DISSERTATION

Trading Consistency for Availability in a Replicated System

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

O.Univ.Prof. Dr.techn. Richard Eier
Univ.Lektor Dr.techn. Karl Michael Göschka
Institut für Computertechnik, E384

und

o.Univ.Prof. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme, E184

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Univ.Ass. Dipl.-Ing. Robert Smeikal
Matrikelnummer: 9327703
Lederergasse 8/5, 1080 Wien

Wien, Juni 2004 _____

Robert Smeikal:
Trading Consistency for Availability in a Replicated System

Faculty of Electrical Engineering and Information Technology
Vienna University of Technology, Austria

This research has been supported by Frequentis GmbH

Abstract

Distributed systems are of unprecedented interest and importance today. Their omnipresence pervades many aspects of our daily lives leading to an increasing demand for dependability of such systems, sometimes very critically as in systems for air traffic control or public safety. As systems are expected to continue functioning even in the presence of failures, fault-tolerance as one means to enhance dependability is of particular interest.

It is common to build such systems using distributed objects, which are replicated to provide the redundancy necessary for fault-tolerance. Furthermore, data integrity rules called constraints are defined among them. As concurrent access from different clients is a basic requirement, the isolation necessary to offer a comprehensible view to clients has to be provided. The system takes care of all those functions.

In this regard, there are three types of consistency to be analyzed, defined, and compared: Replica consistency, which defines the correctness of replicated data, concurrency consistency, which defines the correctness of concurrent access to a single set of replicas, and constraint consistency, which defines the correctness of the system state with respect to the set of constraint conditions.

Deploying these considerations, this thesis examines a very specific aspect of fault-tolerant distributed systems: the explicit trade-off between availability and constraint consistency. The type of replica consistency is used as a means of configuring the trade-off between constraint consistency and replica availability. If the system faces site crashes or network partitions, less but well controlled constraint consistency is accepted to gain higher availability of objects. Furthermore, a model for enabling this trade-off within a distributed system is introduced, the fault-tolerant naming service (FTNS). The key idea of the respective system architecture is to use asynchronous replication of persistent object-states, while operating on objects synchronously. During normal operation the system is set up like a conventional distributed system, while propagating persistent object-states prepares for degraded scenarios.

Additionally, a proof of concept implementation is presented, the Distributed Telecommunication Management System (DTMS): It is an object-oriented, distributed and highly available software for managing a telecommunication network to be used in air traffic control.

Kurzfassung

Verteilte Systeme sind heutezutage von großem Interesse und einer noch nie da gewesenen Wichtigkeit. Ihre Allgegenwart durchdringt viele Aspekte unseres täglichen Lebens, und das wiederum führt zu einem steigenden Bedürfnis nach Zuverlässigkeit solcher Systeme, bisweilen sehr kritisch wie in Systemen zur Flugsicherung oder zur öffentlichen Sicherheit. Es wird von solchen Systemen erwartet, auch in Gegenwart von Fehlern weiter korrekt zu funktionieren, und daher ist Fehlertoleranz als ein wichtiges Mittel zur Steigerung der Zuverlässigkeit von besonderem Interesse.

Es ist üblich, solche Systeme mit Hilfe verteilter Objekte zu bauen, deren Replikation die nötige Fehlertoleranz ermöglicht. Weiters werden Datenintegritätsbedingungen auf diesen Objekten definiert. Da gleichzeitiger Zugriff durch verschiedene Benutzer eine Basisanforderung darstellt, muß auch für eine Isolation dieser Zugriffe gesorgt werden, die eine nachvollziehbare Systemsicht für den Benutzer bewirkt. Das System bietet all diese Funktionalität.

Diesbezüglich müssen drei Typen von Konsistenz analysiert, definiert und verglichen werden: Konsistenz der Replikation, die die Korrektheit der replizierten Daten definiert, Konsistenz der gleichzeitigen Zugriffe, die die Korrektheit der gleichzeitigen Zugriffe auf einen einzelnen Satz an Kopien definiert, und Konsistenz bezüglich der Integritätsbedingungen, die die Korrektheit des Systemzustandes unter Berücksichtigung der Integritätsbedingungen definiert.

Unter Anwendung dieser Überlegungen untersucht diese Arbeit einen sehr speziellen Aspekt fehlertoleranter verteilter Systeme: Die explizite gegenseitige Abhängigkeit von Verfügbarkeit und Konsistenz bezüglich der Integritätsbedingungen. Der Typ der Konsistenz der Replikation wird als Mittel verwendet, um diese gegenseitige Abhängigkeit zu konfigurieren. Wird das System mit dem Absturz von Knoten oder der Trennung des Netzwerkes in Teilbereiche konfrontiert, so wird eine geringere und kontrollierte Konsistenz bezüglich der Integritätsbedingungen akzeptiert, um höhere Verfügbarkeit zu erlangen. Weiters wird ein Model vorgestellt, das ein Ausnützen dieser gegenseitigen Anhängigkeit in einem verteilten System ermöglicht: Das "Fault-tolerant Naming Service" (FTNS). Die Schlüsselidee der zugehörigen Systemarchitektur ist es, asynchrone Replikation von bereits permanent gespeicherten Objekt-Zuständen zu betreiben, während auf den eigentlichen Objekt-Instanzen synchron gearbeitet wird. Im normalen Betrieb arbeitet das System wie ein konventionelles verteiltes System, während das Verteilen von permanent gespeicherten Objekt-Zuständen auf Fehlerszenarien vorbereitet.

Zusätzlich wird eine Implementierung des vorgestellten Konzepts präsentiert: Das "Distributed Telecommunication Management System" (DTMS). Dies ist eine objekt-orientierte, verteilte und hochverfügbare Software zur Steuerung von einem Telekommunikations-Netzwerk, das in der Flugsicherung Verwendung findet.

Acknowledgement

Many people deserve my deepest gratitude:

First of all I would like to thank my thesis advisor Professor Richard Eier for his continued support throughout my work at university. He always helped and encouraged me to pursue the goal of writing this thesis. I am also grateful to my second thesis advisor Professor Mehdi Jazayeri for his assistance and valuable comments in the final phase of my thesis.

I am also very much in debt to Karl Michael Göschka, who taught me how to write scientific publications and was always willing to contribute with his hints and discussion.

My colleagues at university provided countless and useful hints: Friedrich Bauer, Klaus Darilion, Jürgen Falb, Martin Jandl, Wolfgang Kampichler, Christoph Kurth, Rudolf-Michael Liebhart, Roman Popp, Wolfgang Radinger, Paul Smutny, Alexander Szep.

I developed the basic idea of my thesis within a cooperation of my university with the companies Frequentis GmbH and PDTS GmbH. The people there were very willing to help with their hints and background knowledge: Volkmar Hausharter, Hubert Künig, Gerhard Kurz, Dietmar Mittermair, Helmut Reis, Reinhard van Loo.

Most of all I owe many thanks to my friends and family, who supported me throughout my whole professional career. None of my achievements would have been possible without their patience, understanding, and advice. Thank you very much!

Contents

Abbreviations

ACID	Atomicity Consistency Isolation Durability
ATC	Air Traffic Control
CBSE	Component-Based Software Engineering
CORBA	Common Object Request Broker Architecture
DTMS	Distributed Telecommunication Management System
EJB	Enterprise Java Beans
FTNS	Fault-Tolerant Naming Service
GUI	Graphical User Interface
HDLC	High Level Data Link Control
ID	Identifier
IOR	Interoperable Object Reference
LAN	Local Area Network
OCL	Object Constraint Language
OMG	Object Management Group
OSI	Open Systems Interconnect
QC	Quorum Consensus
ROWA	Read One Write All
UML	Unified Modeling Language
VCS	Voice Communication System
WAN	Wide Area Network

Chapter 1

Motivation, Objectives, and Technical Baseline

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport, May 1987

1.1 Motivation

Distributed systems are of unprecedented interest and importance today. Their common principles and practices underlying the design and implementation are an aspect for many disciplines in computer science and engineering, such as hardware, embedded and real-time systems, component-based systems and middleware, software agents, distributed databases and replication, communication systems, and ubiquitous computing. Among others, they serve as means for fault-tolerance, load balance and increased performance, and for connecting users and services.

The omnipresence of distributed systems pervades many aspects of our daily lives leading to an increasing demand for dependability of such systems, not only for apparently safety critical applications, as in systems for air traffic control or public safety, but for many innovative applications of distributed systems in general. However, often dependability is only taken into consideration for highly specialized safety critical applications. Unfortunately, the complexity of distributed systems leaves an undependable system uncontrollable and unmanageable. Therefore, the key element for achieving scalable and maintainable distributed and complex software systems is dependability and it needs to be transparently integrated into the application deployment environment. As systems are expected to continue functioning even in the presence of failures, fault-tolerance as one means to enhance dependability is of particular interest.

One important way of introducing fault-tolerance is to add functionality to the software, which is able to respond to such faulty system conditions [27], where failures in lower system levels (i.e. hardware failures, failures of driver software) can be classified as site failures (a particular site is not working) and network failures (in particular network partitions, where a group of sites cannot be reached but is still

operating). In order to enable reuse and to free deployer and developer from dealing with such fault-tolerance functionality, it has to be encapsulated in well-defined and coherent system parts.

However, improving single component availability is insufficient, but rather fault-tolerance needs to be incorporated into the system architecture itself. Even worse, there is no such thing like a "fault-tolerance"-module that can be added later on. In fact, dependability is an aspect of every part of the system. Therefore, availability, reliability, and fault-tolerance have to be incorporated into design and architecture, component infrastructure services, and interaction and composition standards.

Replication is well proven to provide fault-tolerance and a plethora of replication protocols exists. Unfortunately, state-of-the-art distributed component-based software systems are not just distributed databases. Therefore, replication cannot be deployed isolated, but rather only the integration of distribution, persistence, and replication can provide a dependable distributed system.

When dependability has to be optimized, this can be achieved by trading it against other non-functional requirements, especially performance and consistency. The ***key focus*** of this thesis is on the explicit trade-off between availability (as a measure of dependability) and constraint consistency (as opposed to the other, different types of consistency: replica and concurrency consistency). Furthermore, this thesis introduces a system model for incorporating this trade-off into a distributed system, the fault-tolerant naming service (FTNS), along with a real-life implementation of this model as a proof of concept.

Considering related work, existing frameworks for distributed applications either deploy strong consistency (infrastructure-controlled) or leave replica management entirely to the application. Yet, strong consistency is not always desirable, because it also implies strong limitations of availability. Generally, the trade-off between replication availability and consistency cannot be configured in such systems. The presented approach on the other hand introduces an idea and an implementation for a tuning of this trade-off between availability and constraint consistency in order to optimize dependability. This contribution exceeds the state of the art in the field of distributed systems.

The ***key architectural concept*** of this approach is to use asynchronous replication of persistent object-states, while operating on objects synchronously. During normal operation the system is set up like a conventional distributed system, while propagating persistent object-states prepares for degraded scenarios.

The ***key system model*** for tuning the trade-off is called fault-tolerant naming service (FTNS) and comprises the fault-tolerant mapping of object identity to object reference: Since the mapping of object identity to reference is dependent on the underlying mechanisms instilled to provide fault-tolerance, the naming service has to be aligned with the replication. The FTNS is locally available at each site. The major difference to a highly available naming database is the dependency of the mapping on the current failure scenario.

Putting it together, this thesis aims at a concept for optimizing dependability in distributed component-based software systems by dealing with failures of nodes and links in an innovative way. Replication is used as means to provide fault-tolerance, but with a focus on the trade-off between availability and constraint consistency.

This is done using a mixture of asynchronous and synchronous replication techniques. The long-term goal is the fine-grained tuning and measuring of this trade-off to allow an application-specific optimum of availability. Also, a fault-tolerant naming service is proposed, which is used to implement this trade-off by resolving object identities based on the current failure scenario and the configured replication strategy.

1.2 Technical Baseline

The following section briefly recalls the basic terminology in the realm of fault-tolerance in distributed systems along with relevant technology.

1.2.1 Dependability

The following definition of dependability is quoted from [27]:

> *Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system is its behavior as it is perceived by its users, where a user is another system (human or physical) which interacts with the computer system. Dependability is a general concept, and depending on the application, different attributes can be emphasized. The most significant attributes of dependability are reliability, availability, safety, and security.*

According to [32], the attributes of dependability can be gathered into three main classes: The dependability impairments, the dependability means and the dependability measures. Their taxonomy is shown in figure 1.1.

- Dependability impairments are undesired (not unexpected) circumstances resulting from or causing un-dependability, whose definition is very simply derived from the definition of dependability: reliance cannot, or will not, be any more justifiably placed on the service.
- Dependability means are the methods, tools and solutions enabling (a) to provide with the ability to deliver a service on which reliance can be placed, and (b) to reach confidence in this ability.
- The dependability measures enable the service quality resulting from the impairments and the means opposing them to be appraised.

With respect to fault-tolerance, reliability and availability are the most significant attributes:

Reliability describes the continuity of service. In terms of mathematical definitions, reliability is the probability that once a service request is accepted it is completed successfully. If a service can be completed even after the occurrence of a

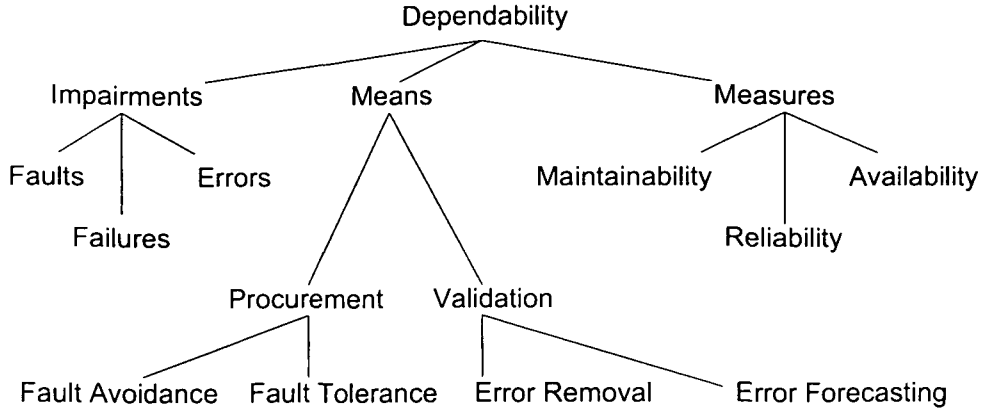

Figure 1.1: Dependability.

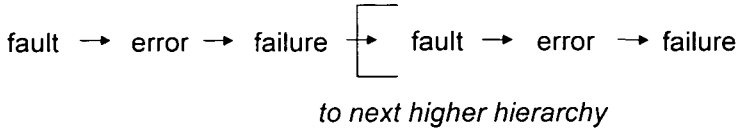

Figure 1.2: Fault/failure chain.

failure (this is typically important if the completion of the service takes much longer than the classical reliability criterion MTTF — mean time to failure) it is also called recovery-enhanced reliability [24].

Availability deals with the readiness for usage. A simple measure for availability is the probability that a system is operational at an arbitrary point in time. Availability and reliability are independent attributes. For instance, a system spending a significant amount of time recovering from failures to complete operations in progress successfully might be highly reliable but due to its recovery time less available [24].

Summarizing [27], the failure of a system can be defined in the following way: A system ***failure*** occurs when the system behavior is not consistent with its specification. An ***error*** is an incorrect internal system state. It is that part of the system state which is liable to lead to subsequent failures. The cause of an error is a fault. A ***fault*** is a system defect that has the potential of generating errors. In addition, a failure can be seen as a fault on the next higher system layer. This taxonomy is shown in figure 1.2. Furthermore, a system is fault-tolerant if it can mask the presence of faults in the system by using redundancy. The goal of fault-tolerance is to avoid system failures, even if faults are present, i.e. it is the ability of a system to continue functioning while a failure is still unrepaired or even undetected [24].

Besides the books of Laprie [32] and Jalote [27], Kopetz [28] provides an excellent treatment of the state of the art in the field of dependability.

1.2.2 Replication

Replication is the process of maintaining multiple copies of the same entity at different sites. Replication protocols and systems achieve high availability by replicating entities in failure-prone distributed computing environments [24]. Several replication algorithms can be used to make a system redundant, whereby the aspect of consistency, which concerns the compliance of the system to well defined rules of replica synchronization, must be considered. Replication is addressed in detail in chapter 2.

1.2.3 Unified Modeling Language

The Unified Modeling Language UML is the industry standard for the analysis and design of object-oriented systems and advanced by the Object Management Group OMG. The OMG is an international organization, that promotes the theory and practice of object-oriented technology in software development. UML is used throughout this work wherever applicable. Details about UML can be found in the official standard at the OMG internet site http://www.omg.org/ and in numerous books.

1.2.4 Component-Based Software Engineering

Component-based software engineering is a subdiscipline of software engineering and primarily concerned with three functions [23]:

- Developing software from preproduced parts.
- The ability to reuse those parts in other applications.
- Easily maintaining and customizing those parts to produce new functions and features.

Therefore, component-based software engineering is primarily concerned with software engineering "in the large". In addition to object-orientation as methodological framework to develop the individual software components with respect to their interactions (software engineering "in the small"), the development described in this work adheres to these principles. Furthermore, the terms related to component-based software engineering described in and quoted from [23] are used, most notably:

> *A **software component** is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
>
> *A **component model** defines specific interaction and composition standards. A **component model implementation**is the dedicated set of executable software elements required to support the execution of components that conform to the model.*

*A **software component infrastructure** is a set of interaction software components designed to ensure that a software system or subsystems constructed using those components and interfaces will satisfy clearly defined performance specifications.*

*An **interaction standard** specifies the type of explicit context dependencies a component may have. It covers both direct and indirect interactions that may exist between components.*

*A **composition standard** defines how components can be composed to create a larger structure and how a producer can substitute one component to replace another that already exists within the structure.*

1.3 Potential Applications

1.3.1 Distributed Telecommunication Management

As a case study and proof of concept the most important facts on concept, architecture and implementation of a Distributed Telecommunication Management System (DTMS) are presented in chapter 4. The DTMS controls a networked voice communication system and is developed by the Austrian company Frequentis GmbH. Major requirements for the DTMS are fault-tolerance against site or network failures, transactional safety, and reliable persistence. In order to provide distribution and persistence both transparently and fault-tolerant, a two-layer architecture facilitating an asynchronous replication algorithm is introduced.

However, the concepts presented in this thesis aim at a general technique for increasing the dependability of such applications, which benefit from the increased availability at the cost of less consistency. In the following, a list of potential fields of application are enumerated [16].

1.3.2 Ubiquitous and Pervasive Computing

Computer systems and communications networks now are ubiquitous in our daily lives. Even more so, they will increasingly be integrated into our personal environment, also known as "ambient", "pervasive", or "ubiquitous" computing. While this benefits society, increases our productivity and our personal comfort, it also makes our life more dependent upon these systems. The concerns of dependability (e.g. faults, errors, and failures) have been enlarged by the massive connectivity and fine-grained distribution provided by the Internet and wireless applications.

1.3.3 Air Traffic Control and Public Safety

As stated in the official document concerning transport challenges until 2010, the European air traffic control (ATC) system is partitioned three times more than a comparable area in the USA. This demands the creation of the Single European Sky, one of the European Union's current priorities. Beside a comprehensive cooperation

of the different national air traffic control systems, distributed intelligence can facilitate the implementation of this vision. "Flight Objects" are a form of distributed intelligence and are currently developed by Eurocontrol. These flight objects are used to model all data relevant for ATC for a flight, motivated by new requirements caused by "Free Flight" — ATC without human air traffic controllers. In contrast to other object-based distributed systems it is not required to provide a consistent view of each attribute for every user at every location. Instead, some users may use outdated data, others are only interested in some data of an object. Together with replication, such requirements allow for higher availability as compared to systems with strict consistency requirements. Dependability is an inherent requirement in air traffic management, where the ability of coping with unscheduled events helps to ensure safety. The same is true for other safety critical applications as for command and control centers for public safety or public transportation.

1.3.4 Health Care System

Today's Health Care Systems are widely distributed but lack of interaction services. Data of patients is in most cases used at different independent facilities (e.g. hospital, family doctor, medical specialists, health insurance companies, etc.) without the ability to exchange once recorded data like X-ray photographs. This inability stresses the patient with repeated, unnecessary tests or even worse with wrong diagnosis. Emergency cases require patient data immediately without putting strong efforts on consistency of the entire patient log. A distributed set of patient data objects each having a certain portion of redundant data is capable of minimizing the number of necessary surveys to a justifiable level. Dependability and fault-tolerance of such a system are an absolute must.

1.3.5 Fleet Management

Saturation of road traffic is still a serious problem in industrialized urban areas such as the Ruhr, the Randstad, northern Italy and southern England [11]. Failure to control road traffic has worsened the situation in the major cities. The stop-start motoring characteristics of bottlenecks result in higher emissions of pollutants and greater energy consumption which is of particular importance in valuable nature and environment. Large dispatch companies having a multitude of transport vehicles need a sophisticated means for the management of their fleet. In order to avoid empty trucks and facilitate operation at high capacity, data objects affiliated to the vehicle can take care in connection with GPS (Global Positioning System) for an autonomous management of their tasks. The service of delivery in time has to be as dependable as possible.

1.3.6 Control Systems in Experimental Physics

Control systems of experimental physics facilities (such as particle accelerators and multi-antenna radio telescopes) need to be fault-tolerant to improve availability and reliability of the devices within the facility. Any unavailability might hinder

acquisition of important scientific data or damage the facility or hurt its personnel. The results of this work can be used to improve the availability of such facilities by enhancing their control systems with fault-tolerance.

1.4 Contribution and Publications

The contribution of this work is threefold. Foremost, new terms are introduced and related to established terminology and theory. The focus lies on clarifying the differences and interrelation of the three types of consistency (chapter 2).

Secondly, presented in chapter 3, the ***key focus*** of this thesis is explained: The trade-off between availability and constraint consistency. Also, the ***key architectural concept*** along with the ***key system model (FTNS)*** is introduced that is designed to enable the mentioned trading in a simple way. The result is a flexible, robust and highly available framework for managing object-oriented software in a distributed and persistent system.

Thirdly, the Distributed Telecommunication Management System (DTMS) is presented. It serves as the proof of concept and is an example of how the proposed conceptual framework can be utilized. Design principles and requirements are outlined, followed by an overview of the proposed architecture along with a description of the system components. The DTMS is described in chapter 4.

Following the presentation of the results, the work concludes with related and future work.

Publications The following publications are used in parts of this thesis without being referenced individually.

[47] Smeikal, R.; Göschka, K.: *Fault-tolerance in a Distributed Management System: a Case Study.* Proceedings (ISBN 0-7695-1877-X) of the 'IEEE/ACM International Conference on Software Engineering', pp.478, Portland, Oregon, USA, May 3–10, IEEE Computer Society/ACM, 2003.

[48] Smeikal, R.; Göschka, K.: *Fault-tolerant Distribution and Persistence of Objects Using Replication.* Poster Session at the '23rd IEEE International Conference on Distributed Computing Systems 2003', Providence, Rhode Island, USA, May 19–22, IEEE Computer Society, 2003.

[17] Göschka, K.; Reis, H.; Smeikal, R.: *XML Based Robust Client-Server Communication for a Distributed Telecommunication Management System.* **(track best paper award nomination)** Proceedings (ISBN 0-7695-1874-5) of the 'Hawaii International Conference On System Sciences HICSS-36', p.122, Big Island, Hawaii, USA, Jan 6–9, IEEE Computer Society 2003.

[18] Göschka, K; Smeikal, R.: *Using Replication for high Availability of a distributed Management System.* Frequentis GmbH Tech:News, Feb 2003.

[16] Göschka, K.; Jandl M.; Smeikal, R.; Szep A. (Authors in alphabetical order): *Dependable Distributed Systems.* European Union Framework Programm 6 Project Proposal, Strategic Objective 2.3.2.3 "Open development platforms for software and services", Feb 2004.

Chapter 2

Model and Terminology

"Buying books would be great if we could also buy the time to read them."

Arthur Schopenhauer, Parerga und Paralipomena

2.1 Distribution, Persistence, and Replication

As describe in chapter 1, replication for fault-tolerance cannot be deployed isolated in a distributed component-based software system. It needs to be integrated with distribution and persistence, which is outlined in the following section.

Reasons for transparently distributed computing are well understood and proven implementations exist [50], for instance:

- The Common Object Request Broker Architecture CORBA [1] from the OMG (Object Management Group) without its complementary services.
- Similarly, the core of the Component Object Model COM+ (formerly Distributed COM) from Microsoft provides transparent distribution.
- The Remote Method Invocation RMI [2] from Sun.
- The Simple Object Access Protocol SOAP [3] from the World Wide Web Consortium.

The physical location of objects is hidden from the clients (other objects using them), where objects can be spread over multiple sites, relate to each other via references and are identified by a system-wide ID. Figure 2.1 illustrates the idea of location transparency.

Reasons for transparent persistence are well understood, too, and solutions exist as well, ranging from commercial products (Java Data Objects [2], OMG's Persistent State Service [1]) to sophisticated, experimental frameworks (THOR [33]). The persistence of objects is hidden from the clients using them, where objects survive the client who created them and can be restored from stable memory. Figure 2.2 illustrates the idea of transparent persistence.

However, large and complex applications usually demand *both* transparent distribution and persistence. A basic setup is depicted in figure 2.3. According frameworks are state of the art and the major products in the field are:

- Enterprise Java Beans (EJB) and the Java 2 Enterprise Edition (J2EE) services: EJB is Sun's answer to the need for component-based distributed business applications. The EJB specification defines system services that are available at EJB platforms: remote method invocation (RMI), persistence, transaction, life cycle, and security.
- Component Object Model (COM+) and its services: Microsoft's COM+ provides distribution along with several system services: security, transactions, scalability. Enhanced with IPersist objects, it provides persistence. As a successor and platform for distributed computing, Microsoft introduced its .Net platform, which enables distributed and persistent objects completely transparently. It is capable of encapsulating COM+ components.
- CORBA and its services: CORBA provides system and language independent distribution. The CORBA services provide transactions, persistence, and other services. A Persistent State Service (PSS) implementation is a CORBA-compliant object-oriented database. PSS storage objects can store any Interface Description Language (IDL) types and can be integrated with the CORBA Transaction Service.

The mentioned systems provide advantages from both, distribution and persistence, but are only satisfying as long as *all* parts of the system are available. However, many applications require fault-tolerance. Improving single component availability for this purpose is insufficient, but rather fault-tolerance needs to be incorporated into the system architecture itself, as quoted in [24]:

> *We must immediately discard the obvious option of composing distributed systems from ultra-available and ultra-reliable components, whose properties are so good that they raise the corresponding properties for the whole system to the desired level. Aside from questions of expense and feasibility, this approach collapses under the weight of increasing scale. As the size of the distributed system grows, its components' availability and reliability would also have to increase with it, violating a basic prerequisite for scalability in distributed systems: that its local characteristics be independent of its global size and structure. Therefore, the distributed system designer needs to instill mechanisms that combat the effects of failures, into the system architecture itself.*

Figure 2.4 depicts the proposed system setup, where the persisted object data is replicated to other sites, in order to provide enough redundant information to replace any failed object dynamically. In the best case and depending on the particular replication strategy, the application is available if *any* site is available, because now objects can be restored at every site. Hence, the idea here is to provide scalable fault-tolerance for both distribution and persistence by using only a single mechanism — replication.

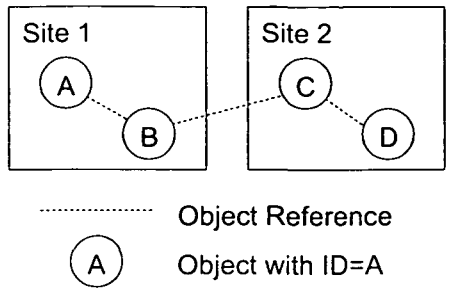

Figure 2.1: Transparent distribution.

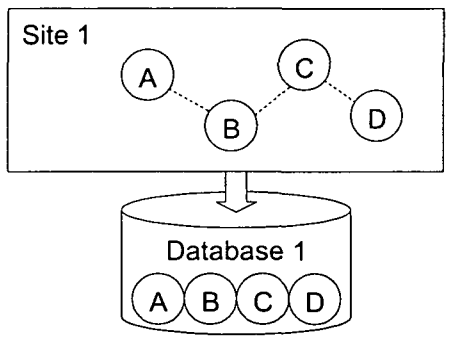

Figure 2.2: Transparent persistence.

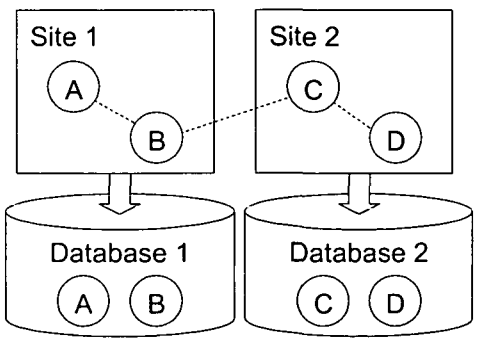

Figure 2.3: Transparent distribution and persistence.

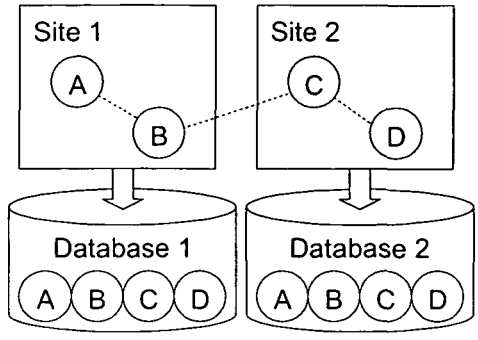

Figure 2.4: Introducing fault-tolerance.

Replication as a means to provide fault-tolerance is well suited and a plethora of replication protocols exist (refer to [24] for comprehensive and in-depth coverage). However, the deployment of replication requires the management of the replicated object data and introduces the new requirement of transparency of the replication. Moreover, if deployed in an object-oriented, persistent and distributed software system, managing multiple physical copies that constitute the state of a single logical copy poses problems on conventional concepts of such systems, which are usually designed to operate on a single copy:

- the preservation of consistency among different logical objects
- the concurrent access to logical objects
- the transactional access to logical objects
- handling of object identities, which are unique among logical objects

A number of different concepts and systems solve the mentioned problems and, in addition, address the trade-off between availability and replica/concurrency consistency. However, none of the existing frameworks explicitly addresses the trade-off between availability and constraint consistency. The following sections clarify the established terminology and differentiate the various kinds of consistency. Bases on this, a detailed reasoning for the proposed trade-off is given in chapter 3 "Trading Consistency for Availability". A number of related frameworks and respective details are described in chapter 5 "Related Work".

2.2 Terminology and Clarification of Terms

Throughout this section the necessary terminology is clarified, which provides the basis for further considerations. Common terms and established theory are briefly summarized and new terms are introduced and made precise by providing definitions. This is especially vital to the topic for two reasons: On the one hand, object-oriented computing and databases use the same terms for different matters, but on the other hand, replication and transactions in the domain of object-orientation are based on older research on databases.

2.2.1 Objects and Persistence

An ***object*** is a runtime entity and is an instance of a class describing the objects' data (member variables) and behavior (methods). Together, the values of an object's member variables make up the ***object-state***. Therefore, the object-state is a structured value. The object's methods may or may not change the object-state upon invocation. Every object has a unique runtime identifier. A member variable storing the identifier of another object is called ***reference***. Other member variables are called ***attributes***. Several objects running on different sites in a distributed system make up a ***distributed application***. All object-states which belong to a particular application make up the ***application-state***, which therefore is a structured value as well. Note, that object-state and object are not the same. An object-state is a value and can be part of an object or it can be serialized and stored. If an object-state is stored to a stable storage, it is called ***persistent object-state***. Together with its class definition, an object-state can be converted into an object (i.e. an instance of the class).

2.2.2 Constraints and Consistency

A distributed object-oriented system is in a ***consistent state***, if a set of data integrity rules called constraints is satisfied. A ***constraint*** defines a condition, which can be evaluated to ***valid*** or ***invalid*** using an application-state as input. Therefore, it defines a mapping from the set of possible application-states to either valid or invalid. A finite number of constraints associated with every application is assumed. Constraints are labelled c_1, c_2, c_3 and so on. The validation vector for all constraints is a property of the current application-state and is called ***constraint consistency***.

Different ***sets of constraints*** are considered and labelled s_n, e.g. $s_1 = \{c_1, c_2\}$ (see [15, 34] for comparable approaches). Sets of constraints range from ***no constraint*** (empty set) to ***all constraints*** (set of all constraints). Typically, certain sets of constraints are made up of strongly related constraints necessary for common use cases, which the application is able to provide independently. A particular application-state is called ***consistent for*** c_n if c_n evaluates to valid using the application-state as input, ***inconsistent for*** c_n otherwise.

An application-state is called ***consistent for*** s_n if all constraints in s_n evaluate to valid using the application-state as input, ***inconsistent for*** s_n otherwise. The process of identifying constraint violations during runtime is called ***validation***.

Formally, constraints can be captured using the OMG's Unified Modeling Language UML and the Object Constraint Language OCL [1], for example.

Constraints can be ***static*** and ***dynamic***. According to the above definition, constraints are considered static in this work. Dynamic constraints can be either event-based or time-based. Event-based constraints are evaluated against an application-state transition rather than an application-state, e.g. "salaries may only increase". Time-based constraints span changes of the application-state over time, e.g. "salaries may increase at most 15 percent within three years". Event-based and

time-based constraints are out of scope but not in contradiction with the concepts presented in this work.

The following code is an example for a static constraint in OCL.

$$self.mood = ('good' \ OR \ 'bad') \tag{2.1}$$

An example for an event-based constraint in OCL is a postcondition using the `@pre`-postfix:

$$\begin{array}{l} context \ Person :: increaseSalary(sum : Integer) \\ post : salary > salaray@pre \end{array} \tag{2.2}$$

2.2.3 Transactions and Consistency

Transactions group subsequent individual method invocations on objects. They possess the usual ***ACID properties*** [7] and are delimited by invoking a begin and a ***commit/rollback*** method:

Atomicity The transaction is performed entirely or not at all. This property is ensured using an ***atomicity control protocol***. In a distributed system this involves a ***distributed commit*** operation. Typically, a coordinator forwards the operations to participants, where the atomicity control protocol ensures, that all sites agree. A widely used such protocol is the ***"two-phase commit"*** protocol. During phase one — the prepare phase — the coordinator asks the participants if they are prepared to commit. In phase two — the commit phase — the coordinator informs the participants of its decision to commit or to rollback.

Apart from the high communication cost, a fundamental property and problem is that no such atomic commitment protocol can guarantee the execution of a transaction to its termination as long as a network partition exists. A detailed reasoning about properties and pitfalls can be found in [7]. However, commit protocols have been designed to handle the failure of the coordinator, which can cause blocking with no participant knowing the outcome until the coordinator is repaired. ***Three-phase commit*** is such a protocol. Recent research deploys ***fault-tolerant consensus algorithms*** to reach agreement on the commit/abort decision without blocking as long as a majority of the participating sites are working. [19] in particular describes the use of the (infamously so-called) Paxos algorithm to achieve distributed consensus on commit/abort and that the two-phase commit is a special case of the general Paxos algorithm using only one coordinator.

Consistency The transaction takes the system from one consistent state (or valid state, i.e. a constraint consistent state, see constraints and consistency in the previous paragraph) to another consistent state. Typically, constraints are validated in the course of performing a transaction which modifies data.

Isolation The transaction is isolated from ongoing update activities, that is, if two or more transactions run concurrently and perform their operations in an interleaved fashion, they serve as the unit of isolation from each other. Without isolation, anomalies can arise, for instance, as presented in [6]: Dirty write, dirty read, fuzzy read, phantom read, lost update, constraint violation (discussed below in more detail). At first, an execution of concurrent transactions is correct, if it is ***serializable***, that is, it is equivalent to some serial execution of the same transactions. If the system takes care that every operation interleaving is serializable, none of these anomalies can occur. However, other weaker ***degrees of isolation*** are possible, which essentially define particular anomalies as being acceptable. Within this work, the type of consistency which comprises anomalies from concurrent access to the same entities is called ***concurrency consistency***. A ***concurrency control protocol*** restricts executions to those executions that are correct.

The formalism to denote theorems and prove such protocols involves the concept of ***"logs"*** or ***"histories"***, which order events belonging to transactions. A common categorization of such protocols is pessimistic and optimistic. Pessimistic protocols prevent situations which potentially lead to non-serializable situations. A well known such protocol is "two-phase locking", which performs lock acquisition in a canonical order in phase one and modifies data and releases locks in phase two. On the other hand, optimistic protocols validate at commit time and resolve conflicts by rollback.

Durability The transaction update is permanently installed. No failure after the commit can cause the results to be lost or be undone.

Constraint consistency and concurrency consistency are logically independent criteria. In accordance with the ACID properties however, validation has to take place before the commit of a transaction is completed in order to enforce a particular constraint consistency. Furthermore, since validation itself is performing operations on the data entities it has to be isolated from other ongoing transactions. Therefore, the transaction concept combines constraint consistency and isolation and hence, they are often implemented in conjunction with each other. This may also be a reason, why they are often treated in a synonymous way in literature, whereas they are in fact different. However, ***concurrency inconsistency (deficient isolation) may affect constraint consistency***.

2.2.4 Client and Server

To avoid confusion due to the overloading of the terms client and server, the following definition is given. A system entity utilized by another system entity is called ***server*** of that entity. A system entity, which utilizes another system entity, is called ***client*** of that entity. Usually, an entity is both, client and server. A system entity can be an object or a component, for instance. This is not to be confused with the GUI-client of the whole system or the client of a client/server-architecture.

2.2.5 Transparent Functionality

Transparency is defined using the above client definition and in respect to a specific client. Functionality of the system (either implemented in the utilized server or elsewhere), that

- neither exposes methods to the client
- nor demands callback methods from the client

is called ***transparent*** to that client. Thus, a server cannot provide fully transparent functionality, only some of its functionality can be transparent.

For instance, if the persistence of an object is transparent to a client, it is called ***transparent persistence***.

2.2.6 Sites, Network, and Distributed System

Sites provide data processing capabilities and stable storage at a particular physical location. Sites are connected via a ***network*** and together, sites and network make up the ***distributed system***. If site or network failures occur, the distributed system is called ***degraded***, ***healthy*** otherwise. Network failures may only lead to ***network partitions*** [43]. In a partitioned state, any two sites in the same partition can communicate and any two sites in different partitions cannot communicate. The number of ways a set of sites with n elements can be partitioned into disjoint, non-empty subsets is described by the Bell numbers (1, 1, 2, 5, 15, 52, 203, 877, ...)

$$B_n = \sum_{k=1}^{n} S_n^k \tag{2.3}$$

where S_n^k describes the Stirling numbers of the second kind, which describe the number of ways a set with n elements can be partitioned into k disjoint, non-empty subsets.

$$S_n^k = \frac{1}{k!} \sum_{i=1}^{k} (-1)^i \binom{k}{i} (k-i)^n \tag{2.4}$$

The Stirling numbers of the second kind can also be defined recursively.

$$S_n^k = S_{n-1}^{k-1} + k S_{n-1}^k \tag{2.5}$$

For example, the set $\{1,2,3\}$ (n=3, $B_3 = 5$) can be partitioned in the following ways:

$$\begin{array}{ll} k=3, & \{\{1\},\{2\},\{3\}\} \\ k=2, & \{\{1,2\},\{3\}\} \\ k=2, & \{\{1,3\},\{2\}\} \\ k=2, & \{\{1\},\{2,3\}\} \\ k=1, & \{\{1,2,3\}\} \end{array} \tag{2.6}$$

Sites exchange ***messages***. Messages are not guaranteed to be delivered, nor can the sender determine, if the message was delivered at all[1]. Thus, network partition is indistinguishable from site crash. Furthermore, arbitrarily long message delays are assumed. Sites and network are ***fail-stop*** [45, 12, 50], Byzantine failures [40, 12, 50] are not considered. ***Reunification*** is the process of merging network partitions in a degraded, but originally non-partitioned system.

2.2.7 Safety and Liveness

To give a brief overview of the theory, parts of the good introductory paper by Gärtner [21] are summarized. In addition to [21], [5] provides an excellent coverage of the topic as well as the main elements of the theory of distributed computing in general.

An execution of a distributed algorithm is an infinite sequence of global system configurations. A property of a distributed algorithm is a set of system executions. A property holds if the set of executions defined by a distributed algorithm is contained in the property's set of executions [4].

In the field of distributed computing, an algorithm is said to be ***correct*** if both the ***safety*** and the ***liveness*** property hold. In [29], Lamport first described these two major classes of properties: Safety informally means that a particular "bad thing" never happens within a system. Liveness on the other hand means that a particular "good thing" eventually happens. A traffic light at a road intersection is an illustrative example: A safety property for this system is the following: "No two traffic lights shall show green at the same point in time". A liveness property for the traffic light could be as follows: "A car waiting at a red traffic light must eventually receive a green signal and be allowed to cross the intersection." Both the safety and the liveness property restrict the possible system executions to correct executions. Together, the liveness and safety property make up a problem specification and can therefore also be considered as a kind of requirement catalogue. ***Termination*** is a common example of liveness in distributed systems. However, liveness properties must not be discrete: ***Guaranteed service***, which states that every request will be satisfied, is such a property.

Examining the affection of safety and liveness by the occurrence of faults results in four possible forms of fault-tolerance depicted in table 2.1:

	live	not live
safe	masking	fail safe
not safe	nonmasking	none

Table 2.1: Four forms of fault-tolerance.

[1] As a matter of fact, message responses can be used to safely conclude that the message did in fact reach the recipient. This has practical relevance: reliable communication over communication media which exhibit transient communication failures, e.g. the TCP/IP protocol. However, without response no assumption about deliverance is possible. In addition, message responses cannot be used to distinguish network partitions from site crash.

However, applications that continuously violate safety are of little practical use and therefore not considered here. If a system is able to stop in a safe state in the presence of faults, it is called ***fail safe***. To continue with the traffic light example, the traffic light would terminate in showing red light in every direction upon a fault. On the other hand, if a system is still safe and live in the presence of faults, it is called ***masking*** as explained in section 1.2.1 about dependability. Such systems are in the focus of this work.

To conclude this section, a quote on the importance of liveness compared to safety is given [32]:

> *While philosophically important, in practice the liveness property [...] is not as important as the safety part, [...]. The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when looking for errors, most of your effort should be devoted to examining the safety part.*

2.2.8 Replication and Consistency

Replication is the process of maintaining multiple copies of the same entity at different sites. Replication protocols and systems achieve high availability by replicating entities in failure-prone distributed computing environments [24]. In the literature different types of entities are considered: un-typed data objects, typed and complex objects, processes, and messages. In this work, typed and complex objects are assumed. The logical object is called ***entity object***, where an entity object's object-state is assembled on a particular site using one or multiple physical copies of the object-state. An entity object is uniquely identified by an ***object ID***.

Replication can be basically classified as synchronous or asynchronous. ***Synchronous replication*** always uses some sort of atomicity control, which ensures that changes are applied in a simultaneous manner, i.e. operations are committed at all participating sites or not at all. Since atomicity control can not guarantee transaction termination as long as network partitions exist, ***service denial*** (also called ***blocking***) must be taken into account. When a process must await the repair of failures at other sites before proceeding, it is called blocked [7]. Generally, algorithms using synchronous replication focus on systems with strict consistency requirements at the cost of possibly blocking behavior.

Asynchronous replication works without atomicity control. Changes are locally committed before they are propagated to participating sites. Therefore, failure at remote sites cannot block a fully functional site (***non-blocking***). On the downside, ***update conflicts*** must be taken into account if the same entity object is altered at different sites or if entity objects are altered in such a way that constraints defined among them are violated. Generally, algorithms using asynchronous replication focus on:

- systems, where denial of service is unacceptable (e.g. CRM - Customer Relationship Management)
- large scale systems, where network failures occur frequently (e.g. wide area networks or ubiquitous computing).
- systems, where heavy transaction load is unacceptable (e.g. small bandwidth). Generally, asynchronous replication is more efficient in terms of communication cost than synchronous replication.
- systems, where updates can be done in such a way that conflicts are not possible (e.g. adding an entry to a set).

The classical correctness criterion (or ***type of replica consistency***) for replicated data is termed ***1-copy serializability*** (1SR): The concurrent execution of a set of transactions on replicated copies must be equivalent to the serial execution of the same transactions on only one copy of each entity. 1-copy serializability is ensured when operations performed on an entity object are reflected on the physical copies of that entity and the system always presents the most current state even under site and network failures. It is appropriate for applications that cannot tolerate any inconsistency of their data. A typical replica inconsistency is ***staleness***. A detailed description and formalization can be found in the appendix of [7].

However, there exists another strong replica consistency criterion, which does not take concurrency issues into account: ***Sequential consistency*** was first defined by Lamport [31] and informally means that the same interleaving of operations occurs at every replica. Any interleaving of operations on a single replica is allowed, but the same interleaving is observed on the corresponding replicas at every node in the system. Using this particular type of replica consistency provides a view of the entity similar as if only a single copy exists, thus decoupling replication consistency entirely from constraint and concurrency consistency criteria.

To clearly differentiate between concurrency control and replica control, concurrency control can be seen as a means to provide the serializability to isolate the concurrent, interleaving access to a particular set of data items, which may also be a single set of replicated copies. The access may stem from parts of even distributed transactions initiated by multiple clients. On the other hand, concurrent access to different sets of replicated copies is not a concurrency control problem, but rather a replica control issue. Therefore, ***replica consistency and concurrency consistency are logically independent.***

Apart from 1SR and sequential consistency, "weaker" types of replica consistency definitions exist, which can be basically classified as ***data centric*** and ***client centric*** [50]. Data centric consistency models are defined with respect to a particular data object. They aim at providing a system-wide consistent view on the data. Client centric consistency models are defined with respect to a particular client of the system and aim at systems which are characterized by a lack of simultaneous updates. They have ***eventual consistency*** in common, meaning that if no updates take place, all replicas will gradually become consistent.

An example for a weaker type of data centric consistency is a correctness criterion called ϵ-serializability [42]. It allows inconsistent data to be seen, but requires that

data will eventually converge to a consistent 1SR state. Transactions are classified as query and update transactions (called ϵ-transactions). Update transactions are propagated to each site asynchronously, therefore replicas of an entity object can differ at any given moment. This is the source of inconsistency, as query transactions can be interleaved with update transactions and therefore allow inconsistent data to be seen. However, the "degree of consistency" (essentially counting the overlapping of query and update transactions) can be controlled and limited. Valid ϵ-logs are serializable logs containing only update transactions (i.e. query transactions are removed).

2.2.9 Replica Control Protocols

Replica control protocols typically guarantee 1-copy serializability, operate in conjunction with concurrency control and atomicity control, and present a logical view, that is equivalent to a non-replicated system. A short enumeration of basic replication techniques is given below.

Read one write all (ROWA): ROWA is the most straightforward type of protocol. Multiple copies must all be updated, of which anyone can be read. Primary copy ROWA and true-copy token ROWA designate a particular copy to be required for writes to succeed, but allow this designation to change dynamically. ROWA protocols can tolerate site failures, but not link failures.

Quorum consensus or voting: ROWA protocols have two major drawbacks: They favor read operations and are unable to tolerate link failures. Quorum Consensus or Voting-based protocols on the other hand don't exhibit these drawbacks. They allow writes to be carried out only on a subset of sites (a so called ***write quorum***). Similarly, reads have to be carried out on a subset of sites (***read quorum***), which must be guaranteed to overlap with any given write quorum (this is termed ***quorum intersection*** requirement). Different Quorum Consensus protocols differ in the way that quorums are formed ranging from simple majority to explicit enumeration and from static to dynamic quorum assignment.

Simple majority QC and ***weighted majority QC*** count the votes of sites in a specific subset to determine if the subset qualifies as a quorum.

A more interesting and flexible QC algorithm is ***general QC for abstract data types (ADT)*** [25]: Data is encapsulated in ADTs and accessed through the invocation of a fixed set of operations at instances of the ADTs. Past invocations relevant to a particular operation are gathered upon invocation of this operation using ***initial quorums*** and results of invocations are applied to copies in ***final quorums***. Each ADT is assigned a set of ***quorum intersection relations***, which are determined according to the necessary information flow between successive operation invocations. The way used to actually form the quorums can be chosen at will as long as the intersection relations are satisfied. The generality of this algorithm arises from the use of arbitrary data types (as opposed to read and write only) and from the use of arbitrary ways to form the quorums.

Quorum consensus on structured networks: This subclass of QC algorithms tries to reduce the number of sites that need to be in quorums to ensure replica consistency by imposing a ***logical structure*** on the sites. An influential

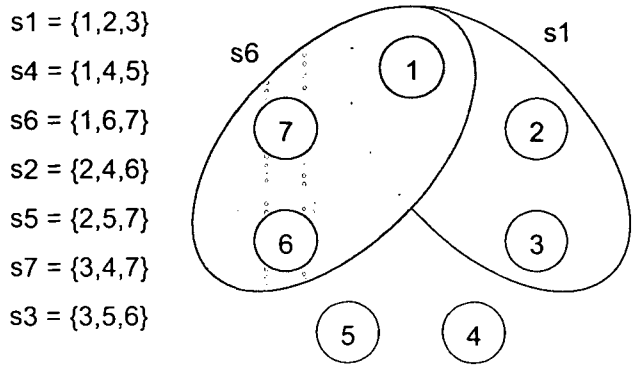

Figure 2.5: Example (n=7) of quorum assignment for the $\sqrt{n}$-algorithm.

work in this field is the ***$\sqrt{n}$-algorithm*** [36]. The major contribution is to calculate the minimal number of sites $\approx\sqrt{n}$ in a system with n sites to participate in quorums in such a way that every quorum overlaps with any other quorum and sites are evenly distributed among the quorums. A possible quorum assignment is depicted in Figure 2.5. Note that each site is in its assigned quorum (e.g. site three is in its quorum $\{3,5,6\}$) and participates in quorums as often as any other site (three in this case). Other protocols using structured networks include the grid protocol, the tree protocol, and multidimensional weighted majority QC. Voting on structured networks can be viewed as a way of trading communication cost (larger quorums) against availability of operations, since the reduced smaller quorums need to be of a specific constitution and cannot be chosen randomly.

To overcome the drawback of rendering objects inaccessible as a result of failure situations, either for read or write access or even for both, ***reconfiguration after site failure or network partitions*** is used.

In the case of site failure, the failure must be reliably detected, which essentially entails that all sites must agree on a new configuration. Furthermore, if any failed site rejoins the protocol, it must run a recovery protocol to get informed or inform about the new system configuration and also to get the latest update. One way of introducing reconfiguration after site failure is the concept of ***generations*** to determine the currency of replicas used in the ***regenerative QC algorithm***.

In the case of network partitions, two major concerns arise: First, entity objects may not be updated in more than one partition concurrently. Secondly, any future partition must be able to see updates performed by previous potentially smaller (reconfigured) quorums. An algorithm achieving this is ***dynamic uniform majority voting***. As opposed to the static uniform majority QC, this algorithm redefines the majority partition as being one which has a majority of the *most current* copies of a data object. This effectively decreases the number of sites necessary to form a valid quorum.

Conventional voting-based algorithms are restrictive in terms of data consistency requirements. ***Weak-consistency algorithms*** are designed to allow data access in different partitions in the case of failure by either classifying transactions or by letting transaction be performed unhindered in different partitions (called ***optimistic***

replication) and resolving conflicts after reunification. However, in the worst case already committed transactions have to be undone later.

Class conflict analysis is an algorithm of the former case. Transactions are classified as particular application-related transaction types, called classes. Conflicts between classes are determined using the read and write set of each class. Upon partitioning, a class conflict graph is constructed, which provides information about any potential order dependencies between classes in different partitions. Cycles spanning multiple partitions in this graph denote the potential for conflicting transaction ordering. Typically, such cycles are removed by deleting certain classes in the graph, i.e. disallowing the system to perform particular transaction types in particular partitions.

Partition logs is an algorithm using optimistic replication and partition logs to record transactions in partitions. After reunification the partition logs are transformed taking semantic properties of transactions into account (e.g. transaction are commutative $T_iT_j = T_jT_i$) in order to reduce conflicts. Remaining conflicts are removed by undoing committed transactions.

2.3 Interrelation of Replica, Constraint, and Concurrency Consistency

As set forth in the previous paragraphs, the term consistency is used in different contexts: replication, concurrency, and constraints. This is known from literature, but many different terms have been used to denote the types of consistency in general and especially the distinction between replication consistency on the one hand and the combination of constraint and concurrency consistency on the other hand. [51] states, for instance, that

> ***internal consistency*** *depends upon the local consistency assertions and* ***mutual consistency*** *depends upon how close the replicas are to each other.*

In [42], the following is said about the types of consistency:

> *[...] we distinguish* ***replica control*** *from the maintenance of* **system internal consistency**, *termed* ***divergence control***. *This distinction is analogous to the distinction between* **coherence control** *(replicas of a single "logical" object) and* **concurrency control** *(system internal consistency).*

In the following, the terms ***replica consistency***, ***concurrency consistency*** and ***constraint consistency*** are used. Apart from clarifying the confusing mixup of terms throughout literature, there is a particular reason for reiterating this distinction here: Even though replica and concurrency consistency are logically different, it is only in the most restrictive case (strong replica consistency and strong concurrency consistency) that they are independent from each other: Replication control

provides ***sequential consistency*** and works on top of the local concurrency control at each site which in turn provides the ***serializability*** necessary to achieve the isolation property of distributed transactions. Together, sequential consistency and serializability provide 1-copy serializability, a correctness criterion combining the independent replication and concurrency consistency criteria.

However, if replica control does not work on top of local concurrency control, concurrency consistency is affected, because replica control now interferes with concurrency control: Concurrency control (building of dependency graphs, acquiring locks [20]) relies on exclusive and up to date access to objects. As weaker types of replica consistency typically don't provide that, update anomalies can no longer be detected safely using these unmodified techniques. This leads to uncontrollable situations in the case of weaker types of replica control. However, even if strong replica consistency is supported, data access from replication needs to be isolated from other client access as well, which is not possible in the case of concurrency control on top of replication control. Therefore, if isolation is needed at all, it has to have exclusive and sole access to all data items — replicated or not — isolating data access not only from clients but also from replication control. In other words, concurrency control is always "closest" to the data items.

As mentioned before, weak types of concurrency control can affect constraint consistency. Similarly, using strong replica control on top of weak types of concurrency control can affect replica consistency. Therefore, the following considerations are limited to types of systems that support ***decoupled strong isolation***. This approach was also chosen in the proof of concept implementation DTMS described in the chapters 3 and 4.

Dependency between replica and constraint consistency arises from the fact that constraints always limit "entity" objects, but are always evaluated using particular, replicated copies, whose state may vary from site to site. In an 1SR environment, this does not make any difference, because all copies reflect the latest changes sufficiently. However, in the case of weaker types of replica consistency, validation might also use a particular outdated view on the entity objects, which later on changes through updates from replication control thereby invalidating object states with respect to particular constraints. This is independent from concurrency consistency, because concurrent access of all clients to a particular set of replicas can be properly controlled but still use an outdated view. The essential conclusion from the analysis of replica and constraint consistency is that ***constraint inconsistency can be a direct consequence of replica inconsistency***.

Figure 2.6 finally illustrates the trading approach. Atomicity, durability, and isolation (AID) are still at the core of the system. Together, replication control and constraint consistency control operate on top of them enabling the envisioned trading.

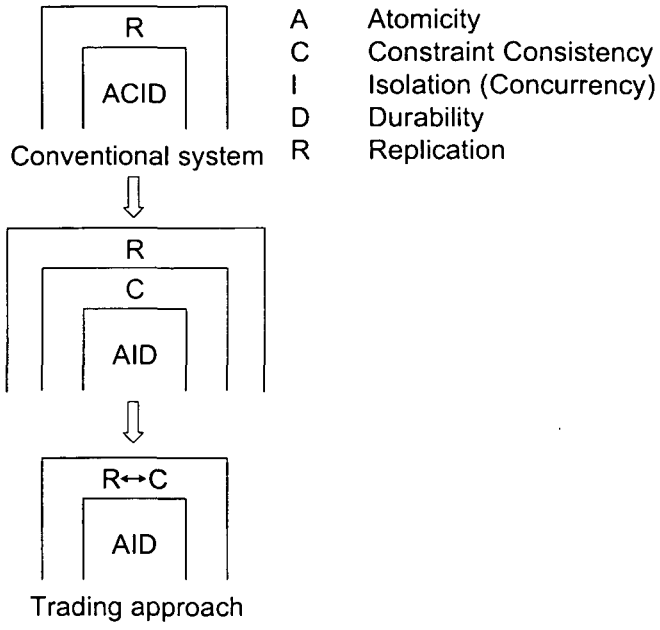

Figure 2.6: Replication, concurrency, and constraint consistency.

Chapter 3

Trading Consistency for Availability

"Don't pay attention to pedantic old farts like me telling you what to do."

Leslie Lamport in reply to the question "What would you like to convey to current and future researchers and practitioners?", IEEE Distributed Systems Online, August 2002

3.1 The Trade-Off

With the intention of optimizing dependability, trading consistency against availability as one of its attributes is described in the following sections.

Generally, there is a trade-off between availability and all different types of consistency. In a replicated distributed system entity objects are available for access at different sites. In a healthy system, object access involves multiple sites to keep the system in a state where it can provide a particular type of replica consistency observed at all sites. Such object access requires the correct functioning of all respective sites. If some of the sites are down or cannot be reached, other sites can be contacted instead, depending on the type of replica consistency and the type of algorithm used to maintain it. However, with the increase of malfunctioning (site crash) or inaccessibility (network partition) of sites it eventually comes down to the point where a client must be denied to invoke a particular operation.

This is the point where replica consistency can be "sacrificed" (i.e. switch to a weaker type of replica consistency) in favor of more replica availability (i.e. invoke the operation nevertheless). Theoretically, this can be exploited to the "lower" bound of replica consistency, where all operations at all replicas regardless of failure scenarios can be conducted. However, since replica inconsistency *can* be the source of constraint inconsistency, such an exploitation might also lead to unwanted constraint inconsistency. Therefore, the type of replica consistency is a means of configuring the trade-off between constraint consistency and replica availability, which depend on each other indirectly. This is the trade-off being addressed in this thesis.

3.2 Architectural Concept and Key Idea

The key architectural concept is motivated by the following requirements, which have initially been derived from the system requirements from the Distributed Telecommunication Management System DTMS. The DTMS was developed by Frequentis GmbH together with researchers at the Vienna University of Technology in the years from 2000 to 2003.

The initial requirement of having particular entity objects write-available at all sites despite arbitrary failure scenarios was abandoned as being too complex, even more so as it turned out to be in contradiction with another requirement: no update conflicts were acceptable. The solution was to require particular entities to be available for non-blocking write access at all times at a particular site only and for read access at all sites despite failures. Quite obviously, a robust asynchronous primary copy ROWA algorithm satisfies this demand. However, asynchronous replication entails potential constraint inconsistency, which is detailed in the previous chapter. Therefore, synchronous operation is used if the system health permits, i.e. during periods where the respective sites are accessible. The system features a synchronous and an asynchronous mode depending on the system health. It switches between asynchronous primary copy ROWA and conventional synchronous operation during runtime. This way, maximum constraint consistency can be guaranteed during periods without failures, but also during degraded operation the constraint consistency demands can be lowered to increase availability.

Therefore, the key idea of the approach is to use asynchronous replication of persisted object-states, but operate synchronously on objects. This approach allows for a rather coarse-grained tuning of the trade-off between replication/constraint consistency and availability, but it is effectively carried out during runtime.

3.3 Switching between Asynchronous and Synchronous Communication

Due to the aforementioned restrictions of distributed consensus (see section 2.2.3), the switching between synchronous and asynchronous communication has to be autonomous at every site with respect to each other site. No distributed consensus is intended to reach a decision on the current mode. Furthermore, from the perspective of a particular site it is disadvantageous to switch all communication to asynchronous if a single remote site cannot be accessed. Therefore, every site monitors the accessibility of each remote site in the system and if a particular site becomes unavailable, only the communication towards this site becomes asynchronous. The remaining communication to all other accessible sites is kept synchronous. Figure 3.1 depicts a simple example. Site 1 and site 2 are in the same partition, hence they communicate synchronously. Site 3 is in a different partition, therefore the communication to site 3 is asynchronous. Updates for site 3 are stored and propagated after reunification.

In the following sections, the operation in a healthy and in a degraded system are described. The term object-readiness is introduced and it is explained how it helps

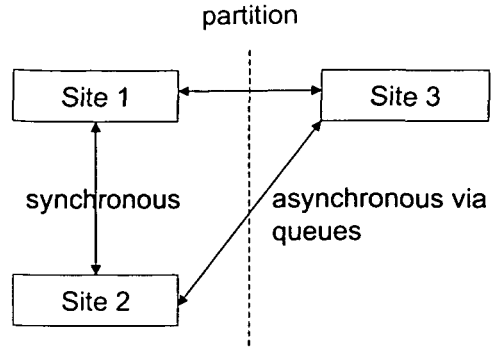

Figure 3.1: Synchronous and asynchronous communication.

to meet the system requirements. The chapter concludes with a discussion of the resulting system properties. The following nomenclature facilitates the explanation throughout the next sections. Within the scope of this work only the replication of complete object-states is considered.

A_1, B_1	Logical entity objects with ID=A_1, B_1 of class A, B
$A_1^{R1}, A_1^{R2}, A_1^{R3}$	Replica of A_1 at site 1, 2, 3
A_1^{S1}	View of A_1 at site 1
$A_1^{S1}(w), A_1^{S1}(r)$	Entity object with ID=A_1 is write-/read-available at site 1

3.4 Operation in a Healthy System

If the system is in a healthy condition, every object runs at a single and pre-defined site, called ***primary site*** of that object. This is illustrated by Figure 3.2. Entity object A_1 (B_1) runs at site 1 (2), its primary site, being referenced by A_1^{S1} (B_1^{S2}). In such a setup, the location and number of objects does not differ from a conventional distributed system, where the location of objects is transparent to the client.

Transactions are executed using these objects. During the distributed commit phase of a transaction, the modified object-states are serialized and passed to the replication algorithm, which is responsible for coordinating the propagation of the modified object-states to remote sites by using a replication algorithm. A "Primary Copy ROWA" algorithm is used. The persistent object-states are read/write-accessed at their primary site. Replication prepares the database at each site to provide replicated object-states in case of a degraded system. It propagates persisted object-states asynchronously to remote sites using queues. The objects themselves do not have to contain replication-related code. If all queues are empty, all replicated object-states reflect all latest changes for all clients.

3.5 Operation in a Degraded System

If the system is degraded, an object is provided at every site using the available replicated object-state as a replacement for the remote object, which is no longer

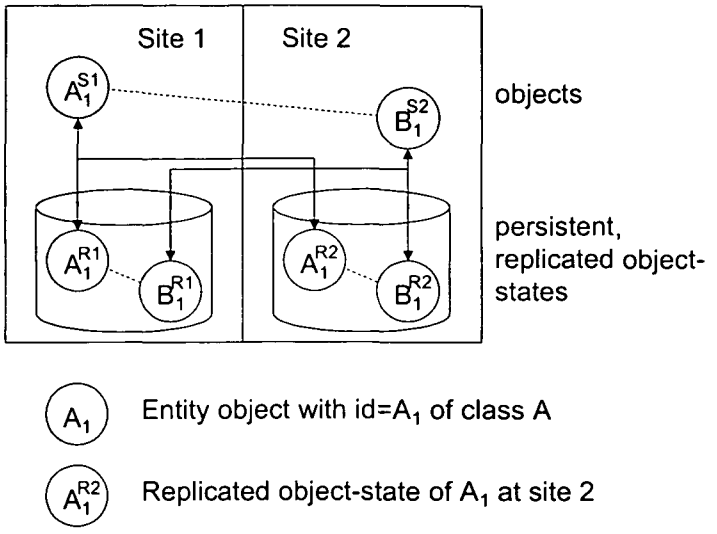

Figure 3.2: Operation in a healthy system.

available. Therefore, at each site every object is available at all times, either remotely and available for read/write-access (healthy system or same partition) or locally and available for read-access (degraded system and different partition). At the primary site of an object read/write-access is always possible. Figure 3.3 shows an example. Object A_1^{S1} runs at its primary site, it can be read and write accessed. As long as the partition remains, the changes to A_1^{S1} are made persistent at site 1 and stored in a queue to be propagated to site 2 after reunification. Object A_1^{S2} runs at the remote site 2 to be available for read access at site 2. The two objects A_1^{S1} and A_1^{S2} are running at different sites but are assigned to the same entity object A_1. From the perspective of a client of the system, not only the location of objects should be transparent, but also replication. Therefore, it is necessary to map the entity object ID A_1 to an object reference, either A_1^{S1} or A_1^{S2}. This mapping depends on the current failure scenario and the respective site: At site 1, the primary site of A_1, the mapping always evaluates to A_1^{S1}. At site 2, the mapping evaluates to A_1^{S1} if site 1 can be reached and to A_1^{S2} if otherwise. The mapping is achieved by using a ***fault-tolerant naming service***. This naming service is locally queried to obtain the current object references.

3.6 Fault-Tolerant Naming Service

Obtaining an object reference (i.e. locating an object) given an object identifier is always of concern if objects are instantiated out of process or even remotely (compare the CORBA Naming Service [1] or the Java Naming and Directory Interface [2]). Though such a naming service is already needed for basic distribution, the matter gets complicated if, as in the proposed architecture, different views of the same entity objects might run in different places due to a particular degradation scenario. On the one hand, the naming service maintains information about the references to objects (where a reference also determines the objects' location), on the other hand,

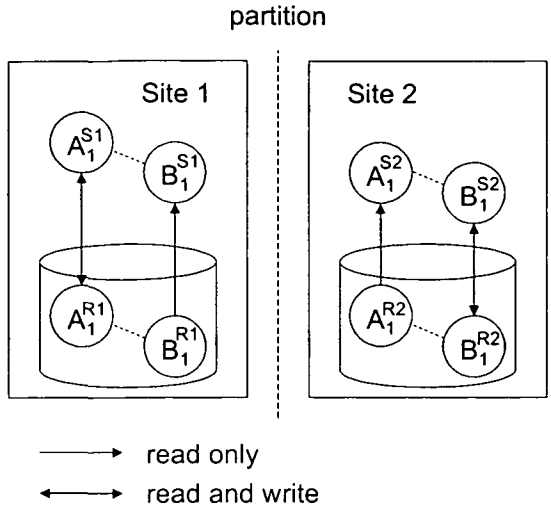

Figure 3.3: Operation in a degraded system.

the replication functionality determines where and how objects can be instantiated if degradation occurs. Thus, the mapping of identity to reference is dependent on the underlying mechanisms instilled to provide fault-tolerance. Furthermore, the required fault-tolerant naming service has to be available at every site at every time.

Additionally, the mandatory use of object identifiers to store references (called "soft references" in the DTMS) introduces the problem of either having to resolve an identity every time the respective reference is used (which usually yields bad performance) or to maintain a kind of identity/reference cache within every object. Within the DTMS this is achieved by using a stereotype `<<inter-site>>`, which denotes "breakable" references. From this modelling construct special methods are generated, which implement this cache.

To avoid a single point of failure, in the proposed approach object identifiers are always resolved using the local FTNS, thereby facilitating a fully distributed FTNS. It resolves object identifiers using two steps. At first, it acquires the presently responsible FTNS, which can be either remote or local (itself) depending on the degradation scenario: If the primary site of an object can be reached, the FTNS there is queried. Otherwise, the local FTNS resolves the identifier to the locally available objects itself. Continuing with the examples in figure 3.2 and 3.3, the naming service at site 1 knows about the references to A_1^{S1} and B_1^{S1}. However, if the system is healthy, the FTNS at the responsible primary site 2 is queried, if a reference to entity object B_1 has to be obtained, which would result in B_1^{S2}.

Remote communication of a local FTNS is always performed with other FTNS components at their respective sites. However, within a site the FTNS communicates with various other components, which is explained in chapter 4 about the DTMS.

3.7 Object-Readiness and Sets of Constraints

In the proposed approach replication provides read access to potentially stale copies in a degraded scenario. Therefore, the system does not provide 1SR. The correctness criterion established is called eventual consistency as described in chapter 2. In the absence of site and network failures the replicated object-states will eventually converge to a consistent 1SR state.

System degradation results in limitations of what the system is currently able to do. Two mechanism are used to communicate current limitations to a client and, even more importantly, allow the client to react selectively: object-readiness and sets of constraints.

Object-Readiness

Object-readiness is a property of every object view at a particular site. It determines the object's current availability for method invocation and is calculated by the replication algorithm. Regardless of the particular correctness criteria, replication algorithms in general limit operations on replicated objects in order to ensure the desired correctness (see [25], for instance). Object-readiness is calculated depending on the availability of the replicated object-states and the condition of the distributed system. Consider the primary copy algorithm used in the presented approach: in presence of a network partition, a write access to a particular object can only be executed if the entity object's primary site can be contacted. Otherwise, only read access is possible. Thus, two types of object-readiness are defined: read-available and write-available. However, with respect to classes, which usually provide many semantically more complex methods other than read and write, this classification has to be made more precise: Method invocation at a read-available object is limited to methods that do not change the object-state. Method invocation at a write-available object is not limited, arbitrary methods can be used.

Generally, the possible values object-readiness can take are implicitly defined by the object's class definition. Every particular method of the n methods a class provides can be either available for invocation or not available, therefore the number of possible types of object-readiness is at first 2^n. However, depending on the replication algorithm only some of the combinations are of use. If the algorithm distinguishes between read and write access, only 4 combinations make sense: unavailable, read only (methods reading the object-state), write only (methods writing the object-state, but don't read), and read/write (methods reading and writing the object-state). In the proposed architecture, only two states are considered: read only (called read-available) and read/write (called write-available). However, if method invocations are recorded rather than the actual changes of data (method invocations are called events in the original work [25]), the method semantics can be exploited more efficiently for higher availability and more types of object-readiness become reasonable. Generally, types of object-readiness range from ***no readiness*** (no method is available for invocation) to ***full readiness*** (every method is available for invocation).

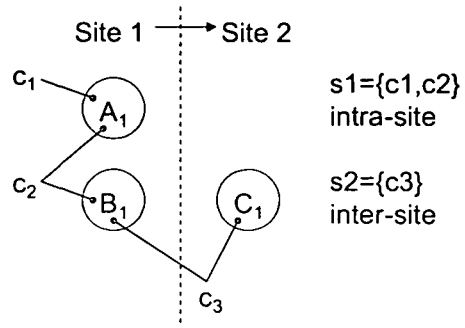

Figure 3.4: Sets of constraints.

Sets of Constraints

Constraints always limit "entity" objects. However, constraints are always evaluated using particular object views. In an 1SR environment, this does not make any difference, because all copies reflect the latest changes sufficiently. On the other hand, 1SR has a drawback: The underlying replication algorithm has to reduce the object-readiness in presence of system failures significantly.

Therefore, in the proposed approach the offered constraint consistency of the application-state is not statically defined, but may change depending on the failure scenario. The current consistency can be communicated to a client using ***sets of constraints***. Consider the following situation depicted in figure 3.4: Three constraints are divided into two sets: s_1 and s_2. s_1 is called "intra-site constraints", because its evaluation is done using constraints which are evaluated using objects which belong to the same primary site, c_1 and c_2 in this case. In contrast, s_2 is called "inter-site constraints", because its evaluation is done using constraints which are evaluated using objects which belong to different primary sites, only c_3 in this case. Entity objects are drawn at their respective primary site: A_1 and B_1 at site 1 and C_1 at site 2.

As mentioned before, sets of constraints make no claim about concurrent access. Control of concurrent access to objects is deployed within the transaction isolation.

Combinations

A client of the system requests to start a transaction along with the ***demand for a particular object-readiness*** of particular objects. The system calculates the possible constraint consistency violations called ***consistency threats*** resulting from conducting the transaction with the demanded object-readiness. The client can react selectively: The client can either "lower" the demanded object-readiness to avoid the potential of violating a set of constraints, or it can conduct the operation if the proposed constraint violation is acceptable. Sometimes the demanded object-readiness cannot be provided to the client at all regardless of constraint violations.

Continuing with the example in figure 3.4, a client at site 1 wants to start a transaction. All of the following considerations apply to a client at site 1 (the index

is omitted), which is indicated by the arrow in figure 3.4. In order to calculate consistency threats, the system uses the following dependencies:

- Validating s_1 needs $A_1(r)$ and $B_1(r)$.
- Validating s_2 needs $B_1(r)$ and $C_1(r)$.

If any of the objects needed for validation of a particular set of constraints is demanded for writing, the according set of constraints needs to be validated at the end of a transaction. For instance, if B_1 is demanded for writing, s_1 and s_2 need to be validated and therefore A_1 and C_1 need to be read.

If the system is healthy, demanding any object-readiness is successful without any consistency threats. All objects are always write-available.

However, if the system is degraded, particular consistency threats are possible. The following table 3.1 lists consistency threats depending on particular client demands for object-readiness.

Client demands	Consistency threat
$A_1(r), B_1(r), C_1(r)$	No threat
$A_1(w)$	No threat
$B_1(w)$	s_2 potentially violated
$C_1(w)$	Not possible

Table 3.1: Consistency threats in a degraded system.

Read-available participation of any particular object in a transaction does not enforce any additional validation, which results in the first line in table 3.1.

If the system is degraded, s_1 can still be normally validated using the write-available object $A_1(w)$ and read access to $B_1(r)$. Therefore, the second line identifies the participation of A_1 in a transaction as being no threat.

Validation of s_2 is triggered by making B_1 write-available, because C_1 is at most read-available (referred to by C_1^{S1} at site 1). However, the evaluation of s_2 with this read-available object may use a stale version of it, because C_1 could have been updated at site 2 (referred to by C_1^{S2} at site 2). If the system is reunified later, this stale object is updated on site 1 with the new object-state from site 2. Since the stale object could have been used for validation at site 1, this can possibly lead to an application-state, which is inconsistent regarding s_2. Thus, the third line in table 3.1 identifies the write-available participation of B_1 as being a consistency threat: "s_2 potentially violated". This constraint inconsistency is a consequence of using a replication consistency criterion other than 1SR. However, the client can obtain the inconsistent application-state after reunification and write to the object-states to correct it, if desired. Alternatively, it can be part of the fault-tolerance strategy to reunify different object-states automatically. ***The advantage of this approach is that B_1 can still be written and C_1 can still be read while being written at site 2 even if the system is degraded.*** Under 1SR-conditions this is not possible.

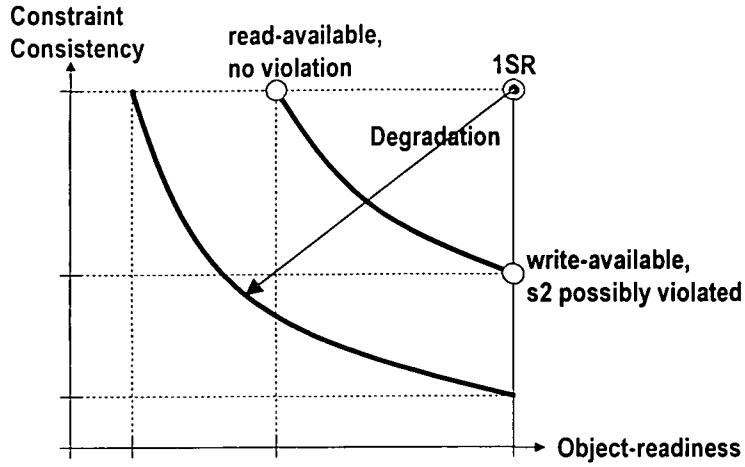

Figure 3.5: Trading constraint consistency for availability — the vision.

Finally, if the system is degraded, C_1 cannot be written at site 1, which is specified in the last line in table 3.1.

In addition, please note that the client demands object-readiness of entity object views and not of particular copies. Therefore, the underlying fault-tolerance strategy is hidden from the client.

However, the above strategy only allows for a rather coarse-grained trading between constraint consistency and availability. The long-term goal is depicted in figure 3.5. In a degraded situation, the system moves along the bold curves increasing availability for consistency and vice versa. For now, only the bold dots are possible system states as described above.

3.8 System Properties

In the following the relevant system properties are discussed.

- Synchronous replication always requires some atomic commitment protocol. In a partitioned network, no protocol can guarantee the execution of a transaction to its termination (either commit or rollback) as long as network partitions exist [24]. Thus, in some cases the protocol has to block the transaction until the failure is repaired.

  In contrast, asynchronous updates are applied instantly and their propagation can be delayed until the network is available. The disadvantage lies in the need to resolve update conflicts, which cannot occur when using primary copy replication. Generally, the possibility of update conflicts depends on the deployed replication strategy.

  The proposed approach uses synchronous communication to get its advantages but avoid its drawbacks by switching to asynchronous communication in the case of degraded scenarios. The disadvantage of asynchronous communication

is reduced by using a primary copy ROWA. However, due to the distribution of primary copies across the system, constraint inconsistency can occur.

- A frequent assumption when building replication-based systems is deterministic behavior of objects [46]. However, this is also very difficult to achieve as quoted in [14]:

  > *Determinism implies that if distinct distributed replicas of the object, starting from the same initial state, receive and process the same set of operations in the same order, they will all reach the same final state. It is this reproducible behavior of the application that lends itself so well to reliability. Unfortunately, pure deterministic behavior is rather difficult to achieve, except for very simple applications. Common sources of non-determinism include the use of local timers, multi-threading, operating system-specific calls, processor-specific functions, shared memory primitives, etc.*

  Also, a good treatment of the subject is given in Poledna's book [41], which is revised from a doctoral dissertation conducted at the Vienna University of Technology.

  In the proposed approach determinism of objects is no prerequisite. The critical operation with respect to non-determinism is the write operation, because it changes the object-state persistently. Since write access is only allowed at the primary site it can be properly managed and synchronized there.

  This is contrary to another paradigm of object replication, which received a lot of scientific attention recently: *group communication* [9, 14]. Here, at all times multiple objects assigned to a single "entity" object exist, even if the system is healthy. The established theory to achieve the necessary update propagations facilitates communication primitives (Atomic Broadcast, View Synchronous Broadcast) to keep a group of replicated objects synchronized. The essential idea of group communication is called "the state-machine approach" [46], where messages are applied to deterministic processes (i.e. object behavior) in the same order and entirely or not at all, thus yielding the same results (i.e. object-states).

- The proposed approach enables the tuning of the trade-off between consistency and availability in principle. The key to configuring this trade-off lies in the design of the correctness criteria and the replication strategy to be more or less restrictive. This ranges from 1SR (along with limited availability) with full consistency to unlimited availability (write access at all times) with possible inconsistency.

- As a disadvantage the system does not support a roll-forward mechanism [14] and therefore does not protect processing. The preservation of consistency is based on a roll-back mechanism, where all changes are undone if a commit fails, thus protecting the data.

- As an advantage the approach does not contain common sources of scalability problems: No centralized services are needed and no centralized data needs to

be accessed. Furthermore, asynchronous communication for replicating data also leverages the wide-area network scalability, because this robust type of communication is much better suited for less reliable wide-area networks: In fact, the DTMS implementation uses a wide-area network. On the other hand, all object-states are replicated to all other sites, which consumes a growing amount of system resources if the number of sites grows.

3.9 Reasoning about Correctness

An algorithm is said to be correct if both the safety and the liveness property hold. Here, safety and liveness are described informally. However, they can be described and proven formally by applying means of temporal logic (as opposed to propositional logic) to a model of a reactive system (as opposed to transformational systems).

The central ***safety*** criterion for the proposed algorithm is that the system always allows access to all model objects at a particular fully functional site. In other words, a client request for accessing model objects is never rejected. However, access may be partly restricted to read access due to system degradation. A violation of this property would be a denial of service at a particular site. Since sites are able to access objects completely autonomously without cooperating with other sites, this is obviously guaranteed.

Along with guaranteed access, the system also guarantees two constraint consistency types observed at all sites:

Strong intra-site constraint consistency is guaranteed at all sites regardless of system degradation. A violation of this property would be intra-site inconsistency. Since intra-site constraints involve only model objects assigned to a particular primary site, write access is possible only at a single site. Since such access is properly isolated, constraint inconsistency cannot occur.

Strong inter-site constraint consistency is only guaranteed in a healthy system before and during system degradation, while inter-site constraint inconsistency may arise only after reunification. A violation of this property would be inter-site inconsistency before or during system degradation. Before degradation, inter-site inconsistency not possible, because sites cooperate synchronously using established atomicity mechanisms. During degradation, inter-site constraints are evaluated locally, similar to intra-site constraints. Therefore, inter-site constraint consistency before and during system degradation is guaranteed.

The central ***liveness*** criterion for the proposed algorithm is the guarantee of eventual replica consistency. Changes of model objects are applied instantly at the primary site and, depending on the system degradation, might not reach other sites for some time, but they are guaranteed to reach every other site eventually. Substantiated by the description in the former sections, this liveness property is satisfied.

Chapter 4

Use Case and Proof of Concept: The DTMS

4.1 Purpose of the DTMS

Throughout this section the "Distributed Telecommunication Management System (DTMS)" software architecture is described, which incorporates the proposed approach. It is an object-oriented, distributed and highly available software for managing a telecommunication network to be used in air traffic control. The DTMS follows a logical client/server structure but is implemented as N-tier fully distributed system by the use of CORBA middleware. Its main features are parametrization, control, and status monitoring of the telecommunication embedded systems. Also, several subsidiary tasks are accomplished, most importantly logging of system events and user activities. Regarding non-functional requirements, high availability and fault-tolerance are of concern within this work and therefore presented in more detail, but also performance requirements were part of the overall product specification.

The architecture adheres to the design principles and requirements of component-based software engineering, especially the encapsulation of coherent functionality while simultaneously separating different functionality. This leads to a modular, component-based system with clear, well-defined interfaces, which in turn improves extensibility (e.g. to incorporate project specific components like different replication mechanisms for different customers), but also enables a flexible composition of different components and therefore enables a flexible combination of system properties. Component names appear using `bold typewriter font`.

4.2 DTMS Overview

The DTMS manages a VCS (Voice Communication System) network, where every VCS is assigned to a DTMS server, as illustrated in figure 4.1. Multiple DTMS servers are connected via a standard IP-based wide-area network and every DTMS server configures, controls and monitors its associated VCS devices. The communication between DTMS and VCS contains proprietary parts and is based on the OSI (Open Systems Interconnect) layer 2 protocol HDLC (High Level Data Link

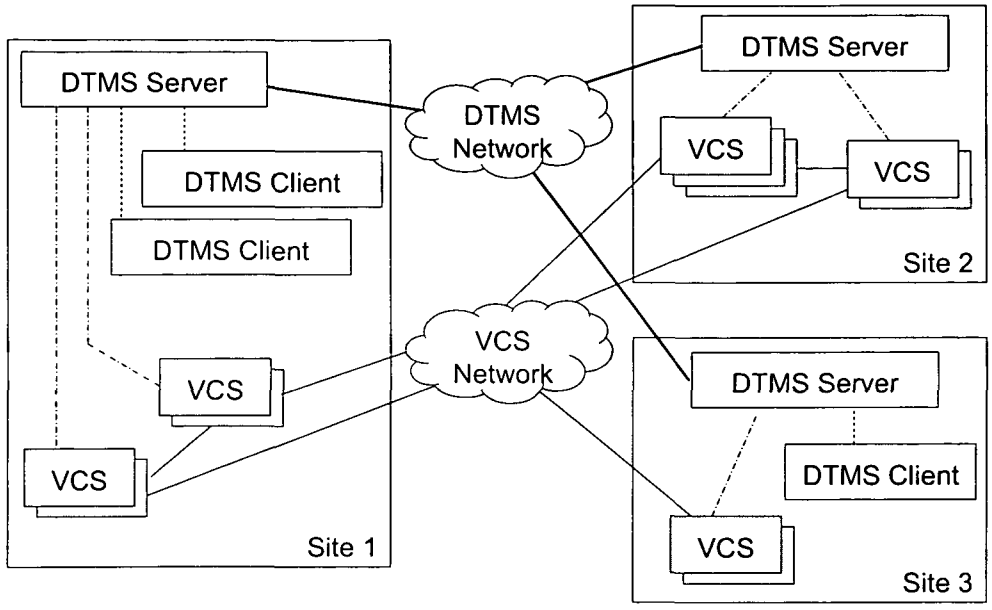

Figure 4.1: DTMS overview.

Control). Every DTMS stores VCS parameter data. DTMS clients and DTMS servers are connected via a standard IP-based local area-network. The VCS devices are connected via a voice communication network, which can be chosen from a wide variety of communication means like the digital ISDN (Integrated Services Digital Network).

The following DTMS requirements are relevant for distribution and persistence:

- Typically, small write and large read transactions need to be performed frequently. This is due to frequent small changes in VCS parametrization and reading of whole parameter sets (all model objects) to generate the necessary binary format for updating the configuration data inside of the VCS hardware.
- Access to the VCS parameter data needs to be fault-tolerant. Site and network failures need to be masked transparently in the following way:
  - Read access: All objects need to be available for read access at any available site at all times despite any system degradation, because parameters from all sites are needed to update the VCS hardware.
  - Write access: For write access the following conditions must hold: Objects need to be available at all sites during periods of healthy system condition. Objects need to be available at a particular site (their primary site) during

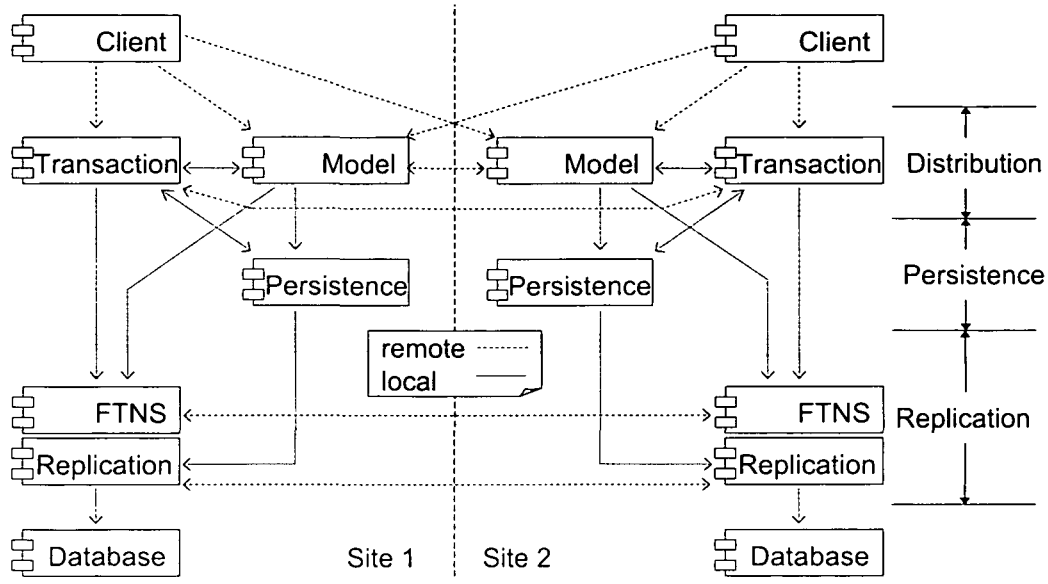

Figure 4.2: Proposed software architecture.

periods of degraded system condition. This is because VCS data needs to be changed at all times, even more so in presence of system failures.

4.3 Components of the DTMS

Figure 4.2 illustrates the proposed software architecture using a UML component diagram (slightly modified, because also deployment aspects are shown). The picture shows how components use each other either locally or remotely. A single site consists of the following components: `Transaction`, `Model`, `Persistence`, `FTNS`, `Replication` and `Database`. `Client` components do not belong to a site. A client can be connected to an arbitrary site. To keep the communication flow between sites organized and comprehensible, the remote communication is established between components of the same type only, for instance, local `Replication` and remote `Replication`. As denoted on the righthand side, the `Model` and `Transaction` components belong to the distribution aspect, the `Persistence` component to the persistence aspect and the `Replication` component to the replication aspect of the architecture. The `FTNS` component is depicted as part of the replication aspect but in fact belongs to the replication *and* distribution aspect of the architecture.

In the following sections, the DTMS components are described in more detail. Some aspects of their intercommunication which are selected according to their relevance for fault-tolerance and replication are explained using sequence diagrams.

4.3.1 Model

The `Model` component contains the model of the VCS and a number of objects, which connect the VCS model to the framework. The model of the VCS makes up

the objects that need to be highly available.

The architecture of the framework and the definition of the interfaces provided by the framework allow the implementation of a VCS model in a way it can be generated completely automatically from its UML specification. Additionally, the "handling" of the model (change and retrieve object data; creation, insertion, and deletion of objects; queries) has to be done in a generic way, in order to reduce the effort for adapting the DTMS for each new VCS: An individual VCS is constructed for each customer depending on his needs. Therefore, the development of the DTMS for this particular VCS mainly encompasses the modelling of a new VCS model. The framework can be seen as the "glue" between different VCS model instances deployed at different sites and the framework services (replication, persistence, distribution). It provides "docking places", where the VCS model can be "plugged in".

The VCS model

- is complex, i.e. it consists of many classes, small object-states and few object-states per class.
- requires complex validation, i.e. many algorithmic constraints involving many object-states.
- requires complex read operations concerning many objects, e.g. assemble information for configuration of the VCS.

Each entity object has a system wide unique identifier. Objects register at component `Transaction` in order to participate in transactions. Objects store their serialized object-state at `Persistence` upon the commit of transactions.

Transactional behavior

Transactional behavior is mandatory for all model objects. Therefore, all model objects implement the `validate()`, `commit()` and `rollback()` method. Upon the invocation of `validate()` the respective object validates its new object-state. Upon the invocation of `commit()` the respective object changes its object-state internally and upon the invocation of `rollback()` the respective object discards its new object-state internally. `validate()` and `commit()/rollback()` are invoked by `Transaction` during the commit phase. All model objects' methods are classified as either read or write method. Read methods

- do not change the object-state and
- do not trigger the registration of the respective object at `Transaction` (this has an impact on serialization, see `Transaction` description in section 4.3.3

Write methods

- do change the object-state,
- do trigger registration of the respective object at `Transaction`, and
- if a model object cannot register at `Transaction`, it refuses to execute the write method.

Persistence of model objects

Persistence is mandatory for all model objects. All model objects implement the `store()` method. Upon invocation of `store()`, the respective object records its object-state at `Persistence` providing its ID, type and serialized object-state.

4.3.2 Client

The `Client` component performs invocations on objects in local or remote `Model`, where the physical location of the model object is transparent to `Client`. An arbitrary number of `Client` components can operate on the model concurrently. The `Client` component starts and finishes transactions at `Transaction`.

4.3.3 Transaction

The `Transaction` component is responsible for serializing concurrent transactions. It coordinates the invocation of `commit()/rollback()` and `validate()` on the objects in the `Model`, it controls the transaction context at `Persistence` and it coordinates distributed transactions in cooperation with remote `Transaction` components.

At the time of writing, the smallest unit of lock granularity is a whole site. Since write transactions are not expected to last long and validation involves many object-states from different classes, this is sufficient. Certainly, many applications require finer lock granularity, which is possible and will be implemented in future versions. Two transaction types are distinguished requiring that the request type is known beforehand:

Write transaction: Upon the start of a write transaction, the site is locked for further operation until the current transaction is finished with commit or rollback. During a write transaction no other transaction can be conducted at this site.

Read transaction: An arbitrary number of clients can perform read transactions concurrently. Read transactions are prevented from interfering with write transaction, thus providing read consistency.

4.3.4 FTNS

The `FTNS` component maintains a list of all sites and and traces their reachability. It provides information about the currently responsible site for a particular object, which is especially important if the primary site of that object cannot be contacted: If a remote site cannot be reached, the site itself is responsible. This mapping from primary site to responsible site is part of the fault-tolerant naming service mechanism. `FTNS` initializes objects accordingly using either remote `Model` (healthy system) or local `Persistence` (degraded system). `FTNS` components provide CORBA Interoperable Object References (IOR, refer to CORBA [1]) pointing at their associated local or remote objects given an entity object ID. `FTNS` components obtain IORs pointing at remote objects by communicating with remote `FTNS` components.

4.3.5 Persistence

The `Persistence` component stores and retrieves serialized object-states using the `Replication` component. Additionally, it provides methods to access object-states in a transactional way, which are used by `Transaction`. `Persistence` does not distinguish between object-states that belong to the local site and object-states that belong to a remote site, but always persists to and retrieves from the local `Replication` component. Factories always ask their local `Persistence` component for retrieving persisted object-states. An object persists itself providing its ID, its type and its object-state to `Persistence`, which in turn provides transactional access to the stable object-store.

4.3.6 Database

The Database component essentially encapsulates the actual stable storage. It provides function primitives to persist and retrieve data. It may also encapsulate local database redundancy for even higher availability.

4.3.7 Replication

The `Replication` component is at the heart of the architecture, as it implements the replication algorithm described in chapter 3. It performs a replication protocol in accordance with `Replication` at other sites to propagate object-states, to provide object-states to other components, and to calculate their object-readiness accordingly. It uses the `Database` component to persist data locally and it provides methods to access the stored object-states.

At the time of writing, object-readiness and sets of constraints are supported implicitly only. That is, clients cannot demand a certain object-readiness and they experience that only a certain object-readiness can be provided depending on the system condition by being refused to write access read-available objects. Also, clients cannot query the current constraint consistency. Instead, they are possibly confronted with inconsistent data. However, system degradation (i.e. disconnection from other sites) is visible to clients, since `FTNS` provides the respective information.

4.4 Typical Sequences of Component Interaction

The following section describes some of the key sequences of component interaction. `S1` and `S2` denote site IDs, `A1` an object ID, where the class is `A` and the ID is `1`. `A1_S1` denotes an object reference pointing at an object, which has class `A`, ID `1`, and is running at site `S1`.

4.4.1 Distributed Object Access in a Healthy System

Scenario: A client wants to execute a write transaction spanning over two sites, `S1` and `S2`. Figure 4.3 (Part 1: Begin and write method invocation) and 4.4 (Part 2:

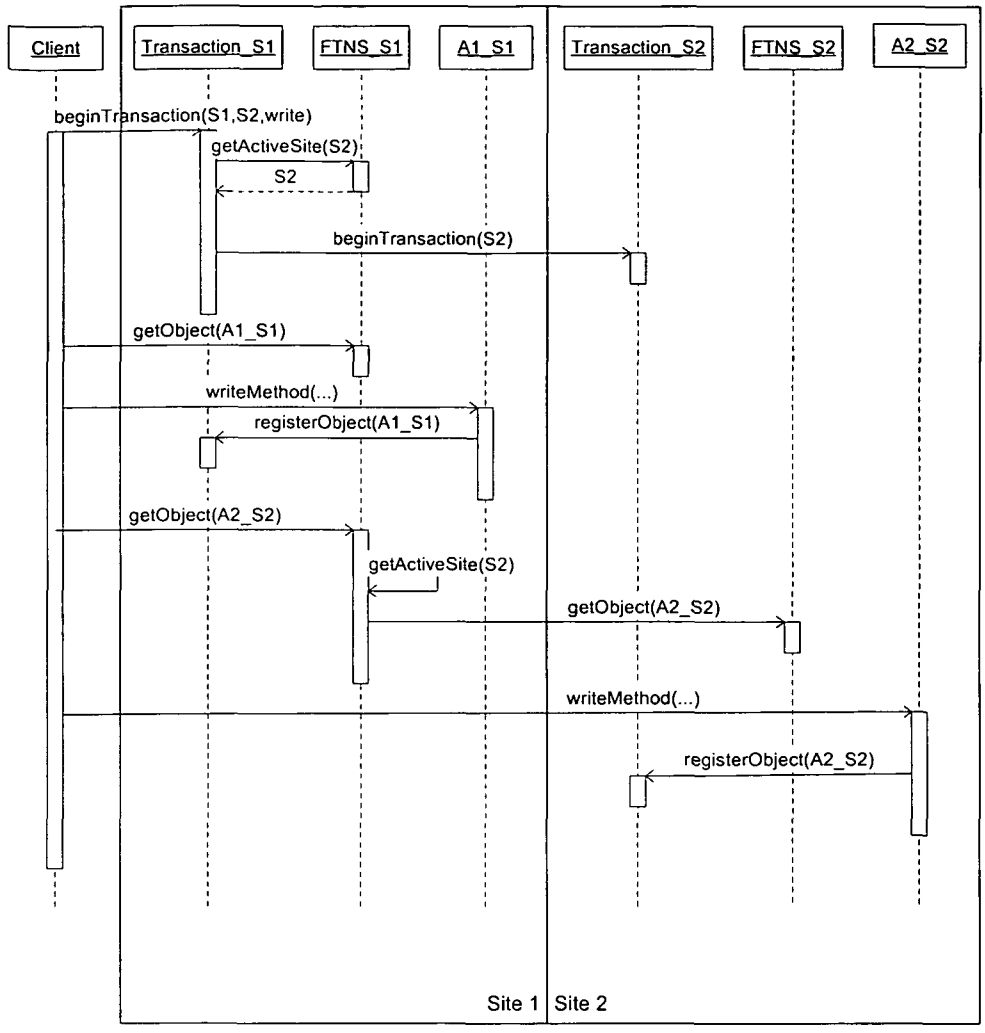

Figure 4.3: Distributed object access in a healthy system. Part 1: Begin and write method invocation.

Commit) belong together and they depict the sequence diagram of the begin, write access, and commit phase of a whole transaction.

- `Transaction_S1` queries `FTNS_S1` for the currently responsible site of primary site S2. In a healthy system the result is S2.
- `Transaction_S1` coordinates the progress of the transaction in cooperation with `Transaction_S2`.
- `FTNS_S1` is inquired for both, `A1_S1` and `A2_S2`. For the remote object `A2_S2`, `FTNS_S1` queries the responsible remote `FTNS_S2`.
- Every participating object (`A1_S1`, `A2_S2`) registers at its local `Transaction`.

4.4.2 Object Access in a Degraded System

Scenario: A client wants to execute a read transaction over two sites, S1 and S2, while S2 is down (figure 4.5). The diagram illustrates how the system is still able to provide the client with the read-available object `A2_S1`.

- Again, `FTNS_S1` determines the responsible site of primary site S2. This time, S2 cannot be contacted, instead S1 is returned. Therefore, `FTNS_S1` invokes `loadObjects(class A, S2)` at `Persistence` to retrieve the respective object-states and creates the objects itself. The object-readiness of the newly created objects is set to read-available.
- The client simply invokes the required read method at object `A2_S1`, similarly to the case without system degradation.

4.4.3 Replication of Transactions

Scenario: `Persistence` uses `Replication` to store object-states. Replication stores object-states locally and builds replica transactions to fill replication queues (figure 4.6).

- `storeReplicaTransaction()` and `store replication entry` are repeated for every remote site. These functions fill the replication queues, which are processes independent from the transaction at a later stage.
- Local object-states and replication queues are committed together. Therefore, all updates to the primary copy will eventually arrive at all remote sites.

4.5 Post Mortem Analysis and Future Work

In object-oriented terminology, a ***post mortem analysis*** is the process of reviewing the design and development of a completed software project with the intention of identifying successes and shortcomings to provide starting points for possible future improvements. Starting with the successes, the following section summarizes such findings of the post mortem analysis of the DTMS design and implementation.

Component coherence: `Persistence`, `Replication`, and `Model` communicate over implementation independent interfaces enabling the flexible exchange of single components.

Fault-tolerance: Clients use the model independently from the system condition as far as object-readiness and sets of constraints permit. Furthermore, the model is set up like a conventional distributed system during periods of a healthy system. Also, the framework is fully distributed and hence not vulnerable to a single failure.

True distribution: All system components exist on all sites locally, equally sharing and coordinating their responsibilities.

Transparency: The client does not use `Persistence` and `Replication`, and it does not care about the physical location of model objects. Model objects do not care about `Replication`, but they have to implement a externalization interface and therefore `Persistence` is not fully transparent to them.

The incorporation of the ideas of object-readiness and sets of constraints presented in the previous chapter was successful as far as describing and assessing the system behavior is concerned. However, the system supports these concepts only implicitly. They are not used to explicitly communicate the system condition. This is a future goal of the DTMS.

In the following, other shortcomings in the DTMS along with planned action items are enumerated:

- A rigor investigation of overall system properties depending on component properties is missing. Especially different algorithms within the replication are of concern, since invocation of write methods at model objects at all sites even during system degradation is a future requirement.

- Better exploitation of objects' operation semantics (now read/write operations) and transaction types (now read/write-available) regarding the trade-off between availability of object-states and constraint consistency requirements is desired. However, even though there is a considerable amount of theory on the subject of exploiting method semantics in order to increase availability of methods, implementation of such concepts is not common. Apart from a increase of implementation complexity (consider a replication mechanism, which knows about method semantics of every object type and calculates the objects' availability accordingly) it has the potential of rendering the system incomprehensible from the users' point of view.

  Alternatively, it is under investigation if the model component can be reduced to a cache and transfer some of its functionality to the underlying database. Especially transaction isolation is of interest for reasons of performance and locking granularity, but also constraint consistency can be provided efficiently. However, it has to be investigated if and how the complex and algorithmic constraints of the DTMS model can be implemented using database functionality only. Furthermore, means and suitability of controlled constraint violation have to be analyzed in order to facilitate the central requirement of increasing replica availability for constraint consistency.

- Support for finer granularity of locking to enable better concurrency of model access for clients. Currently, a site is the unit of lock granularity. Since this is clearly not sufficient, it is envisaged to use object granularity here as well. This entails the ability of each object to associate method invocations with a particular transaction context and to calculate whether a commit is possible or not. Additionally, more complexity is introduced because of "indirect" write operations due to object relationships and because of the necessary locking

of "uninvolved" objects that are used during validation of constraints. It is planned to generate the necessary code.

- Support for a query engine, that allows descriptive queries using the Object Data Management Group's OQL (Object Query Language). Currently, two simple types of query are supported: Get object by ID and get all objects of a certain type. For efficient user interface implementation purposes, additional queries for navigation through the object hierarchy are essential, e.g. get the ancestor or get the descendants of an object where additional conditions on attributes are fulfilled.

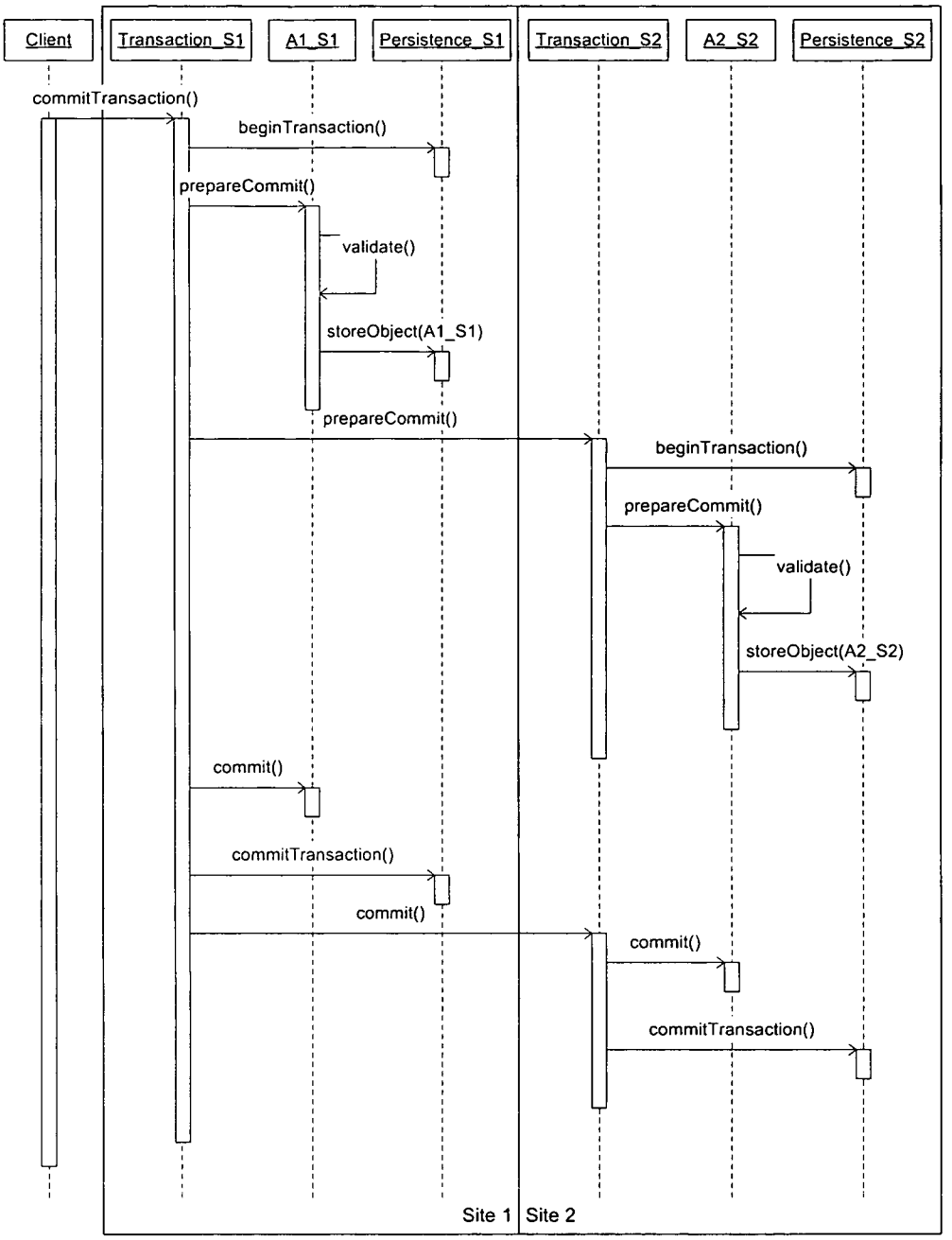

Figure 4.4: Distributed object access in a healthy system. Part 2: Commit.

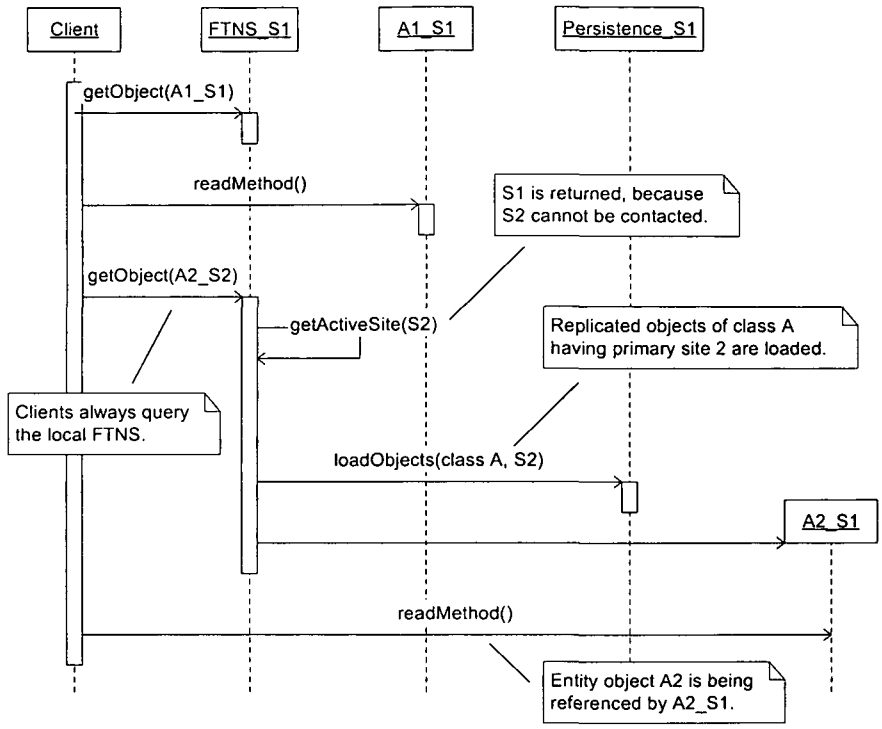

Figure 4.5: Object access in a degraded system.

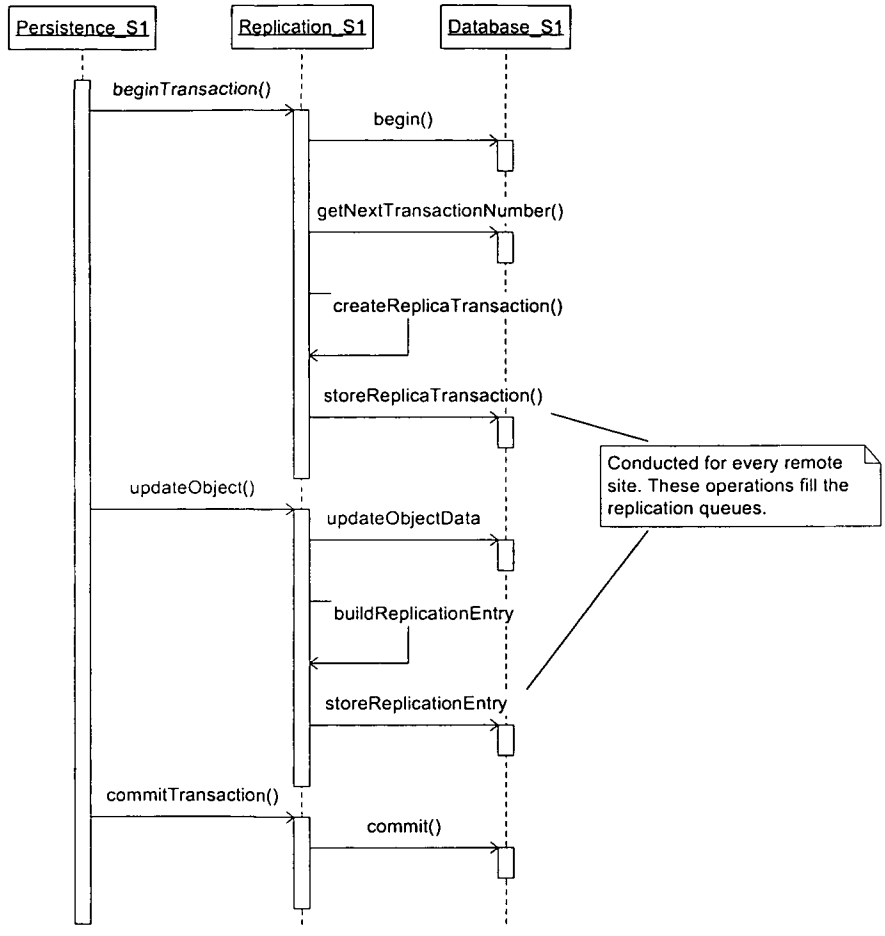

Figure 4.6: Replication of transactions.

Chapter 5

Summary and Conclusion

5.1 Summary

The aim of this thesis was to introduce the terminology, an architectural concept, and a system model to facilitate the trading of availability for constraint consistency of replicated objects in a distributed system.

Three different types of consistency have been analyzed, defined, and compared:

- ***Replica consistency***, which defines the correctness of replicated data or, informally speaking, how replicas of the same logical object may differ from each other.
- ***Concurrency consistency***, which defines the correctness of concurrent access to a single set of replicas or, informally speaking, how concurrent client operations may interfere with each other.
- ***Constraint consistency***, which defines the correctness of the system state with respect to a set of data integrity constraints.

These definitions paved the way for describing the trade-off between availability as one of the most significant attributes of dependability with respect to fault-tolerance and constraint consistency: Availability can be increased if a controlled violation of constraint consistency is acceptable. This trade-off was the ***key focus*** of this work.

As the ***key architectural concept*** for incorporation of this trade-off into a distributed system, a two-layer architecture was introduced: Persistent object-states are replicated asynchronously using a primary copy ROWA algorithm, while distributed transactions are carried out in a synchronous manner. During normal operation the system is set up like a conventional distributed system, while propagating persistent object-states prepares for degraded scenarios.

Within this architecture, a ***key system model*** for tuning the trade-off was introduced: The fault-tolerant naming service (FTNS). It is used to map the object identity possessed by every entity object to an actual object reference. Since this

mapping depends on the fault-tolerance strategy, the mode of operation of the fault-tolerant naming service was aligned with the underlying primary copy ROWA.

As proof of concept, implementation details of the DTMS were presented. Sequence diagrams were used to show how the FTNS component of the DTMS is able to provide a client with operable objects even in the presence of system failures.

5.2 Related Work

5.2.1 General Related Work for High Availability

Clustering is a common solution for high availability, usually facilitating a clustered singleton service, a clustered notification service, and a clustered scheduler service. A number of servers support this type of setup, called server farms (e.g. the open-source Enterprise JavaBeans server JBoss). However, such systems are usually deployed for load-balancing, where it is sufficient to perform the request on a single instance and changes need not be reflected at other nodes. Even if changes are propagated, server farms usually operate in a local area network, where the failure of network partitions is extremely unlikely or even impossible. On the other hand, providing fault-tolerance among topologically dislocated application servers requires to handle concurrent access to different servers. This is a major difference between clustering and the proposed approach in this thesis, since the problem of accessing objects in different network partitions is explicitly addressed. A citation in the fault-tolerant CORBA specification, which is the OMG's (Object Management Group) answer to the need for fault-tolerance, illustrates the need for this functionality:

> *Network partitioning faults separate the hosts of the system into two or more sets, the hosts of each set being able to operate and to communicate within that set but not with hosts of different sets. The current state-of-the-art does not provide an adequate solution to network partitioning faults. Thus, network partitioning faults are not addressed in this specification.*

5.2.2 Related Work about Trading Consistency

All of the mentioned types of consistency mutually depend on the availability of objects for method invocation and the performance of such invocations and can therefore be traded for each other. This is scientifically accepted and regarding the trading between availability/performance and replica/concurrency consistency, a large body of mature research exists.

Some of this research is dealing with replica consistency, for instance: [14, 38, 53, 39, 22, 13, 42, 44, 25, 34]. Among the mature research of the trade-off between availability and replica consistency is the TACT (Tunable Availability and Consistency Tradeoffs) project [54]. So-called consistency units (conit) facilitate a model of continuous consistency, which is based on a set of three metrics: numerical error, order error, and staleness:

Numerical error limits the total weight of writes on a conit that can be applied globally across all replicas before being propagated to a given local replica. Order error limits the number of tentative writes on a conit (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas.

The following enumeration gives examples of other implementations in the field of dependable distributed systems with various replication consistency properties, which either support replication or provide fault-tolerance through replication:

- The famous ***ISIS system*** [9] was one of the first systems to handle fault-tolerance above the operating system layer[1]. As ISIS is a popular system, it is discussed here more at length.

  The key idea in ISIS is to use the concept of *process groups* and according tools for *group programming*. Process groups are distributed groups of cooperating programs. They facilitate fault-tolerance by transparent adaption to failures and recoveries. They provide certain properties to enable reliable operation of applications:

  - Group addressing: Group names are used to address and send messages to the group instead of individual members. Such messages are called *multicast* messages.
  - Message delivery ordering: Apart from overcoming message loss and duplications (messages should be delivered exactly once), which is done at lower system layers in ISIS, causal dependency [30] of messages has to be maintained. In ISIS, this is achieved using two multicast communication primitives, ABCAST and CBCAST. ABCAST has the property of atomic delivery ordering, meaning that messages are guaranteed to be delivered in identical order at all processes. However, since ABCAST is costly to implement and can also involve rather high latency, the "weaker" CBCAST is supported. Here, identical message ordering is only guaranteed for messages which are causally dependent. CBCAST guarantees that only conflicting multicasts are seen by all recipients in the order of causal dependency, other multicasts are delivered in any order. CBCAST is said to be *virtually synchronous*, since the outcome of the execution is the same as if atomic delivery had been used. Obviously, the ability to use CBCAST is highly dependent on the nature of the application.
  - Failure atomicity: ISIS uses a protocol to propagate messages that enables "exactly-once delivery" of each message to those destinations that remain operational in failure scenarios. Since participating processes are assumed to be fail-stop, delivery to failed processes does not have to be of concern. ISIS implements the fail-stop model using an agreement protocol to maintain a system membership list: only processes in this list are permitted to participate.

[1] However, the authors state [10] that "kernelizing" some parts of ISIS can reduce the resulting performance penalty considerably.

- Use of group membership as input: Often, group members need consistent views of group membership to perform tasks. For instance, this is used for fault-tolerant services that need a primary member. If the primary member fails, backups take over in some consecutive order. Unless every group member sees the same group changes in the same order, undesirable situations could arise (no primary, several primaries). ISIS uses the idea of looking at group membership as shared data, which can be "locked" to prevent interference with multicast messages. However, since locking is costly, ISIS replicates group membership data among the members of the group itself. This way, the system takes care that multicast messages do not interleave with changes of the group membership.

ISIS is a proven and popular system and a milestone in the area of reliable distributed computing. However, there are also limitations and some of them are differences to the approach presented in this work:

- Strong replica consistency: ISIS aims at strong replica consistency. Replicas may fail and be dropped from a group and rejoin later and update their state, but cannot operate independently from the group and accept replica inconsistency to provide better availability.
- Reduced availability during network partition failures: ISIS only allows progress in a single majority partition, resuming normal operation only when normal communication is restored.
- The approach presented in this work aims at transactional serializability, whereas ISIS supports the model of virtual synchrony, as quoted from [9]:

  > *The relationship between ISIS and transactional systems originates in the fact that both virtual synchrony and transactional serializability are order-based execution models. However, where the "tools" offered by a database system focus on isolation of concurrent transactions from one another, persistent data and rollback (abort) mechanisms, those offered in ISIS are concerned with direct cooperation between members of groups, failure handling, and ensuring that a system can dynamically reconfigure itself to make forward progress when partial failures occur. Persistency of data is a big issue in database systems, but much less so in ISIS.*

- The ***Horus system*** [52] is a successor of the ISIS system and it provides a general purpose group communication model to application developers. It supports the virtually synchronous execution model and a runtime configurable structured framework for protocol composition. The group communication architecture can be used to introduce reliability or replication transparently.
- The more experimental framework ***Ensemble*** distributed communication system, which is the next generation of the Horus toolkit. It supports the virtually synchronous execution model and a runtime configurable structured framework for protocol composition [8].

- The mature ***Java Messaging Service*** (JMS) by Sun Microsystems. Quote from the documentation: *The Java Message Service (JMS) API is a messaging standard that allows application components [...] to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.*

- The ***Eternal System*** [39], which enhances CORBA using interceptions to provide strong fault-tolerance transparently. No modifications to the Object Request Broker, the operating system, or the application are necessary. In addition, non-deterministic multi-threading of objects is possible. Results from this project contributed significantly to the Fault-tolerant CORBA specification.

- ***Fault-tolerant CORBA*** specification by the OMG [1]. FT CORBA requires deterministic behavior of objects and provides either strong infrastructure-controlled or application-controlled replica consistency through the use of object groups.

- ***Phoenix/APP*** is a Microsoft research project to enhance Microsoft's middleware .NET to support transparent recovery of COM+ components.

- ***Globe*** (http://www.cs.vu.nl/globe/) is a middleware platform to build wide area distributed applications. Again, implementing a particular strategy of replication for fault-tolerance is left to the object provider.

- ***Fleet*** [38] is a middleware system implementing a distributed repository for persistent Java objects. Fleet differs in two issues from the approach presented in this thesis: At first, a quorum consensus based replication strategy is deployed to implement a strong consistency model. Using that technique, consistency can be guaranteed at all times at the price of reduced availability. Secondly, fleet uses object groups to enhance performance at the price of increased complexity. In fact, the Fleet system does not support transactions over multiple objects. Generally, transactions based on group communication primitives are still to be researched [44] and not supported.

Other and older research mainly in the field of databases is dealing with concurrency consistency and how to improve availability by using weaker models than serializability: [26, 6, 15]. Also, some research looks at replica/concurrency consistency in an integrated way [51, 49].

All of the above mentioned research is done under the presumption of ***strong constraint consistency*** or by ignoring constraint consistency at all. However, the ***very specific aspect of partially sacrificing constraint consistency*** in a general way to achieve higher availability is very poorly researched. It embodies a strong potential for improvement of a whole class of applications, that is, such applications where availability is more critical than transient constraint inconsistency. As explained, this trade-off cannot be explored isolated from replica/concurrency consistency.

Regarding other research on ***fault-tolerant naming and respective services***, the following articles deal with general concepts:

[35], for instance, introduces a replicated naming service using primary backup replication:

> *This paper describes the design and implementation of a fault-tolerant CORBA naming service - CosNamingFT. [...] The name service [...] is a critical gateway to all objects in a distributed system; to avoid having a single point of failure, the name service should be made fault-tolerant. CosNamingFT uses the GroupPac package, a CORBA-compliant suite of protocols, to replicate the name server. GroupPac services are built from Common Object Services that function as building blocks to implement fault-tolerant applications.*

[37] also presents a replicated naming service that adheres to the COS specification:

> *High availability of the naming service is important since most CORBA applications need to access it at least once during their lifetime. Unfortunately, the OMG standards do not deal with availability issues; the naming services of many of the commercially available CORBA object request brokers introduce single points of failure. [...] Our naming service can be replicated at run-time, while many applications are installing and retrieving object references.*

However, all of the mentioned work is related but significantly different, since the proposed FTNS does not support a highly-available naming database, but instead supports a mapping from object identity to reference, which is potentially different on every node because it depends on the view of the current failure scenario at a particular node and the replication algorithm. The FTNS is not an isolated concept, but rather a model, which embodies the means of trading constraint consistency for availability.

5.3 Future Work

Trade-off Availability and Consistency In order to utilize the potential of the trading of availability for constraint consistency in a systematic, application-independent, and practical way, the trade-off has to be formalized. This formalization has to consider object-typed entities to enable the seamless integration within standard object-oriented software engineering techniques.

Also, an according syntax or diagrams for denoting the trade-off with respect to particular classes is needed. Classes, which are to be deployed in the system, must be described using this syntax. Every class has to be enhanced with such a configurable profile, which allows the deployer to assign an application-independent, but deployment-specific behavior. Ideally, the syntax should also include ways of assigning particular reunification strategies.

Metrics In order to be able to assess how efficiently a system can trade availability for consistency and the impact on performance and other non-functional requirements, metrics are needed. Standard metrics in distributed systems exist for availability (probability of being operational etc.) and performance (round-trip time, fail-over time, communication overhead etc.) and are evaluated to exhibit particular advantages and shortcomings of distributed algorithms and implementations. However, to be able to assess the quality of results of future improvements of the trading with constraint consistency, standard metrics for constraint consistency are needed here as well. As a major requirement it has to be possible to evaluate the newly defined metrics independent from a particular approach and implementation. Furthermore, the overall system consistency has to be taken into account, since different sets of replica can possibly possess different constraint consistency simultaneously.

Hybrid Replication Protocols Hybrid replication protocols are designed to combine the advantages of both synchronous and asynchronous replication models. Investigation in how such protocols can aid to the system architecture have not been conducted yet but are of interest.

List of Figures

Bibliography

[1] *http://www.omg.org/*. The Object Management Group.

[2] *http://www.sun.com/*. Sun Microsystems.

[3] *http://www.w3c.org/*. World Wide Web Consortium.

[4] B. Alpern and F.B. Schneider. Defining liveness. *Elsevier Information Processing Letters*, 21(4):181–185, October 1985.

[5] H. Attiya and J. Welch. *Distributed Computing - Fundamentals, Simulations, and Advanced Topics.* Wiley, Hoboken, New Jersey, 2004.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM, May 1995.

[7] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[8] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The horus and ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00)*, January 2000.

[9] K.P. Birman. The process group approach to reliable distributed computing. *Communication of ACM*, 36(12):37–53, December 1993.

[10] K.P. Birman and R. Cooper. The isis project: real experience with a fault tolerant programming system. *ACM SIGOPS Operating Systems Review*, 25(2):103–107, April 1991.

[11] European Commission. *European transport policy for 2010: time to decide.* Office for official publications of the European communities, Luxembourg, 2001.

[12] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Designs.* Addison-Wesley, 2001.

[13] P. Felber, B. Garbinato, and R. Guerraoui. The design of a corba group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems.* IEEE, October 1996.

[14] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of corba applications. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2002)*, October 2002.

[15] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[16] K.M. Goeschka, M. Jandl, R. Smeikal, and A. Szep (Authors in alphabetical order). Dependable distributed systems. European union framework programm 6 project proposal, strategic objective 2.3.2.3 open development platforms for software and services, European Union, February 2004.

[17] K.M. Goeschka, H. Reis, and R. Smeikal. Xml-based client-server communication. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36)*. IEEE, January 2003.

[18] K.M. Goeschka and R. Smeikal. Using replication for high availability of a distributed management system. Tech:news, Frequentis GmbH, February 2003.

[19] J. Gray and L. Lamport. Consensus on transaction commit. *Microsoft Technical Report MSR-TR-2003-96*, January 2004.

[20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, California 94104, 1993.

[21] F.C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[22] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA 1998)*, pages 244–258. ACM, October 1998.

[23] G.T. Heineman and W.T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[24] A.A. Helal, A.A. Heddaya, and B.B. Bhargava. *Replication Techniques in Distributed Systems.* Kluwer Academic Publishers, Boston/London/Dordrecht, 1995.

[25] M. Herlihy. A quorum consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[26] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[27] P. Jalote. *Fault-tolerance in Distributed Systems.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1998.

[28] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Boston/London/Dordrecht, 1997.

[29] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[30] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communication of ACM*, 21(7):58–65, July 1978.

[31] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[32] J.C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer Verlag, Vienna, 1992.

[33] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of ECOOP 1999*, 1999.

[34] X.S. Liu, A.S. Helal, and W. Du. Multiview access protocols for large-scale replication. *ACM Transactions on Database Systems*, 23(2):158–198, June 1998.

[35] L. C. Lung, J. S. Fraga, J.-M. Farines, M. Ogg, and A. Ricciardi. Cosnamingft – a fault-tolerant corba naming service. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems SRDS99*. IEEE, October 1999.

[36] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[37] S. Maffeis. A fault-tolerant corba name server. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems SRDS96*. IEEE, October 1996.

[38] D. Malkhi, M.K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2001.

[39] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The eternal system: An architecture for enterprise applications. In *Proceedings of the International Enterprise Distributed Object Computing Conference EDOC 1999*, pages 214–222, September 1999.

[40] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1979.

[41] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Boston/London/Dordrecht, 1996.

[42] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386. ACM, April 1991.

[43] S.B.Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):342–370, September 1985.

[44] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communication of ACM*, 39(4):84–87, April 1996.

[45] R. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1982.

[46] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, April 1990.

[47] R. Smeikal and K.M. Goeschka. Fault-tolerance in a distributed management system: a case study. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, page 478. IEEE/ACM, May 2003.

[48] R. Smeikal and K.M. Goeschka. Fault-tolerant distribution and persistence of objects using replication. In *Poster Session at the 23rd IEEE International Conference on Distributed Computing Systems 2003*. IEEE, May 2003.

[49] M. Stonebraker and E. Neuhold. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 3(3):188–194, May 1979.

[50] A.S. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey 07458, 2002.

[51] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions of Database Systems*, 4(2):180–209, June 1979.

[52] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: a flexible group communication system. *Communication of ACM*, 39(4):76–83, April 1996.

[53] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 264–274. IEEE, April 2000.

[54] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, August 2002.

Curriculum Vitae

Robert Smeikal
Vienna University of Technology
Gußhausstraße 27-29
A-1040 Wien, Austria
Email: smeikal@acm.org

August 19^{th}, 1974	Born in Vienna, Austria.
09/1980 – 06/1984	Primary School.
09/1984 – 06/1993	High School, A-Level.
1998 – 1999	Tutor for software engineering at the Vienna University of Technology.
2000	Award for accomplishing an extraordinary student project, received from the faculty of electrical engineering, Vienna University of Technology.
10/2000	Diplom-Ingenieur (M.Sc.) in electrical engineering, with distinction, Vienna University of Technology.
10/2000 – present	Research Assistant at Vienna University of Technology, Institute of Computer Technology, Austria. *Main research areas*: Multi-device applications, web engineering with focus on XML-based solutions and replication in distributed systems. *Lecturing*: Graduate course on microcomputer and software engineering for electrical engineers. *Industry projects*: Technical consultant for "Frequentis GmbH" in the field of network management for air traffic control systems. Technical consultant for "Siemens AG Österreich" in the field of multi-device applications.