of transmission in computer networking that a unit of data (e.g a packet) can experience before it should be discarded.
TW	**(Time-Window)** A simple measurement mechanism used to estimate a network load. It is best suited to the MS algorithm.
UDP	**(User Datagram Protocol)** A transport layer protocol that is one of the core protocols in the Internet protocol suite. In ns-2, it is implemented as an agent.
UML	**(Unified Modeling Language)** A standardized general-purpose modelling language used in the field of software engineering.
VoIP	**(Voice over IP)** A traffic type that specifies real–time audio data
WFQ	**(Weighted Fair Queueing)** A data packet scheduling technique allowing different scheduling priorities to statistically multiplexed data flows.
WRR	**(Weighted Round Robin)** A best–effort connection scheduling discipline.

Index

List of Source Codes

Listing 1: Admission control header file

```

/*
    **********************************************************************
 * Author:              Vincent Chimaobi Emeakaroha
 * Email:               e0027525@student.tuwien.ac.at
 * Address:             Institute for broadband communication
 *                              Vienna university of Technology (TU)
 **********************************************************************
    */

#ifndef ns_adc_h
#define ns_adc_h

/* Activate dynamic bandwidth borrowing */
#define BORROW

/* Activate outputting of debug messages */
#define DBUG_MS

#include "estimator.h"

/* The maximum number of measurement objects to be used for
    estimation */
#define CLASS 10

/* The admission control base class declaration */
class ADC : public NsObject {

/* declaration of public member functions */
public:
        ADC();
        int command(int,const char*const*);
        virtual int admit_flow(int cl,double r, int b)=0;
        virtual void rej_action(int,double,int){};
        virtual void teardown_action(int,double,int){};
        inline void recv(Packet*, Handler *){}
        inline void setest(int cl,Estimator *est) {est_[cl]=est;
```

```
                                                        est_[cl]->setactype
                                                            (type_);}
        double peak_rate(int cl,double r,int b) {return r+b/est_[cl
            ]->period();}
        char *type() {return type_;}

/* Declaration of protected member variables */
protected:
        Estimator *est_[CLASS];   /* Array of estimator pointer
            variables */
        double bandwidth_;        /* The network total bandwidth */
        char *type_;              /* The particular type of algorithm
             in play */
        Tcl_Channel tchan_;       /* A variable for attaching trace
            file descriptors */
        int src_;                 /* The source address of a packet
            */
        int dst_;                 /* The destination address of a
            packet */
        int backoff_;             /* A boolen variable to regulate HB
            , ACTO & ACTP admission process */
        int dobump_;              /* A boolen variable to regulate
            the change on avload in ACTO & ACTP admission process */
};

#endif
```

Listing 2: Admission control base class source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/adc.cc,v 1.8
            2005/08/26 05:05:27 tomh Exp $";
#endif

#include "adc.h"
#include <stdlib.h>

/* The constructor function definition */
ADC::ADC() :bandwidth_(0), tchan_(0)
{
        /* Bind these member variables so that they can be
            accessible for OTcl scripts */
        bind_bw("bandwidth_",&bandwidth_);
        bind_bool("backoff_",&backoff_);
        bind("src_", &src_);
        bind("dst_", &dst_);
        bind_bool("dobump_", &dobump_);
}

/* The command function to execute member functions invoked from
```

```
    OTcl scripts */
int ADC::command(int argc,const char*const*argv)
{

        Tcl& tcl = Tcl::instance();    /* get the Tcl instance for passing results back and forward b/w C++ and OTcl */
        if (argc==2) {
                if (strcmp(argv[1],"start") ==0) {
                        /* $adc start for single service controlled load */
                                est_[1]->start();
                        }
                        return (TCL_OK);
        } else if (argc == 3) {
                if (strcmp(argv[1],"start") ==0) {
                        /* $adc start for multiservice 3 different classes of flows  added for the multiservice framework*/
                        int cl_num = atoi(argv[2]); /* extract the number of estimators to start */
                        for(int b = 1; b <= cl_num; b++){
                                est_[b]->start();       /* start the estimators */
                        }
                        return (TCL_OK);
                }
        }else if (argc==4) {
                if (strcmp(argv[1],"attach-measmod") == 0) {
                        /* set the measurement object instantiated for a class of traffic */
                        /* $adc attach-measmod $meas $cl */
                        MeasureMod *meas_mod = (MeasureMod *) TclObject::lookup(argv[2]);    /* get the measurement object */
                        if (meas_mod== 0) {
                                tcl.resultf("no measuremod found");
                                return(TCL_ERROR);
                        }
                        int cl=atoi(argv[3]);    /* extract the class of traffic */
                        est_[cl]->setmeasmod(meas_mod);   /*set the measurement object */
                        return(TCL_OK);
                } else if (strcmp(argv[1],"attach-est") == 0 ) {
                        /* set the estimator object instantiated for a class of traffic */
                        /* $adc attach-est $est $cl */
                        Estimator *est_mod = (Estimator *)TclObject::lookup(argv[2]);    /* get the estimator object */
                        if (est_mod== 0) {
                                tcl.resultf("no estmod found");
                                return(TCL_ERROR);
                        }
                        int cl=atoi(argv[3]);   /* extract the class of traffic */
                        setest(cl,est_mod);    /* set the estimator object */
                        return(TCL_OK);
                }
        }
        else if (argc == 3) {
                if (strcmp(argv[1], "attach") == 0) {
                        /* attach a file descriptor for writing trace events */
                        int mode;
                        const char* id = argv[2];
                        tchan_ = Tcl_GetChannel(tcl.interp(), (char *)id, &mode);
                        if (tchan_ == 0) {
                                tcl.resultf("ADC: trace: can't attach %s for writing", id);
                                return (TCL_ERROR);
                        }
                        return (TCL_OK);

                }
                if (strcmp(argv[1], "setbuf") == 0) {
                        /* some sub classes actually do something here */
                        return(TCL_OK);
                }


        }
        return (NsObject::command(argc,argv));
}
```

Listing 3: Measured sum admission control source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/ms-adc.cc,v 1.7 2005/08/26 05:05:27 tomh Exp $";
#endif


#include "adc.h"
#include <stdio.h>
#include <stdlib.h>
#include "bandwidthAlloc.h"

```

```
/* The measured sum class declaration */
class MC_MS_ADC : public ADC {

/* public member functions declaration */
public:
        MC_MS_ADC();
        void rej_action(int, double,int);

/*protected member functions and member variables declaration */
protected:
        int admit_flow(int,double,int);
        /* the get rate member function definition to return the
            rate of traffic flows */
        inline virtual double get_rate(int /*cl*/, double r,int /*b
            */)
                { return r ; };
        double utilization_;    /* member variable to hold the value
            configured for the utilization factor */

        /* Member variable for the dynamic bandwidth borrowing
            mechanism */
        double voClAvgLd, viClAvgLd, stdClAvgLd;
        BandwidthAlloc bw;
};

/* The class constructor definition */
MC_MS_ADC::MC_MS_ADC()
{
        /* Bind this variable so that it will be configured from
            OTcl scripts */
        bind("utilization_",&utilization_);
        type_ = new char[5];
        strcpy(type_, "MSAC");          /* set the type of admission
            algorithm in use */

        voClAvgLd = 0.0; viClAvgLd = 0.0; stdClAvgLd = 0.0;
        /* make the total network bandwidth available to the
            bandwidth sharing function */
        //bw = new BandwidthAlloc () ;

        /* Initialize the bandwidth sharing and dynamic bandwidth
            borrowing mechanism */
        bw.bwAllocInit(bandwidth_);
}

/* A member function for decreasing the average network load when a
    flow is rejected */
void MC_MS_ADC::rej_action(int cl,double r,int b)
{
        double rate = get_rate(cl,r,b);         /* get the rate of
            the flow */
        est_[cl]->change_avload(-rate);         /* reduce the flow
            class average network load by the flow rate */
}

/* The member function responsible for admitting flow into the
    network */
int MC_MS_ADC::admit_flow(int cl,double r,int b)
{
        double rate = get_rate(cl,r,b);  /* get the flow rate */

        /* select the class of the traffic flow */
        switch(cl)
        {
        case 1:
                /* Admission point for voip flow traffic class. this
                    class has the highest priority */
                voClAvgLd = rate+est_[cl]->avload();
                if (voClAvgLd < utilization_* bw.cl_bw[cl -1]) {
                        est_[cl]->change_avload(rate);          /*
                            artificially adjust the voip class
                            average load */
#ifdef BORROW
                        /* function call to dynamically borrow
                            bandwidth from standard class */
                        bw.voipBorrowBw(voClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print out the flow id, average load and
                            class bandwidth values*/
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                            voClAvgLd, utilization_*bw.cl_bw[cl-1]);
#endif
                        return 1;
                }
                return 0;
                break;

        case 2:
                /* Admission point for video flow traffic class.
                    this class has middle priority */
                viClAvgLd = rate+est_[cl]->avload();
                if (viClAvgLd < utilization_* bw.cl_bw[cl -1]) {
                        est_[cl]->change_avload(rate);          /*
                            artificially adjust the video class
                            average load */
#ifdef BORROW
                        /* function call to dynamically borrow
                            bandwidth from standard class */
                        bw.videoBorrowBw(viClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print out the flow id, average load and class bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl, viClAvgLd, utilization_*bw.cl_bw[cl-1]);
#endif
                        return 1;

                }
                return 0;
                break;

        case 3:
                /* Admission point for standard flow traffic class. this class has lowest priority */
                stdClAvgLd = rate+est_[cl]->avload();
                if (stdClAvgLd < utilization_* bw.cl_bw[cl -1]) {
                        est_[cl]->change_avload(rate);          /* artificially adjust the standard class average load */
#ifdef BORROW
                        /* function call to recover the dynamical borrowed bandwidth from voip and video classes */
                        bw.stdRecoverBw(voClAvgLd, viClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print out the flow id, average load and class bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl, stdClAvgLd, utilization_*bw.cl_bw[cl-1]);
#endif
                        return 1;
                }
                return 0;
                break;

        default:
                printf("Invalid flow id\n");
                return 0;
        }

}

/* A static linkage class to make the MC_MS_ADC class objects accessible from Intepreted hierarchy */
static class MC_MS_ADCClass : public TclClass {
public:
        MC_MS_ADCClass() : TclClass("ADC/MC_MS") {}
        TclObject* create(int,const char*const*) {
                return (new MC_MS_ADC());
        }
}class_mc_ms_adc;
```

Listing 4: Hoeffding bounds admission control source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/hb-adc.cc,v 1.6
            2005/08/26 05:05:27 tomh Exp $";
#endif


#include "adc.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "bandwidthAlloc.h"


/* The hoeffding bound class declaration */
class MC_HB_ADC : public ADC {

/* Public member functions declarations */
public:
        MC_HB_ADC();
        void teardown_action(int,double,int);
        void rej_action(int,double,int);

/* Protected member function declarations */
protected:
        int admit_flow(int,double,int);
        int rejected_;                  /* a member variable to
            indicate that a flow is rejected admission */
        double epsilon_;                /* a member variable that
            holds the equivalent bandwidth value in the decision
            algorithm */
        double  sump2[3];               /* array member variable to
            store the sum of the square of peak rates of the three
            different traffic flows */

        /* Member variable for the dynamic bandwidth borrowing
            mechanism */
        double voClAvgLd, viClAvgLd, stdClAvgLd;
        BandwidthAlloc bw;
};

/* class constructor definition */
MC_HB_ADC::MC_HB_ADC() : rejected_(0)
```

```
{
        /* bind this member variable so that it can be configured from OTcl scripts */
        bind("epsilon_", &epsilon_);
        type_ = new char[3];
        strcpy(type_, "HB");            /* set the type of admission algorithm in use */

        /* member variables initializations */
        for(int s = 0; s < 3; s++)
        {
                sump2[s]= 0;
        }
        voClAvgLd = 0.0; viClAvgLd = 0.0; stdClAvgLd = 0.0;

        /* Initialize the bandwidth sharing and dynamic bandwidth borrowing mechanism */
        bw.bwAllocInit(bandwidth_);

}

/* The member function responsible for admitting flow into the network */
int MC_HB_ADC::admit_flow(int cl,double r,int b)
{
        double p=peak_rate(cl,r,b);             /* get the flow peak rate */

        /* when the flow of a traffic class should not be admitted, reject the flow immediately */
        if (backoff_) {
                if (rejected_)
                        return 0;
        }

        /* select the class of the traffic flow */
        switch(cl)
        {
        /* Admission point for voip flow traffic class. this class has the highest priority */
        case 1:
                // Admit flow according to class bandwidth share
                voClAvgLd = (p+est_[cl]->avload()+sqrt(log(1/epsilon_)*sump2[cl-1]/2));
                if (voClAvgLd <= bw.cl_bw[cl-1]) {
                        sump2[cl-1] += p*p;
                        est_[cl]->change_avload(p);             /* artificially adjust the voip class average load */
#ifdef BORROW
                        /* function call to dynamically borrow
```

```
                              bandwidth from standard class */
                         bw.voipBorrowBw(voClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                         /* print out the flow id, average load and
                              class bandwidth values */
                         printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                              voClAvgLd, bw.cl_bw[cl-1]);
#endif
                         return 1;
                    }
                    else {
                         rejected_=1;     /* indicate to reject
                              further connections of this flow type */
                         return 0;
                    }
                    break;

          case 2:
                    /* Admission point for video flow traffic class.
                         this class has middle priority */
                    viClAvgLd = (p+est_[cl]->avload()+sqrt(log(1/
                         epsilon_)*sump2[cl-1]/2));
                    if (viClAvgLd <= bw.cl_bw[cl-1]) {
                         sump2[cl-1] += p*p;
                         est_[cl]->change_avload(p);             /*
                              artificially adjust the video class
                              average load */
#ifdef BORROW
                         /* function call to dynamically borrow
                              bandwidth from standard class */
                         bw.videoBorrowBw(viClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                         /* print out the flow id, average load and
                              class bandwith values */
                         printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                              viClAvgLd, bw.cl_bw[cl-1]);
#endif
                         return 1;
                    }
                    else {
                         rejected_=1;     /* indicate to reject
                              further connections of this flow type */
                         return 0;
                    }
                    break;

          case 3:
                /* Admission point for standard flow traffic class.
                   this class has lowest priority */
                stdClAvgLd = (p+est_[cl]->avload()+sqrt(log(1/
                   epsilon_)*sump2[cl-1]/2));
                if (stdClAvgLd <= bw.cl_bw[cl-1]) {
                        sump2[cl-1] += p*p;
                        est_[cl]->change_avload(p);             /*
                           artificially adjust the standard class
                           average load */
#ifdef BORROW
                        /* function call to recover the dynamical
                           borrowed bandwidth from voip and video
                           classes */
                        bw.stdRecoverBw(voClAvgLd, viClAvgLd,
                           stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print out the flow id, average load and
                           class bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                           stdClAvgLd, bw.cl_bw[cl-1]);
#endif
                        return 1;
                }
                else {
                        rejected_=1;    /* indicate to reject
                           further connections of this flow type */
                        return 0;
                }
                break;

        default:
                printf("Wrong class flow id");
                return 0;
        }

}

/* The member function to reject admission request when the network
   is overloaded */
void MC_HB_ADC::rej_action(int cl,double r,int b)
{
        double p=peak_rate(cl,r,b);     /* get the flow peak rate */
        sump2[cl-1] -= p*p;                     /* decrease the sum
           of the square of the flow peak rate */

}

/* The member function to tear down a connection */
void MC_HB_ADC::teardown_action(int cl,double r,int b)
```

```
{
        rejected_=0;                              /* indicate
            to accept once more connections of this flow type */
        double p=peak_rate(cl,r,b);           /* get the flow peak
             rate */
        sump2[cl-1] -= p*p;                       /* decrease
            the sum of the square of the flow peak rate */

}

/* A static linkage class to make the MC_HB_ADC class objects
     accessible from Intepreted hierarchy */
static class MC_HB_ADCClass : public TclClass {
public:
        MC_HB_ADCClass() : TclClass("ADC/MC_HB") {}
        TclObject* create(int,const char*const*) {
                return (new MC_HB_ADC());
        }
}class_mc_hb_adc;
```

Listing 5: Acceptance region tangent at origin admission control source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/acto-adc.cc,v 1.7
            2005/08/26 05:05:27 tomh Exp $";
#endif


#include "adc.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "bandwidthAlloc.h"

/* The acceptance region tangent at origin class declaration */
class MC_ACTO_ADC : public ADC {

/* public member functions declarations */
public:
        MC_ACTO_ADC();
        void teardown_action(int,double,int);

/* protected member function and member variables declarations */
protected:
        int admit_flow(int,double,int);
        int rejected_;          /* a member variable to indicate
            that a flow is rejected admission */
        double s_;                          /* a member variable
            representing the space parameter of the chernoff bounds
            in the decision algorithm */

        /* Member variable for the dynamic bandwidth borrowing mechanism */
        double voClAvgLd, viClAvgLd, stdClAvgLd;
        BandwidthAlloc bw;
};

/* class constructror definition */
MC_ACTO_ADC::MC_ACTO_ADC() : rejected_(0)
{
        /* bind this member variable so that it can be configured from OTcl scripts */
        bind("s_", &s_);
        type_ = new char[5];
        strcpy(type_, "ACTO");          /* set the type of admission algorithm in use */

        /* member variables initializations */
        voClAvgLd = 0.0; viClAvgLd = 0.0; stdClAvgLd = 0.0;

        /* Initialize the bandwidth sharing and dynamic bandwidth borrowing mechanism */
        bw.bwAllocInit(bandwidth_);
}

/* The member function responsible for admitting flow into the network */
int MC_ACTO_ADC::admit_flow(int cl,double r,int b)
{
        double p=peak_rate(cl,r,b);             /* get the flow peak rate */

        /* when the flow of a traffic class should not be admitted, reject the flow immediately */
        if (backoff_) {
                if (rejected_)
                        return 0;
        }

        /* select the class of the traffic flow */
        switch(cl)
        {
        case 1:
                /* Admission point for voip flow traffic class. this class has the highest priority */
                voClAvgLd = exp(p*s_)*est_[cl]->avload();
                if (voClAvgLd <= bw.cl_bw[cl-1]) {

                        /* when the class average load should be artificially increased, do it */
                        if (dobump_) {
```

```
                    est_[cl]->change_avload(p);
                }
#ifdef BORROW
                /* function call to dynamically borrow
                    bandwidth from standard class */
                bw.voipBorrowBw(voClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                /* Print out the flow id, average load and
                    class bandwidth values */
                printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                    voClAvgLd, bw.cl_bw[cl-1]);
#endif
                return 1;
            }
            else {
                rejected_=1;    /* indicate to reject
                    further connections of this flow type */
                return 0;
            }
            break;

        case 2:
            /* Admission point for video flow traffic class.
                this class has middle priority */
            viClAvgLd = exp(p*s_)*est_[cl]->avload();
            if (viClAvgLd <= bw.cl_bw[cl-1]) {

                /* when the class average load should be
                    artificially increased, do it */
                if (dobump_) {
                    est_[cl]->change_avload(p);
                }
#ifdef BORROW
                /* function call to dynamically borrow
                    bandwidth from standard class */
                bw.videoBorrowBw(viClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                /* print out the flow id, average load and
                    class bandwidth values*/
                printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                    viClAvgLd, bw.cl_bw[cl-1]);
#endif
                return 1;

            }
            else {
                rejected_=1;    /* indicate to reject
```

```
                                further connections of this flow type */
                        return 0;
                }
                break;

        case 3:
                /* Admission point for standard flow traffic class.
                    this class has lowest priority */
                stdClAvgLd = exp(p*s_)*est_[cl]->avload();
                if (stdClAvgLd <= bw.cl_bw[cl-1]) {

                        /* when the class average load should be
                            artificially increased, do it */
                        if (dobump_) {
                                est_[cl]->change_avload(p);
                        }
#ifdef BORROW
                        /* function call to recover the dynamical
                            borrowed bandwidth from voip and video
                            classes */
                        bw.stdRecoverBw(voClAvgLd, viClAvgLd,
                            stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print our the flow id, average load and
                            class bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                            stdClAvgLd, bw.cl_bw[cl-1]);
#endif
                        return 1;
                }
                else {
                        rejected_=1;    /* indicate to reject
                            further connections of this flow type */
                        return 0;
                }
                break;

        default:
                printf("Invalid flow id\n");
                return 0;
        }
}

/* The member function to tear down a connection */
void MC_ACTO_ADC::teardown_action(int /*cl*/,double /*r*/,int /*b*/)
{
        rejected_=0;    /* indicate to accept once more connections
            of this flow type */
}
```

```

/* A static linkage class to make the MC_ACTO_ADC class objects
    accessible from Intepreted hierarchy */
static class MC_ACTO_ADCClass : public TclClass {
        public :
        MC_ACTO_ADCClass() : TclClass("ADC/MC_ACTO") {}
        TclObject* create(int,const char*const*) {
                return (new MC_ACTO_ADC());
        }
}class_mc_acto_adc;
```

Listing 6: Acceptance region tangent at peak admission control source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/actp-adc.cc,v 1.6
            2005/08/26 05:05:27 tomh Exp $";
#endif

#include "adc.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "bandwidthAlloc.h"

/* The acceptance region tangent at peak class declaration */
class MC_ACTP_ADC : public ADC {

/* public member functions declarations */
public:
        MC_ACTP_ADC();
        void teardown_action(int,double,int);
        void rej_action(int,double,int);

/* protected member function and member variables declarations */
protected:
        int admit_flow(int,double,int);
        int rejected_;          /* a member variable to indicate
            that a flow is rejected admission */
        double s_;                      /* a member variable
            representing the space parameter of the chernoff bounds
            in the decision algorithm */
        double sump[3];         /* array member variable to store
            the sum of the peak rates of the three different traffic
            flows */

        /* Member variable for the dynamic bandwidth borrowing
            mechanism */
        double voClAvgLd, viClAvgLd, stdClAvgLd;
        BandwidthAlloc bw;
};
```

```

/* class constructor definition */
MC_ACTP_ADC::MC_ACTP_ADC() : rejected_(0)
{
        /* bind this member variable so that it can be configured from OTcl scripts */
        bind("s_", &s_);
        type_ = new char[5];
        strcpy(type_, "ACTP");          /* set the type of admission algorithm in use */

        /* member variables initializations */
        for(int g = 0; g < 3; g++)
        {
                sump[g] = 0;
        }
        voClAvgLd = 0.0; viClAvgLd = 0.0; stdClAvgLd = 0.0;

        /* Initialize the bandwidth sharing and dynamic bandwidth borrowing mechanism */
        bw.bwAllocInit(bandwidth_);

}

/* The member function responsible for admitting flow into the network */
int MC_ACTP_ADC::admit_flow(int cl,double r,int b)
{
        double p=peak_rate(cl,r,b); /* get the flow peak rate */

        /* when the flow of a traffic class should not be admitted, reject the flow immediately */
        if (backoff_) {
                if (rejected_)
                        return 0;
        }

        /* select the class of the traffic flow */
        switch(cl)
        {
        case 1:
                /* Admission point for voip flow traffic class. this class has the highest priority */
                voClAvgLd = sump[cl-1]*(1-exp(-p*s_))+exp(-p*s_)*est_[cl]->avload();
                if (voClAvgLd <= bw.cl_bw[cl-1]) {
                        sump[cl-1] += p;          /* increase the sum of the flow's peak rate */

                        /* when the class average load should be artificially increased, do it */
                        if (dobump_) {
                                est_[cl]->change_avload(p);
                        }
#ifdef BORROW
                        /* function call to dynamically borrow
                            bandwidth from standard class */
                        bw.voipBorrowBw(voClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print the flow id, average load and class
                             bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                            voClAvgLd, bw.cl_bw[cl-1]);
#endif
                        return 1;
                }
                else {
                        rejected_=1;    /* indicate to reject
                            further connections of this flow type */
                        return 0;
                }
                break;

        case 2:
                /* Admission point for video flow traffic class.
                    this class has middle priority */
                viClAvgLd = sump[cl-1]*(1-exp(-p*s_))+exp(-p*s_)*
                    est_[cl]->avload();
                if (viClAvgLd <= bw.cl_bw[cl-1]) {
                        sump[cl-1] += p;        /* increase the sum
                            of the flow's peak rate */

                        /* when the class average load should be
                            artificially increased, do it */
                        if (dobump_) {
                                est_[cl]->change_avload(p);
                        }
#ifdef BORROW
                        /* function call to dynamically borrow
                            bandwidth from standard class */
                        bw.videoBorrowBw(viClAvgLd, stdClAvgLd);
#endif

#ifdef DBUG_MS
                        /* print the flow id, average load and class
                             bandwidth values */
                        printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                            viClAvgLd, bw.cl_bw[cl-1]);
#endif
                        return 1;
                    }
                    else {
                            rejected_=1;    /* indicate to reject
                               further connections of this flow type */
                            return 0;
                    }
                    break;

            case 3:
                    /* Admission point for standard flow traffic class.
                        this class has lowest priority */
                    stdClAvgLd = sump[cl-1]*(1-exp(-p*s_))+exp(-p*s_)*
                        est_[cl]->avload();
                    if (stdClAvgLd <= bw.cl_bw[cl-1]) {
                            sump[cl-1] += p;        /* increase the sum
                                of the flow's peak rate */

                            /* when the class average load should be
                                artificially increased, do it */
                            if (dobump_) {
                                    est_[cl]->change_avload(p);
                            }
#ifdef BORROW
                            /* function call to recover the dynamical
                                borrowed bandwidth from voip and video
                                classes */
                            bw.stdRecoverBw(voClAvgLd, viClAvgLd,
                                stdClAvgLd);
#endif

#ifdef DBUG_MS
                            /* print the flow id, average load and class
                                 bandwidth values */
                            printf("cl: %d, AVGL: %f, bw: %f\n", cl,
                                stdClAvgLd, bw.cl_bw[cl-1]);
#endif
                            return 1;
                    }
                    else {
                            rejected_=1;    /* indicate to reject
                               further connections of this flow type */
                            return 0;
                    }
                    break;
            default:
                    printf("Invalid flow id\n");
                    return 0;
            }
}

/* The member function to reject admission request when the network
```

```
    is overloaded */
void MC_ACTP_ADC::rej_action(int cl,double r,int b)
{
        double p=peak_rate(cl,r,b);             /* get the flow peak
             rate */
        sump[cl-1] -= p;                                /* decrease
            the sum of the flows peak rate */

}

/* The member function to tear down a connection */
void MC_ACTP_ADC::teardown_action(int cl,double r,int b)
{
        rejected_=0;                                    /* indicate
            to accept once more connections of this flow type */
        double p=peak_rate(cl,r,b);             /* get the flow peak
             rate */
        sump[cl-1] -= p;                                /* decrease
            the sum of the flow peak rate */

}

/* A static linkage class to make the MC_ACTP_ADC class objects
    accessible from Intepreted hierarchy */
static class MC_ACTP_ADCClass : public TclClass {
public:
        MC_ACTP_ADCClass() : TclClass("ADC/MC_ACTP") {}
        TclObject* create(int,const char*const*) {
                return (new MC_ACTP_ADC());
        }
}class_mc_actp_adc;
```

Listing 7: Estimator header file

```

#ifndef ns_estimator_h
#define ns_estimator_h

#include <stdio.h>
#include "connector.h"
#include "measuremod.h"
#include "timer-handler.h"

/* Declaration of the estimator class */
class Estimator;

/* Declaration of a timer handler for scheduling estimating events
    */
class Estimator_Timer : public TimerHandler {
public:
        Estimator_Timer(Estimator *est) : TimerHandler() {
                est_ = est;
        }

protected:
        virtual void expire(Event *e);
        Estimator *est_;
};

/* Declaration of estimator member functions and member variables */
class Estimator : public NsObject {

/* Declaration and definitions of public member funtions */
public:
        Estimator();
        inline double avload() { return double(avload_);}; /*
            member function to return the estimated network average
            load */
        inline virtual void change_avload(double incr) { avload_ +=
            incr;} /* member function to artificially change the
            network average load */
        inline virtual void newflow(double) {};
        int command(int argc, const char*const* argv);
        virtual void timeout(int);
        inline void recv(Packet *,Handler *){} /*member function for
             receiving packets (not used here) */
        virtual void start();
        void stop();
        void setmeasmod(MeasureMod *);
        void setactype(const char*);
        inline double &period(){ return period_;}     /* member
            function to return the sampling period in the estimation
            process */
        void trace(TracedVar* v);

/* Declaration of protected member function and member variables */
protected:
        MeasureMod *meas_mod_;                  /* a pointer to the
            measurement module object */
        TracedDouble avload_;                   /* a variable for
            storing actualized network average load */
        double period_;                                 /* the
            sampling period of the estimation process */
        virtual void estimate()=0;
        Estimator_Timer est_timer_;             /* timer variable
            for scheduling estimating events */
        TracedDouble measload_;                 /* a variable for
            storing current network average load */
        Tcl_Channel tchan_;                             /* variable
            for holding file descriptors for writting trace events */
        int src_;                                               /*
            source address of packets */
        int dst_;                                               /*
```

```
                destination address of packets */
        double omeasload_;                              /* helper variable for writting measured network utilization in trace file */
        double oavload_;                                /* helper variable for writting estimated network utilization in trace file */
        char *actype_;                                  /* a pointer variable to the type of estimator in use */
};

#endif
```

Listing 8: Estimator base class source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/estimator.cc,v 1.8 2005/08/26 05:05:27 tomh Exp $";
#endif

#include "estimator.h"

/* The estimator class constructor for initialing and binding member variables */
Estimator::Estimator() : meas_mod_(0),avload_(0.0),est_timer_(this), measload_(0.0), tchan_(0), omeasload_(0), oavload_(0)
{
        /* bind these member variables to make them accessible for OTcl scripts */
        bind("period_",&period_);
        bind("src_", &src_);
        bind("dst_", &dst_);

        avload_.tracer(this);
        avload_.name("\"Estimated Util.\"");
        measload_.tracer(this);
        measload_.name("\"Measured Util.\"");
}
/* The command function to execute C++ member functions invoked in OTcl */
int Estimator::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance(); /* get the Tcl instance for transfering results back and forward b/w C++ and OTcl */
        if (argc==2) {
                if (strcmp(argv[1],"load-est") == 0) {
                        /* transfer the estimated network average load return in C++ to OTcl */
                        tcl.resultf("%.3f",avload());
                        return(TCL_OK);
                } else if (strcmp(argv[1],"link-utlzn") == 0) {
                        /* transfer the link utilization measured in
                            C++ to OTcl */
                        tcl.resultf("%.3f",meas_mod_->bitcnt()/
                            period_);
                        return(TCL_OK);
                }
        }
        if (argc == 3) {
                if (strcmp(argv[1], "attach") == 0) {
                        /* attach a file descriptor for writting
                            trace events */
                        int mode;
                        const char* id = argv[2];
                        tchan_ = Tcl_GetChannel(tcl.interp(), (char
                            *)id, &mode);
                        if (tchan_ == 0) {
                                tcl.resultf("Estimator: trace: can't
                                    attach %s for writing", id);
                                return (TCL_ERROR);
                        }
                        return (TCL_OK);
                }
                if (strcmp(argv[1], "setbuf") == 0) {
                        /* some sub classes actually do something
                            here */
                        return(TCL_OK);
                }
        }
        return NsObject::command(argc,argv);
}
/* a member function to set an instantiated measurement object from
    OTcl*/
void Estimator::setmeasmod (MeasureMod *measmod)
{
        meas_mod_=measmod;
}

/* a member function to start the estimation process */
void Estimator::start()
{
        avload_=0;
        measload_ = 0;
        est_timer_.resched(period_);
}

/* a member function to stop the estimation process */
void Estimator::stop()
{
        est_timer_.cancel();
}
```

```

/* a member function for scheduling the estimation process */
void Estimator::timeout(int)
{
        estimate();
        est_timer_.resched(period_);
}

/* a function for checking the end of estimation process and
    rescheduling it */
void Estimator_Timer::expire(Event* /*e*/)
{
        est_->timeout(0);
}

/* a member function to write estimation trace events to a trace
    file */
void Estimator::trace(TracedVar* v)
{
        char wrk[500];
        double *p, newval;

        /* check for right variable */
        if (strcmp(v->name(), "\"Estimated Util.\"") == 0) {
                p = &oavload_;
        }
        else if (strcmp(v->name(), "\"Measured Util.\"") == 0) {
                p = &omeasload_;
        }
        else {
                fprintf(stderr, "Estimator: unknown trace var %s\n",
                    v->name());
                return;
        }

        newval = double(*((TracedDouble*)v));

        if (tchan_) {
                int n;
                double t = Scheduler::instance().clock();
                /* f -t 0.0 -s 1 -a SA -T v -n Num -v 0 -o 0 */
                sprintf(wrk, "f -t %g -s %d -a %s:%d-%d -T v -n %s -
                    v %g -o %g",
                        t, src_, actype_, src_, dst_, v->name(),
                            newval, *p);
                n = strlen(wrk);
                wrk[n] = '\n';
                wrk[n+1] = 0;
                (void)Tcl_Write(tchan_, wrk, n+1);
        }

        *p = newval;
        return;
}

/* a member function for setting the type of estimator in use */
void Estimator::setactype(const char* type)
{
        actype_ = new char[strlen(type)+1];
        strcpy(actype_, type);
        return;
}
```

Listing 9: Time window estimator source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/timewindow-est.cc,v
            1.7 2005/08/26 05:05:28 tomh Exp $";
#endif


#include "estimator.h"
#include <stdlib.h>

/* The time window estimator class declaration */
class TimeWindow_Est : public Estimator {
public:
        /* the class consructor definition for initializing and
            binding member variables */
        TimeWindow_Est() :scnt(1),maxp(0)
        {
                /* bind this member variable so that it will be
                    accessible from OTcl scripts */
                bind("T_",&T_);
        };

        /* a member function to artificially change the network
            average load */
        inline void change_avload(double incr)
        { avload_ += incr;
                if (incr >0) scnt=0; /* restart the time window
                    frame count */
        }
protected:
        void estimate();
        int scnt;       /* a variable to count the sampling periods
            to indicate the end of a time window frame */
        double maxp;    /* the maximum network average load measured
             in the previous time window frame */
        int T_;                 /* variable that hold the value for
            the time window frame */
};

/* The time window estimation function */
void TimeWindow_Est::estimate() {
        measload_ = meas_mod_->bitcnt()/period_;          /* measure the current network average load */
        if (meas_mod_->bitcnt()/period_ >avload_)        /* test if the current network average load is greater than the previous */
                avload_=meas_mod_->bitcnt()/period_;     /* actualize the network average load with the current value */
        if (maxp < meas_mod_->bitcnt()/period_)          /* check if the maximum value in the previous time window frame is less than the current value */
                maxp=meas_mod_->bitcnt()/period_;                /* make the current value the maximum value */

        /* check if the end of the time window frame has been reach */
        if (scnt == T_)
                {
                        scnt-=T_;                /* reduce the sampling count by the time window frame value */
                        avload_=maxp;    /* actualize the average load with the maximum average load measured in previous time window frame */
                        maxp=0;                  /* reset the previous maximum value */
                }

        meas_mod_->resetbitcnt(); /* reset the measured value at the end of each sampling */
        scnt++;         /* increase the sampling count */
}

/* A static linkage class to make the TimeWindow_Est class objects accessible from Intepreted hierarchy */
static class TimeWindow_EstClass : public TclClass {
public:
        TimeWindow_EstClass() : TclClass ("Est/TimeWindow") {}
        TclObject* create(int,const char*const*) {
                return (new TimeWindow_Est());
        }
}class_timewindow_est;
```

Listing 10: Exponential averaging estimator source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/expavg-est.cc,v 1.4
            2005/08/26 05:05:27 tomh Exp $";
#endif


#include <math.h>
#include "estimator.h"

/* The exponential averaging estimator class declaration */
class ExpAvg_Est : public Estimator {
public:
        /* the class contructor definition */
        ExpAvg_Est()
        {
                /* bind this member variable to make it accessible
                    from OTcl scripts */
                bind("w_",&w_);
        };
protected:
        void estimate();
        double w_;              /* a member variable to hold the
            value of the averaging weight in the impulse response
            function */
};

/* The exponential averaging estimation function */
void ExpAvg_Est::estimate()
{
        /* estimating the average network load with the impulse
            response function */
        avload_=(1-w_)*avload_+w_*meas_mod_->bitcnt()/period_;

        meas_mod_->resetbitcnt(); /* reset the measured value at the
             end of each sampling */
}

/* A static linkage class to make the ExpAvg_Est class objects
    accessible from Intepreted hierarchy */
static class ExpAvg_EstClass : public TclClass {
public:
        ExpAvg_EstClass() : TclClass ("Est/ExpAvg") {}
        TclObject* create(int,const char*const*) {
                return (new ExpAvg_Est());
        }
}class_expavg_est;
```

Listing 11: Point sample estimator source file

```

#ifndef lint
static const char rcsid[] =
        "@(#) $Header: /cvsroot/nsnam/ns-2/adc/pointsample-est.cc,v
```

```
            1.4 2005/08/26 05:05:28 tomh Exp $";
#endif

#include "estimator.h"
#include <stdlib.h>
#include <math.h>

/* The point sample estimator class declaration */
class PointSample_Est : public Estimator {
public:
        /* the class constructor definition */
        PointSample_Est() {};
protected:
        void estimate();
};

/* The point sample estimation function */
void PointSample_Est::estimate()
{
        /* measure the current average network load */
        avload_=meas_mod_->bitcnt()/period_;

        meas_mod_->resetbitcnt();  /* reset the measured value at
            the end of each sampling period */
}

/* A static linkage class to make the PointSample_Est class objects
    accessible from Intepreted hierarchy */
static class PointSample_EstClass : public TclClass {
public:
        PointSample_EstClass() : TclClass ("Est/PointSample") {}
        TclObject* create(int,const char*const*) {
                return (new PointSample_Est());
        }
}class_pointsample_est;
```

Listing 12: Multi-queue scheduler source file

```

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsroot/nsnam/ns-2/adc/simple-intserv-sched.cc,v
        1.7 2005/08/26 05:05:28 tomh Exp $ (LBL)";
#endif


#include <stdio.h>
#include "config.h"
#include "queue.h"

// define the number of queue classes to be created
#define CLASSES 3
```

```

/* A queue management class used to create packet queues and schedule packets */
class MultiClassServ : public Queue {
public:
        /* Class constructor for creating packet queues and intializing member variables */
        MultiClassServ() {
                int i;
                char buf[10];
                for (i=0;i<CLASSES;i++) {
                        q_[i] = new PacketQueue;    //create the queues and store the pointers in array
                        qlimit_[i] = 0;             //initialize the queue lengths
                        sprintf(buf,"qlimit%d_",i);
                        bind(buf,&qlimit_[i]);      // bind the queue length so that it will be configurable from OTcl scripts.
                }
        }
        protected :
        void enque(Packet *);   //member function to add packets to queues
        Packet *deque();        //memeber variable to remove packets from queues
        PacketQueue *q_[CLASSES]; //an array to hold pointers to the packet queues address
        int qlimit_[CLASSES];     //an integer array to specify the sizes of the packet queues.
};

/* A static linkage class to make the MultiClassServ class available in the interpreted class hierarchy */
static class MultiServClass : public TclClass {
public:
        MultiServClass() : TclClass("Queue/MultiClassServ") {}
        TclObject* create(int, const char*const*) {
                return (new MultiClassServ);
        }
} multi_class_serv;

/*The enqueue member function definition. It has one function parameter: the packet to be
 * added to a queue
 */
void MultiClassServ::enque(Packet* p)
{
        hdr_ip* iph=hdr_ip::access(p);     //access the ip header variables
        int cl= iph->flowid();    //get the packets flow id

        /* check if the flow id is among the defined class flow ids */
        if(cl >= 1 && cl <= 4)
        {
                /*Check if the queue size is greater than the specified queue size */
                if (q_[cl - 1]->length() >= (qlimit_[cl -1]-1)) {
                        hdr_cmn* ch=hdr_cmn::access(p);     // access the common header variables
                        packet_t ptype = ch->ptype();       // check the type of the present packet
                        /*do not allow the signalling protocol packets to be dropped */
                        if ( (ptype != PT_REQUEST) && (ptype != PT_REJECT) && (ptype != PT_ACCEPT) && ( ptype != PT_CONFIRM) && (ptype != PT_TEARDOWN) ) {
                                drop(p);
                        }
                        else {
                                q_[cl - 1]->enque(p); //add packets to queues even if they are oversized.
                        }
                }
                else {
                        q_[cl - 1]->enque(p);  //add packets to under sized queues
                }
        }
        else
        {
                printf("Invalid flow id\n");
        }

}

/* The deque member function definition to remove packets from queues*/
Packet *MultiClassServ::deque()
{
        int i;
        /* select the queue from where to remove packet*/
        for (i=CLASSES-1;i>=0;i--)
                if (q_[i]->length())   // check if there is any packet there at all
                        return q_[i]->deque();  // remove the packet from the queue
        return 0;
}
```

Listing 13: Signalling mechanism header file

```

#ifndef MULTISALINK_H_
#define MULTISALINK_H_

//no of pending flows decisions
#define NFLOWS 5000

/* Structure to save the number of reserved flows */
struct pending {
        int flowid;
        int status;
};
/* A class declaration for realising the signal protocol mechanism
    */
class MultiSALink : public Connector {
public:
        MultiSALink();          /*class constructor */
        int command(int argc, const char*const* argv);  /* member
            function to execute OTcl commands */
        void trace(TracedVar* v);  /* member function for tracing
            flows */

protected:
        void recv(Packet *,Handler *);  /* member function to
            receive flow request packets */
        ADC *adc_;      /* variable for calling the ADC decision
            algorithms */
        //int RTT;
        pending pending_[NFLOWS];    /* struct to hold the state of
            requested flows */
        int lookup(int);     /* member function to check if a flow is
             already accepted */
        int get_nxt();      /* member function to get the next empty
            state for a flow requesting admission */
        TracedInt numfl_;     /* store the number of flows accepted
            */
        Tcl_Channel tchan_;    /* pointer to the trace file */
        int onumfl_;  /* XXX: store previous value of numfl_ for nam
             */
        int src_;    /* id of node we're connected to (for nam traces
            ) */
        int dst_;    /* id of node at end of the link */
        int last_;       /* previous ac decision on this link */
};


#endif /*MULTISALINK_H_*/
```

Listing 14: Signalling mechanism source file

```

```

```
#include <stdio.h>
#include "packet.h"
#include "ip.h"
#include "resv.h"
#include "connector.h"
#include "adc.h"
#include "multisalink.h"

/* A static linkage class to make the MultiSAlink class objects
    accessible from Intepreted hierarchy */
static class MultiSALinkClass : public TclClass {
public:
        MultiSALinkClass() : TclClass("MultiSALink") {}
        TclObject* create(int, const char*const*) {
                return (new MultiSALink());
        }
}class_multisalink;

/* Class Constructor for initialising Member variable and
    reservation states */
MultiSALink::MultiSALink() : adc_(0), numfl_(-1), tchan_(0), onumfl_
    (0), last_(-1)
{
        int i;
        /* Initialize the reservation states */
        for (i=0;i<NFLOWS;i++) {
                pending_[i].flowid=-1;
                pending_[i].status=0;
        }
        /*Make the src_ and dst_ member variable to be accessible
            from the intepreted hierarchy */
        bind("src_", &src_);
        bind("dst_", &dst_);

        numfl_.tracer(this);
        numfl_.name("\"Admitted Flows\"");
}

/* The receive member function receives the reservation request and
    process it accordingly
 * Parameter P: is the request packet, Parameter h: is a handler
    instance for the scheduler */
void MultiSALink::recv(Packet *p, Handler *h)
{
        int decide;  /* to hold the admission algorithms decision */
        int j;  /* Running variable for the reservation state */

        hdr_cmn *ch=hdr_cmn::access(p);    /* get access to the
            common header variables */
        hdr_ip *iph=hdr_ip::access(p);    /* get access to the IP
            header variables */
        hdr_resv *rv=hdr_resv::access(p); /* get access to the reservation header variables */

        /* Store the flow id of the current packet */
        int cl=iph->flowid();

        /* Check the reservation message type */
        switch(ch->ptype()) {
        case PT_REQUEST:  /* the case of flow request */
                decide=adc_->admit_flow(cl,rv->rate(),rv->bucket()); /* the admission decision for a flow */
                if (tchan_) /* when tracing is desired, log the reservation process in the trace file */
                        if (last_ != decide) {
                                int n;
                                char wrk[50];
                                double t = Scheduler::instance().clock();
                                sprintf(wrk, "l -t %g -s %d -d %d -S COLOR -c %s",
                                        t, src_, dst_, decide ? "MediumBlue" : "red" );
                                n = strlen(wrk);
                                wrk[n] = '\n';
                                wrk[n+1] = 0;
                                (void)Tcl_Write(tchan_, wrk, n+1);
                                last_ = decide;
                        }
                /* include the admission decision in the reservation packet header */
                rv->decision() &= decide;
                if (decide) { /* when a flow is admitted, add its state to the reservation states */
                        j=get_nxt();
                        pending_[j].flowid=iph->flowid();
                        //pending_[j].status=decide;
                        numfl_++;    /* Increase the number of admitted flows */

#ifdef DBUG_MS
                        /* show flow ids admitted */
                        printf("Flow id %d admitted\n", iph->flowid());
#endif
                }
                break;
        case PT_ACCEPT:
        case PT_REJECT:
                break;

        case PT_CONFIRM:    /* the case of request confirmations */
```

```
                {
                        j=lookup(iph->flowid()); /* get the state of the flow from the reservation states */
                        if (j!=-1) {
                                if (!rv->decision()) {
                                        /* when the flow is rejected decrease the avload for its class */
                                        adc_->rej_action(cl,rv->rate(),rv->bucket());
                                        numfl_--;  /* decrement the number of flows admitted */

#ifdef DBUG_MS
                                        /* show flow ids rejected */
                                        printf("Flow id %d rejected\n", iph->flowid());
#endif
                                }
                                pending_[j].flowid=-1;    /* release its state from the reservation states */
                        }
                        break;
                }
        case PT_TEARDOWN: /* the case of breaking down a connection */
                {
                        /* when the connection is disconnected, decrease the avload of the flow's class */
                        adc_->teardown_action(cl,rv->rate(),rv->bucket());
                        numfl_--;  /* decrement the number of flows admitted */
                        break;
                }
        default:
#ifdef notdef
                error("unknown signalling message type : %d",ch->ptype());
                abort();
#endif
                break;
        }
        send(p,h); /* send the packet to the next hub */
}

/* the command member function for invoking C++ functions from OTcl
 * the two parameters represents the argument vectors of the invoked
```

```
    function */
int MultiSALink::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();  /* the Tcl instance for communicating results in C++ back to OTcl space */
        char wrk[5000];   /* buffer for holding trace data */

        if (argc ==3) {
                if (strcmp(argv[1],"attach-adc") == 0 ) {
                        adc_=(ADC *)TclObject::lookup(argv[2]);
                        if (adc_ ==0 ) {
                                tcl.resultf("no such node %s", argv[2]);
                                return(TCL_ERROR);
                        }
                        return(TCL_OK);
                }
                if (strcmp(argv[1], "attach") == 0) {
                        int mode;
                        const char* id = argv[2];
                        tchan_ = Tcl_GetChannel(tcl.interp(), (char *)id, &mode);
                        if (tchan_ == 0) {
                                tcl.resultf("MultiSALink: trace: can't attach %s for writing", id);
                                return (TCL_ERROR);
                        }
                        return (TCL_OK);
                }
        }
        if (argc == 2) {
                if (strcmp(argv[1], "add-trace") == 0) {
                        if (tchan_) {
                                sprintf(wrk, "a -t * -n %s:%d-%d -s %d",
                                        adc_->type(), src_, dst_, src_);
                                int n = strlen(wrk);
                                wrk[n] = '\n';
                                wrk[n+1] = 0;
                                (void)Tcl_Write(tchan_, wrk, n+1);
                                numfl_ = 0;
                        }
                        return (TCL_OK);
                }
        }
        return Connector::command(argc,argv);
}

/* member function for checking the state of a flow in the resource reservation state table
```

```
 * the parameter is the flow id. It returns -1 when the state is not
     in the reservation table */
int MultiSALink::lookup(int flowid)
{
        int i;
        for (i=0;i<NFLOWS;i++)
                if (pending_[i].flowid==flowid)
                        return i;
        return(-1);
}

/* member function to get the next available free state in the
    resource reservation state table
 * it returns a state when available or a warning message if there
     is no state available */
int MultiSALink::get_nxt()
{
        int i;
        for (i=0;i<NFLOWS;i++)
                {
                        if (pending_[i].flowid==-1)
                                return i;
                }
        printf("Ran out of pending space \n");
        exit(1);
        //return i;
}

/* member function for writing the trace data.
 * Parameter: the traced data to be written */
void MultiSALink::trace(TracedVar* v)
{
        char wrk[5000];
        int *p, newval;

        if (strcmp(v->name(), "\"Admitted Flows\"") == 0) {
                p = &onumfl_;
        }
        else {
                fprintf(stderr, "MultiSALink: unknown trace var %s\n
                    ", v->name());
                return;
        }

        newval = int(*((TracedInt*)v));

        if (tchan_) {
                int n;
                double t = Scheduler::instance().clock();
                /* f -t 0.0 -s 1 -a SA -T v -n Num -v 0 -o 0 */
                sprintf(wrk, "f -t %g -s %d -a %s:%d-%d -T v -n %s -
                            v %d -o %d",
                                t, src_, adc_->type(), src_, dst_, v->name()
                                    , newval, *p);
                        n = strlen(wrk);
                        wrk[n] = '\n';
                        wrk[n+1] = 0;
                        (void)Tcl_Write(tchan_, wrk, n+1);

        }

        *p = newval;

        return;
}
```

Listing 15: Bandwidth allocation mechanism header file

```

#ifndef BANDWIDTHALLOC_H_
#define BANDWIDTHALLOC_H_


/* A structure to hold data for the bandwidth allocation mechanism
    */
struct BandwidthAlloc {
        double netBandwidth;    /* store the total bandwidth capacity
            */
        double cl_bw[3];                /* array variable to hold
            the class bandwidth for each of the three classes of
            service */

        /* Group of member variables for realizing the dynamic
            bandwidth allocation */
        double voClThold, viClThold, stdClMidThold, stdClNorBwBor,
            stdClMaxThold;
        double stdClMinAvgLd, stdClMinThold, stdClMaxBwBor;
        bool voBorFrStd, viBorFrStd, voBorMaxFrStd, viBorMaxFrStd;
        bool stdRcvMax1BorFrVo, stdRcvMax1BorFrVi, stdRcv1BorFrVo,
            stdRcv1BorFrVi, stdRcvMax2BorFrVo;
        bool stdRcvMax2BorFrVi, stdRcv2BorFrVo, stdRcv2BorFrVi,
            stdNRcvBor;



        /* Function declarations */
        void bwAllocInit(double& band);
        void voipBorrowBw(double& voClAvgLd, double& stdClAvgLd);
        void videoBorrowBw(double& viClAvgLd, double& stdClAvgLd);
        void stdRecoverBw(double& voClAvgLd, double& viClAvgLd,
            double& stdClAvgLd);

};
```

```
#endif /*BANDWIDTHALLOC_H_*/
```

Listing 16: Bandwidth allocation mechanism source file

```

#include "bandwidthAlloc.h"


/* Initialization function */
void BandwidthAlloc::bwAllocInit(double& band)
{
        netBandwidth = band;  /* the total bandwidth capacity */

        /* member variable initialization */
        voBorFrStd = true; viBorFrStd = true;
        voBorMaxFrStd = true; viBorMaxFrStd = true;

        /* static sharing of the net bandwidth among the traffic
           flow classes */
        cl_bw[0] = 0.5 * netBandwidth;     /* Voip class bandwidth
           share */
        cl_bw[1] = 0.35 * netBandwidth; /* video class bandwith
           share */
        cl_bw[2] = 0.15 * netBandwidth; /* standard class bandwidth
           share */

        voClThold = cl_bw[0] * 0.85;            /* voip class
           threshold for borrowing bandwidth */
        viClThold = cl_bw[1] * 0.9;             /* video class
           threshold for borrowing bandwidth */
        stdClMinThold =  cl_bw[2] * 0.1;        /* standard class
           minimum threshold for collecting back half of the maximum
            borrowed bandwidth */
        stdClMidThold = cl_bw[2] * 0.47;        /* standard class
           middle threshold for collecting back half of the normal
           borrowed bandwidth */
        stdClNorBwBor = cl_bw[2] * 0.5;         /* standard class
           threshold under which normal bandwidth borrow should take
            place */
        stdClMinAvgLd = cl_bw[2] * 0.15;        /* standard class
           minimum average load under which maximum bandwidth borrow
            should take place */
        stdClMaxThold = cl_bw[2] * 0.7;         /* standard class
           maximum threshold for collecting back the rest of the
           borrowed bandwidth */
        stdClMaxBwBor = cl_bw[2] * 0.85;        /* standard class
           maximum bandwidth to be borrowed */
}

/* function that is used by the voip class to borrow bandwidth from
   standard class */
void BandwidthAlloc::voipBorrowBw(double& voClAvgLd, double&
    stdClAvgLd)
{

        /* When the voip class average load is more than 85% of its
            class bandwidth and the standard class
           average load is less than 15% or 50% of its class
               bandwidth, borrow some bandwith from the standard
               class */
        if (voClAvgLd > voClThold && stdClAvgLd < stdClNorBwBor)
        {
                /* when the standard class average load is less than
                    15% of its class bandwidth, borrow maximum
                   bandwidth */
                if(stdClAvgLd < stdClMinAvgLd && voBorMaxFrStd)
                {
                        cl_bw[2] -= (stdClMaxBwBor * 0.6);        /*
                            borrow 60% of the maximum borrowable
                            bandwidth from standard class*/
                        cl_bw[0] += (stdClMaxBwBor * 0.6);        /*
                            add the borrowed bandwidth to the voip
                            class bandwidth */
                        voBorMaxFrStd = false;
                                              /* check that this borrow
                             is done once until the bandwidth is
                            released back */
                        stdRcvMax1BorFrVo = true;
                                   /* check that the first step of
                            recovering half of the maximum borrowed
                            bandwidth is done once until the
                            bandwidth is borrowed again by voip class
                             */
                        stdRcvMax2BorFrVo = true;
                                   /* check that the second step of
                            recovering the rest of the maximum
                            borrowed bandwidth is done once until the
                             bandwidth is borrowed again by voip
                            class */
                        voBorFrStd = false;
                                              /* when maximum bandwidth
                             is borrowed, do not allow normal
                            bandwidth borrow */
                }

                /* when the standard class average load is less than
                    50% of its class bandwidth, borrow normal
                   bandwidth */
                if(stdClAvgLd > stdClMinAvgLd && stdClAvgLd <
                    stdClNorBwBor && voBorFrStd)
                {
                        cl_bw[2] -= (stdClNorBwBor * 0.6);        /*
                            borrow 60% of the normal borrowable
```

```
                                bandwidth from standard class*/
                        cl_bw[0] += (stdClNorBwBor * 0.6);        /*
                                add the borrowed bandwidth to the voip
                                class bandwidth */
                        voBorFrStd = false;
                                        /* check that this borrow
                                 is done once until the bandwidth is
                                released back */
                        stdRcv1BorFrVo = true;
                                /* check that the first step of
                                recovering half of the normal borrowed
                                bandwidth is done once until the
                                bandwidth is borrowed again by voip class
                                 */
                        stdRcv2BorFrVo = true;
                                /* check that the second step of
                                recovering the rest of the normal
                                borrowed bandwidth is done once until the
                                 bandwidth is borrowed again by voip
                                class */
                        voBorMaxFrStd = false;
                                /* when normal bandwidth is
                                borrowed, do not allow maximum bandwidth
                                borrow */
                }
        }

}

/* function that is used by the video class to borrow bandwidth from
     standard class */
void BandwidthAlloc::videoBorrowBw(double& viClAvgLd, double&
    stdClAvgLd)
{
        /* When the video class average load is more than 90% of its
             class bandwidth and the standard class
        * average load is less than 15% or 50% of its class
            bandwidth, borrow some bandwith from the standard class
            */
        if (viClAvgLd > viClThold && stdClAvgLd < stdClNorBwBor)
        {
                /* when the standard class average load is less than
                     15% of its class bandwidth, borrow maximum
                    bandwidth */
                if(stdClAvgLd < stdClMinAvgLd && viBorMaxFrStd)
                {
                        cl_bw[2] -= (stdClMaxBwBor * 0.4);        /*
                                borrow 40% of the maximum borrowable
                                bandwidth from standard class*/
                        cl_bw[1] += (stdClMaxBwBor * 0.4);        /*
                                add the borrowed bandwidth to the video
```

```
                                    class bandwidth */
                                viBorMaxFrStd = false;
                                                    /* check that this borrow
                                     is done once until the bandwidth is
                                    released back */
                                stdRcvMax1BorFrVi = true;
                                             /* check that the first step of
                                    recovering half of the maximum borrowed
                                    bandwidth is done once until the
                                    bandwidth is borrowed again by video
                                    class */
                                stdRcvMax2BorFrVi = true;
                                             /* check that the second step of
                                    recovering the rest of the maximum
                                    borrowed bandwidth is done once until the
                                     bandwidth is borrowed again by video
                                    class */
                                viBorFrStd = false;
                                                    /* when maximum bandwidth
                                     is borrowed, do not allow normal
                                    bandwidth borrow */
                        }

                        /* when the standard class average load is less than
                             50% of its class bandwidth, borrow normal
                           bandwidth */
                        if(stdClAvgLd > stdClMinAvgLd && viBorFrStd)
                        {
                                cl_bw[2] -= (stdClNorBwBor * 0.4);        /*
                                    borrow 40% of the normal borrowable
                                    bandwidth from standard class*/
                                cl_bw[1] += (stdClNorBwBor * 0.4);        /*
                                    add the borrowed bandwidth to the video
                                    class bandwidth */
                                viBorFrStd = false;
                                                    /* check that this borrow
                                     is done once until the bandwidth is
                                    released back */
                                stdRcv1BorFrVi = true;
                                             /* check that the first step of
                                    recovering half of the normal borrowed
                                    bandwidth is done once until the
                                    bandwidth is borrowed again by video
                                    class */
                                stdRcv2BorFrVi = true;
                                             /* check that the second step of
                                    recovering the resf of the normal
                                    borrowed bandwidth is done once until the
                                     bandwidth is borrowed again by video
                                    class */
                                viBorMaxFrStd = false;
```

```
                                        /* when normal bandwidth is borrowed, do not allow maximum bandwidth borrow */
                        }
                }

}

/* function that is used by the standard class to recover borrowed bandwidth from voip and video classes */
void BandwidthAlloc::stdRecoverBw(double& voClAvgLd, double& viClAvgLd, double& stdClAvgLd)
{
        /* when the standard class average load is more than 10% and less than 47% of its class bandwidth,
         * collect back half of the maximum borrowed bandwidth from voip and video classes */
        if (stdClAvgLd > stdClMinThold && stdClAvgLd < stdClMidThold)
        {
                /* when voip class borrowed maximum bandwidth from standard class bandwidth and it is not using it at the moment
                 * and standard class has not recovered this bandwidth for the first time, then give half of the borrowed bandwidth back */
                if(cl_bw[0] > ((netBandwidth * 0.5) + (stdClNorBwBor * 0.3)) && voClAvgLd < (netBandwidth * 0.5) && stdRcvMax1BorFrVo)
                {
                        cl_bw[0] -= (stdClMaxBwBor * 0.3);      /* remove the borrowed bandwidth from voip class bandwidth */
                        cl_bw[2] += (stdClMaxBwBor * 0.3);      /* add back the bandwidth to the standard class bandwidth */
                        stdRcvMax1BorFrVo = false;
                                        /* do not allow standard class to request for this bandwidth again until voip class borrows it again */
                }

                /* when video class borrowed maximum bandwidth from standard class bandwidth and it is not using it at the moment
                 * and standard class has not recovered this bandwidth for the first time, then give half of the borrowed bandwidth back */
                if(cl_bw[1] > ((netBandwidth * 0.35) + (stdClNorBwBor * 0.2)) && viClAvgLd < (netBandwidth * 0.35) && stdRcvMax1BorFrVi)
                {
                        cl_bw[1] -= (stdClMaxBwBor * 0.2);       /*
                            remove the borrowed bandwidth from video
                            class bandwidth */
                        cl_bw[2] += (stdClMaxBwBor * 0.2);       /*
                            add back the bandwidth to the standard
                            class bandwidth */
                        stdRcvMax1BorFrVi = false;
                                        /* do not allow standard class to
                             request for this bandwidth again until
                            video class borrows it again */
                }


        }
        /* when the standard class average load is more than 47% of
            its class bandwidth,
         * collect back the rest of the borrowed bandwidth from voip
              and video classes */
        else if (stdClAvgLd > stdClMidThold)
        {
                /* when voip class borrowed normal bandwidth from
                    standard class bandwidth and it is not using it
                    at the moment
                 * and standard class has not recovered this
                     bandwidth for the first time, then give half of
                     the borrowed bandwidth back */
                if(cl_bw[0] > ((netBandwidth * 0.5) + (stdClNorBwBor
                     * 0.3)) && voClAvgLd < (netBandwidth * 0.5) &&
                    stdRcv1BorFrVo)
                {
                        cl_bw[0] -= (stdClNorBwBor * 0.3);       /*
                            remove the borrowed bandwidth from voip
                            class bandwidth */
                        cl_bw[2] += (stdClNorBwBor * 0.3);       /*
                            add back the bandwidth to the standard
                            class bandwidth */
                        stdRcv1BorFrVo = false;
                                        /* do not allow standard class to
                             request for this bandwidth again until
                            voip class borrows it again */
                }

                /* when video class borrowed normal bandwidth from
                    standard class bandwidth and it is not using it
                    at the moment
                 * and standard class has not recovered this
                     bandwidth for the first time, then give half of
                     the borrowed bandwidth back */
                if(cl_bw[1] > ((netBandwidth * 0.35) + (
                    stdClNorBwBor * 0.3)) && voClAvgLd < (
                netBandwidth * 0.35) && stdRcv1BorFrVi)
            {
                    cl_bw[1] -= (stdClNorBwBor * 0.2);      /*
                        remove the borrowed bandwidth from video
                        class bandwidth */
                    cl_bw[2] += (stdClNorBwBor * 0.2);      /*
                        add back the bandwidth to the standard
                        class bandwidth */
                    stdRcv1BorFrVi = false;
                                    /* do not allow standard class to
                         request for this bandwidth again until
                        video class borrows it again */
            }
        }

        /* when the standard class average load is more than 70% of
            its class bandwidth,
         * collect back the rest of the borrowed bandwidth from voip
              and video classes */
        if (stdClAvgLd > stdClMaxThold)
        {
            /* when voip class borrowed maximum bandwidth from
                standard class bandwidth and it is not using it
                at the moment
             * and standard class has not recovered this
                 bandwidth for the first time, then give the rest
                  of the borrowed bandwidth back */
            if(cl_bw[0] > (netBandwidth * 0.5) && voClAvgLd < (
                netBandwidth * 0.5) && stdRcvMax2BorFrVo)
            {
                    cl_bw[0] -= (stdClMaxBwBor * 0.3);      /*
                        remove the borrowed bandwidth from voip
                        class bandwidth */
                    cl_bw[2] += (stdClMaxBwBor * 0.3);      /*
                        add back the bandwidth to the standard
                        class bandwidth */
                    stdRcvMax2BorFrVo = false;
                                    /* do not allow standard class to
                         request for this bandwidth again until
                        voip class borrows it again */
                    voBorMaxFrStd = true;
                                    /* allow voip class to borrow
                        maximum bandwidth again from standard
                        class */
            }

            /* when video class borrowed maximum bandwidth from
                standard class bandwidth and it is not using it
                at the moment
             * and standard class has not recovered this
                 bandwidth for the first time, then give the rest
                of the borrowed bandwidth back */
            if(cl_bw[1] > (netBandwidth * 0.35) && viClAvgLd < (
                netBandwidth * 0.35) && stdRcvMax2BorFrVi)
            {
                    cl_bw[1] -= (stdClMaxBwBor * 0.2);        /*
                        remove the borrowed bandwidth from video
                        class bandwidth */
                    cl_bw[2] += (stdClMaxBwBor * 0.2);        /*
                        add back the bandwidth to the standard
                        class bandwidth */
                    stdRcvMax2BorFrVi = false;
                                /* do not allow standard class to
                         request for this bandwidth again until
                        video class borrows it again */
                    viBorMaxFrStd = true;
                                /* allow video class to borrow
                        maximum bandwidth again from standard
                        class */
            }


            /* when voip class borrowed normal bandwidth from
                standard class bandwidth and it is not using it
                at the moment
             * and standard class has not recovered this
                bandwidth for the first time, then give the rest
                of the borrowed bandwidth back */
            if(cl_bw[0] > (netBandwidth * 0.5) && voClAvgLd < (
                netBandwidth * 0.5) && stdRcv2BorFrVo)
            {
                    cl_bw[0] -= (stdClNorBwBor * 0.3);        /*
                        remove the borrowed bandwidth from voip
                        class bandwidth */
                    cl_bw[2] += (stdClNorBwBor * 0.3);        /*
                        add back the bandwidth to the standard
                        class bandwidth */
                    stdRcv2BorFrVo = false;
                                /* do not allow standard class to
                         request for this bandwidth again until
                        voip class borrows it again */
                    voBorFrStd = true;
                                        /* allow voip class to
                        borrow normal bandwidth again from
                        standard class */
            }

            /* when video class borrowed normal bandwidth from
                standard class bandwidth and it is not using it
                at the moment
             * and standard class has not recovered this
                bandwidth for the first time, then give the rest
```

```
                        of the borrowed bandwidth back */
                if(cl_bw[1] > (netBandwidth * 0.35) && viClAvgLd < (
                    netBandwidth * 0.35) && stdRcv2BorFrVi)
                {
                        cl_bw[1] -= (stdClNorBwBor * 0.2);        /*
                            remove the borrowed bandwidth from video
                            class bandwidth */
                        cl_bw[2] += (stdClNorBwBor * 0.2);        /*
                            add back the bandwidth to the standard
                            class bandwidth */
                        stdRcv2BorFrVi = false;
                                /* do not allow standard class to
                             request for this bandwidth again until
                            video class borrows it again */
                        viBorFrStd = true;
                                     /* allow video class to
                            borrow normal bandwidth again from
                            standard class */
                }
        }

}
```