Internet-Wide Caching of Distributed Objects

Roman Vitenberg
Internet-Wide Caching of Distributed Objects

Research Thesis

Submitted In Partial Fulfillment Of The Requirements
For The Degree Of Doctor Of Philosophy

Roman Vitenberg

Submitted To The Senate Of
The Technion - Israel Institute Of Technology

Tishrey, 5763 Haifa October, 2002
First, I wish to express my deepest gratitude to my advisor Dr. Roy Friedman for his patient guidance and valuable suggestions, for his friendly attitude and warm support, and for his ability to always find time to listen to my (not always brilliant) ideas. The way you care about your students deserves the highest appreciation.

I am also deeply grateful to Prof. Hagit Attiya and Prof. Assaf Shuster for their wise comments and insights that helped me look more deeply at the subject of my research. Thanks go to Prof. Yair Amir, Dr. Miguel Castro, and Prof. Maarten van Steen for those short meetings that helped me look at various issues of a caching service from different angles and contributed to forming the right view on building complex distributed systems. I am thankful to Prof. Michel Raynal for fruitful discussions about locality of consistency conditions.

While designing and developing the caching service for distributed objects, I enjoyed numerous discussions with Grisha Chockler who was always able to come up with interesting and useful ideas. I would particularly like to thank Grisha for his active participation in the design of the hierarchy construction algorithm. Special thanks go to Hilla Atzmohn who contributed to the caching service development in many ways. Itai Frenkel, Kobi Hadash and Igor Zarivach provided valuable feedback and suggestions while working on porting the distributed Scorched Earth game to Cascade. I would also like to thank all other students who contributed to the project by building applications that employed the service and providing important end user feedback.

Adnan Agbaria and Alon Kama read some parts of this thesis and provided helpful comments. Sharing an office with Adnan was part of what made my years in the Technion such an enjoyable experience.

The following people kindly allowed us to use the computing resources of their labs for testing Cascade in the Internet: Francisco Brasiliero, Danny Dolev, Alan Fekete, Yuh-Jzer Joung, Idit Keidar, Keith Marzullo, Luis Rodrigues, and Maarten van Steen.

Last but not least, I appreciate the love of my wife Lilia who has been so understanding and patient during those long evening hours I spent at Technion.

THE GENEROUS FINANCIAL HELP OF THE TECHNION, INTEL CORPORATION, VATAT SCHOLARSHIP, AND WOLF FUND IS GRATEFULLY ACKNOWLEDGED
# Contents

**Abstract**  

List of Symbols and Abbreviations  

1 Introduction  

1.1 General Background  

1.2 Results of This Work  

1.2.1 Introducing CASCADE  

1.2.2 Applications  

1.2.3 Communication Infrastructure  

1.2.4 Consistency Maintenance  

1.2.5 Cache Management  

1.2.6 Mobile Code  

1.2.7 Performance and Flow Control  

1.3 Road-map  

2 Related Work  

2.1 Caching in Various Software Systems  

2.1.1 Object Caching in CORBA Compliant Systems  

2.1.2 Web Caching  

2.1.3 Caching in File Systems  

2.1.4 Caching and Replication in Object-Oriented Database Systems (OODS)  

2.2 Individual Aspects of a Caching Service  

2.2.1 The Service Approach to Caching and Migration  

2.2.2 Communication Infrastructure and Dissemination Mechanisms  

2.2.3 Consistency in Shared Memory and Distributed Systems  

2.2.4 Communication Analysis in Wide Area Networks  

3 Introducing a Caching Service for Distributed Objects  

3.1 The Model of the CORBA Middleware  

3.2 General Design Goals  

3.2.1 A General Purpose Service  

3.2.2 Service Transparency  

3.3 Compliance with the CORBA Services Standard  

3.4 Choosing the Programming Framework  

3.5 CASCADE System Overview  

3.6 CASCADE System Modules  

3.6.1 The Client Structure  

3.6.2 The DCS Structure  

3.6.3 The Proxy Object Structure  

4 Consistency Model for Wide-Area Distributed Services  

4.1 System Model  

4.2 Consistency Conditions for Objects of General Type  

4.2.1 Refining Object Type System  

4.2.2 Definitions of Consistency Conditions  

5 Conclusion
Contents (Cont.)

4.2.3 On the Relative Strength of Consistency Conditions ................. 29
4.2.4 Local Consistency Conditions ...................................... 31
4.3 Weak Consistency Conditions ............................................ 33
4.3.1 Serialization Set and Definition of a Weak Consistency Condition ... 33
4.3.2 Locality of Weak Consistency Conditions ............................ 34
4.4 Composable Consistency Conditions for Wide-Area Distributed Services 35
4.4.1 Composability Framework ............................................. 35
4.4.2 Proposed Set of Elementary Consistency Conditions ................. 36
4.4.3 Examples of Useful Compositions .................................... 37

5 Implementation and Algorithms ........................................... 41
5.1 Hierarchy Construction ................................................... 41
5.2 Failure Model and Persistence ............................................ 44
5.3 Implementation of Cached Copies Consistency ........................... 46
5.3.1 Assumptions on the Application ...................................... 46
5.3.2 Implementation of Eventual Propagation and Total Order ............ 47
5.3.3 Implementing Session Guarantees in Presence of Total Order .......... 48
5.3.4 Implementing Session Guarantees Without Total Order ............... 50
5.3.5 Object Group-Based Consistency ...................................... 53
5.4 Support for Atomic Operations and Locking ............................ 54
5.5 Overview of Cache Management in CASCADE ............................ 54
5.5.1 Calculation of Object’s Size ......................................... 55
5.6 Mobile Code ............................................................... 56
5.6.1 Code Fetching and Loading ......................................... 56
5.6.2 Code Evacuation ....................................................... 63
5.7 Flow Control .............................................................. 64
5.7.1 Discussion of TCP Performance ...................................... 64
5.7.2 Flow Control Implementation in CASCADE .......................... 66
5.8 Concurrency and Queues Management ................................... 70

6 Applications .............................................................. 73
6.1 Yellow Pages Service ..................................................... 73
6.2 Distributed Bulletin Board ............................................... 74
6.3 Tickets Reservation Service .............................................. 74
6.4 Distributed Scorched Earth Game ....................................... 74
6.5 Caching Multimedia Objects With CASCADE .......................... 76
6.6 Utilizing CASCADE for Efficient Implementation of Other CORBA Services and Facilities .............................................. 78

7 Performance Measurements ............................................... 79
7.1 Running CASCADE in a Wide Area Network ............................ 79
7.1.1 The Testing Environment ............................................. 79
7.1.2 The Experimental Hierarchy ......................................... 81
7.1.3 Invocation Time Measurements ...................................... 82
7.1.4 Assessing Network Level Properties of the Communication .......... 93
7.2 Running Simulations in a Local Area Network .......................... 101
Contents (Cont.)

8 Conclusions and Future Work .................................................. 102
  8.1 Summary of The Results .................................................... 102
  8.2 CORBA Lessons Learned ................................................... 103
  8.3 Future Research ............................................................ 105
    8.3.1 Dynamic Hierarchy Reorganization ................................ 105
    8.3.2 Support for Security ................................................. 105
    8.3.3 Further Research on Consistency Conditions .................... 107
    8.3.4 Resource Management ............................................... 108
    8.3.5 Interrelated Objects ................................................ 108
    8.3.6 Porting to Wireless Ad-Hoc Networks ............................. 109

A CASCADE Core API ............................................................ 118
  A.1 Registering with CASCADE .............................................. 118
  A.2 Working with Cached Objects .......................................... 118

B Caching Service IDL ........................................................... 120
List of Figures

3.1 The CORBA Architecture ........................................ 17
3.2 The Caching Service Architecture .......................... 20
3.3 The Client Structure .......................................... 22
3.4 The CASCADE System Modules ............................ 23
4.1 Various orders and serializations for the given history $H$ .......................... 30
5.1 Joining the hierarchy ........................................ 42
5.2 A structure required for efficient computation of version vector differences .......... 53
5.3 Code class graph ............................................. 58
5.4 The algorithm computing a list of classes to transfer .................................. 60
5.5 Connection management in CASCADE .......................... 68
5.6 Connectors, queues and thread management in CASCADE .......................... 71

6.1 An improvement in the performance of the Distributed Scorch game .................. 76
7.1 The experimental hierarchy ..................................... 81
7.2 Contribution to response time .................................. 84
7.3 Sample invocation time changes over time ........................ 86
7.4 Invocation time histogram, 100b requests, MIT to Amsterdam .................. 87
7.5 Invocation time histogram for the Brasil to Amsterdam link .................. 88
7.6 Effect of request length and write percentage on queries .......................... 89
7.7 Response times for long update requests .................................. 90
7.8 Effect of the hierarchy depth ..................................... 90
7.9 Impact of the Nagle algorithm for a non-lossy link (MIT to AM) ................. 91
7.10 Impact of the Nagle algorithm for a lossy link (BR to AM) ..................... 91
7.11 Average invocation time as a function of update request size .................. 92
7.12 Average invocation time for deeper hierarchies ................................... 92
7.13 A TCP segment size graph ..................................... 94
7.14 A time sequence graph for the centralized object .......................... 95
7.15 A time sequence graph for the centralized object (a closeup) .......... 95
7.16 A time sequence graph for CASCADE communication .................. 96
7.17 A time sequence graph for CASCADE communication (a closeup) .......... 96
7.18 An outstanding data graph for the centralized object .......................... 98
7.19 An outstanding data graph for CASCADE communication .................. 98
7.20 A throughput graph for the centralized object .......................... 99
7.21 A throughput graph for CASCADE communication when Nagle is disabled ........ 99
7.22 A throughput graph for CASCADE communication when Nagle is enabled .... 100
List of Tables

5.1 The implementation cost of session guarantees . . 52
7.1 Round trip times and loss percentage of the links .................. 80
Abstract

Caching systems and architectures have proven exceptionally effective in the large scale World Wide Web environment, and Web caching techniques, tools and infrastructures have been becoming increasingly mature during the recent decade. At the same time, the more general problem of caching generic distributed objects remains mainly unresolved. In particular, very few caching systems have been developed for object oriented platforms like .NET, CORBA or EJB. This is believed to be one of the major reasons for hindering development of truly large scale distributed applications based on such middlewares.

In this thesis we present CASCADE, a caching service for distributed CORBA objects, aimed at increasing the availability, locality, predictability and responsiveness of object accesses. CASCADE is unique in supporting dynamic transfer and deployment of active objects (that include both data and code), and by employing a dynamically built hierarchy, which makes it suitable for scalable Internet services. Among the other main benefits of CASCADE are transparency for applications and flexible consistency maintenance.

We deliberate on the design goals and the general architecture of CASCADE. In particular, we develop a theoretical consistency model that lends itself to distributed services operating in wide area networks. Specifically, we propose a highly flexible composability approach to consistency in such services. We also consider properties of consistency conditions that are important for service scalability, e.g., locality.

We describe the implementation of various CASCADE's mechanisms and protocols in detail. In particular, we present the communication infrastructure maintenance, support of different consistency guarantees, cache management, code caching, and flow control algorithms.

In order to show how applications can benefit from a caching service like CASCADE, we consider several applications of different classes. We analyze the requirements of these applications from a caching service as well as the difficulties of porting already existing applications to CASCADE. In some cases, we indicate the speedup in performance that is achieved from such porting.

Finally, we present the performance of CASCADE as observed in the experiments that we conducted in various settings. Specifically, we show the results of running CASCADE in a real wide area environment including both service response times and network level characteristics of the communication such as throughput. Based on these results, we discuss flow control techniques for reducing response times of distributed services that use TCP for communicating over WANs.
List of Symbols and Abbreviations

\[ pr(op) \] The process at which an operation \( op \) is performed
\[ obj(op) \] The object on which an operation \( op \) is performed
\[ name(op) \] The name of operation \( op \)
\[ inval(e) \] The input value of an invocation event
\[ oval(e) \] The output value of an invocation event
\[ lin \] Linearizability consistency order generator
\[ proc \] PRAM consistency order generator
\[ obj \] Object consistency order generator
\[ norm \] Normality consistency order generator
\[ \sim_H \] The consistency order based on consistency order generator \( G \) and history \( H \)
\[ <_H \] The linearizability consistency order based on history \( H \)
\[ \overset{H}{\rightarrow} \] The linearizability consistency order based on history \( H \)
\[ CC(G) \] Consistency condition based on consistency order generator \( G \)
\[ WCC(G, TO) \] Weak consistency condition based on consistency order generator \( G \) and total order constraint \( TO \)
\[ H|p \] Subhistory of process \( p \)
\[ H|X \] Subhistory of object \( X \)
\[ H|w \] The subsequence of all events in \( H \) which are invocations or responses of write operations
\[ H|p + w \] The subsequence of all events in \( H \) which are either in \( H|p \) or in \( H|w \)
\[ \rightarrow^* \] The causality relation
\[ DCS \] Domain caching server
Chapter 1

Introduction

1.1 General Background

One of the main goals of modern middlewares, such as CORBA [93] and .NET [101], is to facilitate the design of interoperable, extensible, and portable distributed systems. This is done by standardizing a programming language-independent Interface Definition Language (IDL), a large set of useful services, a generic inter Object Request Broker (ORB) communication protocol (GIOP/IIOP in the case of CORBA and SOAP in the case of .NET), and bridges to other common middlewares. Such middlewares, combined with the global connectivity of the Internet, create a potential for truly global services that are available for clients anywhere in the world.

However, the long and unpredictable latencies of the Internet as well as its unreliability complicate the realization of this potential. This thesis was inspired by a simple test using the UNIX ping program to measure the Internet delays. The results of this test demonstrate the variance of latencies incurred by the Internet: Pinging a local host in the same LAN takes less than one millisecond; pinging a host in the Hebrew University from the Technion takes about 11ms; while pinging a machine in the USA takes hundreds of milliseconds. Therefore, the difference in the response time in accessing objects spread over the Internet might be dramatic, regardless of the Object Request Broker (ORB) being used. Furthermore, the time of each specific service access cannot be forecast because end-to-end Internet performance is extremely difficult to predict and analyze [47]. Moreover, the scalability of such services is likely to be limited by the relatively high communication overhead imposed on servers exporting the services. These can adversely affect the end-user experience, rendering such services unreliable and therefore useless for practical purposes.

As in other areas of computer science, caching can be used to improve responsiveness, availability, predictability, and scalability of distributed services offered through middlewares. Specifically, accessing a copy of an object that is cached near the client is much faster than accessing a far away object. Moreover, shorter connections have lower chances of becoming congested and failing, and thus accessing a local copy is more likely to succeed and behave in a predictable manner. Finally, if most client requests are handled by caches, fewer requests will reach the server. Thus the system as a whole will be able to handle more concurrent client requests.
1.2 Results of This Work

1.2.1 Introducing CASCADE

In this thesis, we strive to enrich the CORBA middleware with a general purpose caching service and to explore both theoretical and practical aspects of its construction. We developed a Caching Service for Distributed CORBA Objects (CASCADE) [34, 35] that offers a scalable and flexible framework for general CORBA objects. CASCADE facilitates scalable application design by building cache hierarchies for the objects it manages. The hierarchy construction is dynamically adaptive with respect to service demand. As different applications have different consistency, security, and persistence requirements, our architecture is highly configurable with regard to a broad spectrum of application parameters. CASCADE allows client applications to fully control many aspects of object caching by specifying a variety of policies for cache management, consistency maintenance, persistence, security, etc.

Our work is based on a high abstraction level provided by standard, commercially available CORBA compliant ORBs, and is aimed at preserving native programming models wherever possible. Furthermore, CASCADE design strictly follows the standard CORBA services design principles outlined in [94].

CASCADE caches both object data and code. Code caching allows us to preserve the standard CORBA programming model: The application works with the cached copy through the same interface as it would have worked with the original object. In addition, all object methods (including updates) can be invoked locally eliminating the need to contact the remote object.

1.2.2 Applications

Probably the class of applications that would gain the greatest benefit from using CASCADE in the most straightforward manner is electronic information services. A typical representative of this wide application class is a “yellow pages” service. The performance of this application can be boosted by using CASCADE because it only requires weak consistency guarantees and because queries are typically far more prevalent than updates. Yet, in this thesis we also consider how CASCADE can be exploited by applications with stronger requirements such as a ticket reservation service and distributed games. Furthermore, we show how CASCADE can be used in multimedia services and distributed collaborative applications due to enhanced scalability, flexible policy governed guarantees and support for code caching, concurrent updates on multiple cached copies, and pluggable cache replacement algorithms such as those intended for partial object evacuation.

1.2.3 Communication Infrastructure

As we mentioned above, CASCADE strives for better scalability by employing dynamically built cache hierarchies that are used for disseminating information about this object (e.g., object updates) among the caches. Some advantages of using a hierarchy as compared with other distributed architectures are:
Conserved WAN bandwidth consumption: The bandwidth consumption of communication over a tree is low because each message is sent only $|V| - 1$ times, where $V$ is the number of nodes in the tree.

Improved scalability: The stateful communication over a hierarchical architecture is known to be scalable because of the low number of simultaneously opened node to node connections (i.e., small fan-in and fan-out degrees) and because of the small communication state kept at each node.

Easy management: It is easier to add or to remove a cache node in a tree than in other distributed architectures. In addition, a tree can be relatively easily reconfigured (a root of the tree can be moved etc.).

Easy consistency maintenance: If strong consistency among cached copies is required, a universally known root of the hierarchy can impose a total order on object updates.

1.2.4 Consistency Maintenance

One of the main challenges that a caching services faces is keeping cached copies of the same object consistent, at least to some degree. The definition of the consistency semantics provided by a replicated system is known as a consistency condition. There is a known tradeoff between the performance and scalability of a replicated system and the strength of the consistency condition it may guarantee, as discussed, e.g., in [16, 17]. A weak consistency condition can be implemented more efficiently. On the other hand, a weak condition means that data in replicas may diverge considerably. This may be an annoyance for clients of database systems, may complicate the programming model in distributed shared memory systems, and in extreme cases may even be too weak for solving certain problems [15]. This tradeoff is the motivation behind much of the research dealing with consistency conditions, whether it is in the area of databases, distributed shared memory, replicated objects, or Internet caching.

In this thesis, we explore the consistency requirements from object based distributed services operating in wide area networks. In particular, we propose a highly flexible composability approach to consistency in such services. We start by presenting a formal framework that enables composing a collection of consistency conditions into a more restrictive condition. We then identify a list of six very basic consistency conditions and prove that various compositions of these basic conditions yield several well known consistency conditions, such as sequential consistency [72], causal memory [10], and Pipelined RAM (PRAM) [74]. Moreover, we list several applications that can benefit from weaker consistency semantics than PRAM that are easily expressed as a composition of a small set of the basic conditions.

From a theoretical point of view, looking at some basic consistency conditions and composing them into higher level ones presents an important insight into the differences between the high level conditions. Also, for a given set of $n$ consistency conditions, there can be $2^n$ possible compositions. The composability framework gives us access to all of these combinations without having to formally define each of them independently.

From a practical point of view, this has two main advantages: First, it can simplify proving that a given implementation of a high level condition is correct. Specifically, the
basic conditions we propose are very simple. Thus, it is likely to be easier to show that a given implementation obeys the relevant basic conditions rather than proving directly that it obeys the corresponding high level condition. In case an implementation is incorrect, it may be possible to identify the errors when considering the simpler conditions in a more effective and accurate way than when looking directly at the composed condition. Second, we hope that our work can be used to devise flexible and composable implementations of consistency conditions. In such an ideal system, each basic condition will be implemented as a layer of code. When a given application needs a specific high level consistency condition, it could simply pick the layers that implement the collection of basic conditions from which the high level condition is composed of.

The basic consistency conditions that we list are inspired by the work on the Bayou project [118]. However, in Bayou the conditions were specified in an informal, operational, and implementation-dependent way, while our definitions are formal and implementation-independent. Moreover, the work on Bayou did not address the issues of composability.

We then describe how we have implemented this framework of composable consistency conditions within CASCADE. Although our implementation of consistency conditions is based on the widely known notion of version number and vectors, it is nevertheless unique in exploiting the peculiarities of hierarchical cache architectures such as the one used in CASCADE. We envision that similar techniques can be applied to other systems that employ hierarchical caches. It should be emphasized that our implementation preserves consistency guarantees even when clients access cached copies at different servers during the execution. Furthermore, we have designed novel optimizations that reduce the amount of information transferred between roaming clients and static servers. We believe that this latter contribution can be applied to other systems where it is required to maintain consistency for mobile clients. As Internet mobile clients become more common, we expect that our techniques will be useful for a wider range of applications.

In recent years it has become increasingly common to construct complex distributed systems and applications from independently built separate components, where each component is responsible for a well defined part of the complex system functionality and tailored for a specific operational environment. While this trend is especially prominent in modern object-oriented middlewares like CORBA, .NET, and EJB where each component is viewed as a distributed object, it can also be seen in other areas of distributed computing, e.g., in interconnecting shared memory systems [43]. Typically, components in these systems are autonomous and even built without a priori knowledge of each other which limits the ability to coordinate the operation of different components. This poses the challenge of achieving system-wide consistency guarantees when components are replicated. Fortunately, it holds for some consistency conditions that once the condition is preserved independently by each replicated component implementation, it will be preserved by the system as a whole without any component coordination. This highly desired property of consistency conditions called locality [59] is explored in this thesis. Not only does locality facilitate software engineering of composite systems, but it also greatly simplifies the proof of correctness because it is enough to consider the operation of each individual component separately.

In the past, it was known that linearizability, which is the strongest consistency condition, is local [51, 59]. It was also established that sequential consistency is not local for a particular object type system [59]. However, there was no systematic approach to locality analysis
of consistency conditions. In this thesis we introduce new techniques, such as separating sequence construction, for testing locality of a consistency condition by exploiting few basic properties of all existing object type systems. These techniques allow us to formally prove that among strong consistency conditions only linearizability and equivalent conditions have the locality property while for weak conditions only conditions weaker than or incomparable to PRAM can be local. In particular, most consistency conditions that are known to present a good compromise in the tradeoff between consistency and performance (e.g., PRAM, causal consistency [9, 10], and cache consistency [53]) turn out to be non-local. Furthermore, sequential consistency is shown to be non-local for virtually any object type system.

In order to prove these general results, we establish a generic framework for defining consistency conditions based on the notions of consistency order generator and serialization set. This framework is shown to be useful for comparing the relative strength of various consistency conditions. It is applicable for consistency conditions that can be specified using partial orders, which includes most known consistency conditions in the area of distributed shared memory. While a few consistency conditions have been left uncovered by the definition of a consistency condition in the form presented in this thesis (e.g., release consistency [32], entry consistency [25] and hybrid consistency [49]), this definition can be extended to prove that those conditions are not local as well.

1.2.5 Cache Management

Another inherent task of any caching service is cache management. Since the cache size is limited, the service can hold only a limited number of cached copies. When an object copy is to be brought to a cache that has no more space, some other object must be evacuated from that cache first. Note that a general cached object typically consumes much more resources than a simple web document. While a web document is usually kept on a disk, objects have to be in the main memory in order to execute operations on them. Furthermore, since objects are running, they consume CPU and possibly other resources. Therefore, the choice of replacement policy can have significant influence on performance.

CASCADE supports a wide range of cache replacement policies, including LRU [115], LFU [115], LFU with aging [14], Size and Inverse Size [122], Greedy Dual Size Frequency [33], and others. In addition CASCADE introduces policies that take into account parameters related to the knowledge of the hierarchy such as the number of descendant nodes this DCS has in the examined object’s hierarchy. Implementation of these policies as well as their analysis are part of Atzmon’s master thesis [18].

1.2.6 Mobile Code

For many objects, code is longer than data. Therefore, proper code caching is essential for efficient service operation. In particular, it is important to take into account that not all pieces of an object code are necessary for a successful object deployment: while some parts of the code can be executed quite often, others can be called infrequently or not used at all. Such unused parts of the code do not need to be cached and transferred over the network.

Typically, a CORBA object is coarse grain: it incorporates complex application logic and it is composed of many programming language objects. In particular, CORBA object
code comprises multiple programming language code classes. This partition determines the granularity of code caching in CASCADE: each individual class can be cached as a unit. Thus, the goal of efficient code caching in CASCADE is to bring classes into the cache right before their first use and evacuate them from the cache when they are no longer used. To this end, CASCADE introduces a notion of directed object class graph whose nodes are classes and whose edges denote various relations between the classes. By using efficient graph traversal algorithms tailored for a class graph and analyzing the relations between the classes, CASCADE strives to determine what classes need to be brought to cache and at what moment.

1.2.7 Performance and Flow Control

To evaluate the performance of CASCADE, we conducted many experiments in various settings. We ran over a hundred of tests on nine hosts at geographically disperse locations over the Internet, each test containing about 100,000 invocation requests. These tests were conducted at different hours of the day and on different days of the week during a period of about four months. Testing the system in an actual wide area environment allowed us to assess the response time of the caching service as well as the network level properties such as throughput. We also ran simulations in a local area network to evaluate hit rate, service scalability and other cache properties.

A substantial amount of data can be exchanged between the caches, mainly due to propagation of updates and update results. Studying the properties of the communication between the caches in the Internet allowed us to gain insight necessary for devising and evaluating flow control techniques that lend themselves to multi-party distributed applications operating in wide area. The internal communication between caches is done through CORBA, which uses IIOP/GIOP as the main communication protocol. IIOP is built atop TCP, which has an elaborate flow control mechanism that resulted from many years of research. However, this mechanism is tailored mainly for “classical” (Ethernet, token ring, and similar) LANs and for interactive applications like Telnet. It has serious drawbacks for most other applications as suggested by studies of persistent HTTP [57, 89], Sun RPC [38], ATM networks [85], and X Windows System. The results of this dissertation confirm some of these problems such as the effect of the Nagle algorithm, conservatism of most TCP implementation in handling message losses, etc. While recent TCP performance extensions such as TCP-Vegas [27] alleviate some of these issues, they are far from being solved yet. Furthermore, most currently existing middlewares built atop TCP do not take into account these TCP performance issues when running over WANs and do not attempt to cope with them.

In this thesis, we present the flow control techniques used in CASCADE for handling CORBA and TCP based communication, such as the buffering mechanism, connection management, interaction between the threads servicing the same connection, etc. We also discuss the performance gain resulted from employing these techniques. In addition, we draw a number of suggestions for middleware architects such as ways of optimizing the TCP connection management in middlewares that operate in the Internet.
1.3 Road-map

The rest of this thesis is organized as follows: Chapter 2 discusses relevant related work such as the use of caching in various contexts. Chapter 3 presents an overview of CASCADE design and architecture. Chapter 4 describes the theoretical consistency model of CASCADE that lends itself to wide-area distributed services. The details of the CASCADE implementation including hierarchy construction, consistency maintenance, cache management, mobile code support, and flow control mechanisms are given in Chapter 5. Chapter 6 discusses different classes of applications that could benefit from using CASCADE. Chapter 7 presents the experimental design of the testbed that was used for running CASCADE in a wide-area environment and conducting simulations in local area networks. Then it describes the performance measurements of response time and other communication characteristics that were evaluated in these experiments. Finally, Chapter 8 summarizes the thesis and outlines potential future work.
Chapter 2

Related Work

2.1 Caching in Various Software Systems

2.1.1 Object Caching in CORBA Compliant Systems

To the best of our knowledge, there is no programming framework provided by commercially available ORBs that allows for caching general CORBA objects. A very limited solution is provided by the so called smart stub mechanism provided by some existing CORBA implementations (e.g., Orbix [62] and VisiBroker). This mechanism allows an application programmer to override automatically generated client stubs. Using smart stubs it is possible to cache results of method invocations so that subsequent method invocations will return locally cached values without invoking the remote operation. This framework, however, is not general enough for implementing a generic caching mechanism that will allow for caching true CORBA objects and not just some method-specific information. Moreover, with smart stub based caching the burden of maintaining coherency of cached object copies lies entirely on the application programmer.

MinORB [80] is a research ORB that allows caching partial results of read method invocations at the client side. Since this system is conceptually similar to a caching solution that can be built using smart stubs, it bears the limitations incurred by this approach as described above.

The ScaFDOC system [68] is another research project implementing an object caching service for CORBA compliant systems. ScaFDOC provides multiple consistency levels for copies of cached objects. This system supports several protocols for various cache consistency semantics [67]. However, it does not support caching of active CORBA objects (both data and code). In addition, though cache consistency protocols used in this system scale relatively well, the system architecture is not hierarchical and is not aimed to operate in wide area networks.

A transactional-based model, called COCC (CORBA object-based caching with consistency), is described in [121]. A COCC transaction is defined as a sequence of one or more read operations on one or more objects, followed by an update operation on a single object. The COCC model supports inter-transaction caching, which allows clients to retain the contents of the cache across transaction boundaries. This kind of caching requires a cache consistency protocol to ensure that a client’s view of the data is globally consistent.
The COCC model employs a backward validation optimistic scheme rather than using locks at the server or client. COCC allows to read locally cached objects without server intervention. However, prior to update/commit, the server performs backward validation on the transactional objects. In this phase, the server must ensure that the validating transaction has not read a stale version of an object, i.e. one that was successfully updated by another transaction. COCC also handles data shipping and replication granularity. By default, CORBA ships attributes to clients and avoids data replication. Every request on an object results in an attribute read. This shipping approach of attributes causes a large number of remote invocations. The COCC model performs object level data shipping and object level replication management, which decreases the network traffic significantly.

The work reported in [117] improves COCC by locking operations in the server instead of locking objects, based on a new variation of optimistic two phase locking. This new method avoids locking at the client side, by using a per-process caching system.

2.1.2 Web Caching

Caching web pages is nowadays a hot research topic (see, e.g., [7, 11, 23, 29, 82, 83, 92, 100, 124, 125]). Web caching is intended for use in wide area internets, similarly to our caching service, and scalability is also a major concern here. However, the contents of caches can change only as a result of a primary copy update. Moreover, the web caching model is limited to data objects (HTML pages) only and does not deal with general objects that include executable code. This lack of ability to cache executable code such as the scripts that produce Web documents severely limits the possibility of caching dynamic content. This is considered a major deficiency because most Web content is dynamic nowadays (see, e.g., [48]).

The idea of Active Web Caching [30] can be considered a step towards active object caching. This work proposes to attach an applet to each web document. When a document (or its cached copy) is retrieved, the applet is executed.

Another point to be stressed about web caching is that the user has extremely limited control over caching decisions. The work of [90] describes several typical Enterprise network scenarios when incorrect caching decisions eliminate any improvements in response time gained from using web caching. In contrast, caching decisions in our service are application-controlled. Therefore, proper application choice can eliminate all of the above mentioned problems.

2.1.3 Caching in File Systems

Distributed file systems consisting of workstations and shared fileservers typically have main memory disk block caches at both the workstations and the fileservers. These caches form a two-level hierarchy, in which file I/O requests (making up a disk reference stream) generated by applications running on the clients may be satisfied at either the local cache or the file server cache. These caches improve system performance by reducing the frequency with which costly disk operations are performed, and, in the case of the caches at the workstations, the frequency of requests to the fileservers. The suitability of various replacement policies for file server caches is studied in [123].
Another application of caching in distributed file systems is for improved availability in the case of network failures. An example is the Coda file system, in which caching is the basis of disconnected operation [66]. Disconnected operation is a mode of operation that enables a client to continue accessing critical data during temporary failures of a shared data repository. The idea is that caching of data, which is used for performance, can also be exploited to improve availability. Clients view Coda as a single, location-transparent shared Unix file system. The Coda name-space is mapped to individual file servers at the granularity of subtrees called volumes. At each Coda client, a cache manager, called Venus, dynamically obtains and caches volume mappings. Venus uses a cache coherence protocol based on callbacks to guarantee that an open of a file yields its latest copy. It also employs whole-file caching. Whole-file caching enhances scalability and adds the advantage of a simple failure model: A cache miss can only occur on an open, but never on a read, write, seek or close. This substantially simplifies the implementation of disconnected operation.

2.1.4 Caching and Replication in Object-Oriented Database Systems (OODS)

Some OODS have been designed to provide persistent storage for objects that include methods, e.g., O2 [40], GemStone [28] and Thor [75]. For example, the Thor system [75] supports highly-reliable and highly-available access to storage. For this purpose its object repositories (ORs) can be replicated, and objects can migrate from one OR to another. This system also supports client-side caching, providing a mix of consistency guarantees that can be determined by the user. It uses a variant of copying garbage collection [20] to manage the cache. In order to achieve type-safe sharing, all object implementations in Thor are required to be written in the Theta [76] programming language.

2.2 Individual Aspects of a Caching Service

2.2.1 The Service Approach to Caching and Migration

Java\textsuperscript{TM} [24] enables writing mobile programs that run on different hardware platforms and operating systems. Recently, several distributed systems and architectures (e.g., Voyager [91], FarGo [3] etc.) that utilize these features to provide a transparent or application-controlled object migration have emerged. Yet, these systems dictate their own programming model to the application. In contrast, our work is based on a higher abstraction level provided by standard, commercially available CORBA compliant ORBs, and is aimed at preserving native programming models wherever possible.

Many services use a common technique of intercepting application messages and piggybacking service related information on them transparently for the application. This is done in order to enrich the service semantics without breaking the application programming model. Specifically, the sender appends the required information to the message prior to sending it and the receiver strips the message from this information before the message is delivered to the application. For example, the CORBA Object Transaction Service [95] passes transaction context in this manner. [70] employs interceptors to piggyback consistency related data, which is very similar to the use of interceptors in CASCADE.
2.2.2 Communication Infrastructure and Dissemination Mechanisms

Hierarchy is the most commonly used logical topology by many decentralized distributed applications operating in the Internet. For example, DNS servers are organized into a hierarchy so that a resolution request that causes a cache miss on some DNS server is forwarded to the parent DNS server. Web proxies also form hierarchies through which requests for documents are propagated.

Hypercube is another logical topology used by applications in which communicating peers are widely dispersed. In particular, many peer-to-peer systems and resource location services (e.g., Chord [114], Pastry [108], Tapestry [127] and the work of [102]) are built upon hypercube graphs. Another application of the hypercube topology is for stability detection [50].

Few other graph types serve as a basis for some peer-to-peer systems and location services. For example, the Viceroy system [78] emulates a butterfly network while the CAN algorithm of [104] emulates a multi-dimensional torus.

All aforementioned topologies are known for their scalability as the node degree in these geometrical forms is limited. Hierarchy is the most scalable and simplest topology compared to others but it is more vulnerable to failures. Being a sparse structure, tree conserves bandwidth because it has no redundant links between the nodes but propagation over a tree takes longer because of its bigger diameter. Probably the most important advantage of hierarchies for a caching service is that the hierarchical architecture simplifies implementation of semantic guarantees such as consistency conditions.

When we started working on CASCADE in 1998, no standard information dissemination mechanism that was available at that time suited the needs of a caching service. Network-level protocols such as IP multicast [39], Reliable Multicast Transport Protocol (RMTT) [99] and Scalable Reliable Multicast (SRM) [45] do not provide flexibility as they do not expose any control to the application. Furthermore, IP multicast lacks semantic guarantees such as reliability whereas RMTT does not scale well.

During the last fifteen years we have witnessed the emergence of various replication systems that use propriety protocols for exchanging information between the replicas. Like a caching service these systems typically focus on achieving a variety of semantic guarantees. However, the event dissemination mechanisms (e.g., quorum based schemes) used in these systems exhibit limited scalability and they are usually geared toward computing within a cluster rather than Internet wide distribution. A study that evaluates availability of such quorum based systems over the Internet can be found in [13].

Just recently, peer-to-peer Internet applications have been popularized through file sharing applications like Napster, Gnutella and FreeNet [4, 6, 36]. The two major objectives of such systems are balanced distribution of information and scalable location service. However, all these systems are primarily intended for sharing of data files; reliable content location is not guaranteed or necessary in this environment. A second generation of such applications, including CAN [104], Chord [114], Pastry [108], Tapestry [127] and Viceroy [78], improve on the earlier work by guaranteeing a definite resource location within a bounded number of network hops.

Peer-to-peer applications share many goals with a caching service such as self-organization and scalability. In particular, a caching service can benefit from advanced techniques that peer-to-peer systems use for adaptation to network and node failures. A strong interest of
the scientific community in this new type of applications led to a large amount of research on finding the shortest path between the peers. It should be noted that this research typically chooses the number of network hops as a metric of distance between a pair of nodes. Furthermore, it is generally assumed for the sake of efficient service routing that this metric preserves the triangle inequality. However, this assumption does not hold in wide-area networks as shown in this thesis and previous research on end-to-end performance of the Internet.

While peer-to-peer applications came up with many useful techniques that could be applicable in a widely distributed caching service, they did not actually provide a mechanism for information dissemination. Such a mechanism, however, can be found in recently emerging publish-subscribe systems. We are especially interested in the publish-subscribe services that are built on top of peer-to-peer platforms such as Bayeux [128] and Scribe [31] because they retain scalability and other beneficial properties of peer-to-peer computing. At the same time, these systems provide only limited variety of semantic guarantees such as reliability and ordering. While the API of Scribe allows for incorporating additional semantics by defining and implementing application upcalls, it remains to be seen how various consistency guarantees can be maintained in presence of network failures and dynamic reorganization of the logical topology.

2.2.3 Consistency in Shared Memory and Distributed Systems

Our caching service is highly configurable with respect to a great variety of consistency disciplines that can be enforced on the cached object copies. Here we benefited from the vast amount of research that was dedicated to implementing shared memory systems with various consistency guarantees, including sequential consistency (sometimes referred to as strong consistency) [72], weak consistency [61], release consistency [52], causal consistency [9, 10], lazy release consistency [65], entry consistency [25], and hybrid consistency [49]. In contrast to our service, such systems are geared towards high-performance computing, and generally assume non-faulty environments and fast local communication. We refer the reader to [16, 17, 49, 64, 70, 77, 129] for other applied and theoretical studies of consistency strategies.

Much less attention, however, was devoted to exploring consistency guarantees suitable for object-oriented middlewares, especially for middlewares in which a client is not bound to a particular server and can switch the servers all the time. The idea of defining and using fine grain consistency conditions, which is explored in this thesis, is motivated by Bayou project [118]. That work introduced a set of basic consistency conditions for sessions of mobile clients and discussed version vectors as a possible way of their implementation. It also brought numerous examples illustrating that these conditions are indeed useful for applications. However, these definitions are introduced in [118] as constraints on an implementation and are defined in a framework of a particular database model. Furthermore, that work did not consider composability of consistency conditions.

The Globe system [120] follows an approach similar to CASCADE by providing a flexible framework for associating various replication coherence models with distributed objects. Among the coherence models supported by Globe are the PRAM coherence, the causal coherence, the eventual coherence, etc.

The recent LOTEC protocol [54] maintains consistency for nested object transactions.
This protocol spares the programmer the burden of explicitly specifying synchronization operations needed for transactional processing. Finally, the novel Millipage [63] technique enables efficient control over the granularity of a shared unit, thus enabling applications to achieve good performance while maintaining sequential consistency.

Object-based shared memory systems, in which consistency guarantees are given per object, have also been studied. Orca [22] supports object replication and migration with strong consistency guarantees. However, all objects in this system must be written in a special Orca language.

Spring [87] is a distributed operating system that provides a unified caching architecture that can be used for caching different types of remote objects. However, this system does not provide generic support for a variety of consistency and other requirements inherent for object caching. In order to address the requirements of a specific object type, the application developer must reimplement the object itself.

[59] was allegedly the first work to consider locality of consistency conditions. In particular, it established that linearizability is local regardless of the object type system. This work also showed that sequential consistency is non-local for a particular object type (the Enq-Dec type system for FIFO queues). [51] proved that normality is equivalent to linearizability which implies its locality. However, we are not aware of any work that studies locality of consistency conditions in a systematic manner.

2.2.4 Communication Analysis in Wide Area Networks

Obtaining data on different aspects of Internet communication is an emerging research direction. Some active research in this area focuses on measuring and analyzing the constancy of Internet path characteristics such as routing, loss, and throughput [126, 47]. Such research focuses primarily on general form of point-to-point communication. It does not consider particular communication patterns such as RPC-based communication. Neither it pays specific attention to multicast communication that is used, e.g., for cooperation between the caches. Another related project [32] studies the nature of communication failures duration and location and how they effect the end-to-end availability of wide-area services. These research efforts are orthogonal and complementary to our performance analysis of the caching service. A few studies (e.g., [21]) evaluate the running times of distributed algorithms in the settings of wide area networks. While the communication infrastructure used in this work bears many similarities with ours, the goals of that study and the communication patterns used in it are completely different.
Chapter 3

Introducing a Caching Service for Distributed Objects

In this chapter we present the overview of CASCADE, a Caching Service for Distributed Objects that we introduced.

3.1 The Model of the CORBA Middleware

Object oriented middlewares such as CORBA are based on an RPC style mechanism (see Figure 3.1). At the core of such middlewares lies the notion of a distributed object that provides some service. Distributed objects run in an address space of a server that instantiates them. Application clients may access the service by invoking remote requests on the objects. The most popular request invocation style is synchronous in which a client that invoked the request is blocked until the request execution is finished and a reply is returned to the client. An invocation is performed via a remote object pointer that is called an object reference.

The service provided by an object is defined by the object interface that should be public and readily accessible for interested clients. On the other hand, the implementation of the object is hidden from the clients; it is rather internal to the server that runs the object.

The most important part of the CORBA architecture is the Object Request Broker, commonly abbreviated to ORB, which is responsible for delivering invocation requests and replies. Objects connect to the ORB through an object adaptor that is responsible for accepting an invocation request, demultiplexing it to the intended object, activating the object if necessary, assigning a thread to execute the request, and performing other common server tasks.

In addition to this request invocation architecture, CORBA comprise a set of standard services that facilitate development and deployment of distributed applications. For example, the naming service translates a human readable object name to a remote object reference, thus allowing a CORBA application client to locate an object by its name. Furthermore, the architecture can be enriched by other generic services that are not defined by the standard. Several such services have been already developed (e.g., the CORBA replication service [42]). Adding a caching service would definitely benefit numerous CORBA applications.
3.2 General Design Goals

CASCADE is designed along the following guidelines: the service is given by a set of caching servers that provide their address space for cached objects. Object copies are never held by the application itself: caching objects on the server side allows for better manageability and scalability because the number of application clients can be potentially very large.

3.2.1 A General Purpose Service

CASCADE aims to be a general purpose caching service that is not tailored to one specific application. In particular, we should take into account that general objects are substantially more complex than, e.g., simple Web documents, because they contain structured data and executable code. In CASCADE, objects are cached along with their code. This way, a cached copy preserves all of the advantages of a distributed object. Specifically, the client application should only know the interface of the object but it should not be concerned with the internal object implementation nor with the structure of object data. Furthermore, caching an object code and transferring it on the fly eliminates the need to pre-distribute this code at all of the locations where the object can be potentially used.

Another challenge that a general purpose service has to face is the wide variety of application requirements. Indeed, various distributed applications have quite different requirements with respect to consistency, persistence, security etc. If the service always provides the strongest possible guarantees, the performance of the applications that do not need such strong guarantees will suffer. Thus, the service should be flexible enough to provide a range of guarantees that would lend itself to applications both with weak and strong requirements. To this end, CASCADE introduces a set of policies that determine how a cached object is handled by the service. Each policy is responsible for a particular aspect of the object maintenance, e.g., consistency policy, persistence policy etc.

We define several policy classes: *per-object*, *per-session* and *per-request* policies. Per-
object policies determine global object semantics that hold for all cached copies of the cached object. An implementation of such policies requires global coordination between the servers (e.g., imposing a total order on all object updates). Once such policies are defined for a particular object, they do not change over the object lifetime. In contrast, per-session policies apply only to individual application clients. Their implementation do not need to be globally coordinated and they can be modified dynamically. Per-request policies provide even more flexibility as they allow an application developer to specify the semantics of each individual request.

3.2.2 Service Transparency

Another important design goal of a caching service is preserving the original application programming model, as was presented in Section 3.1, to the maximal possible extent. This is particularly challenging because we need to be able to introduce the flexibility of policies without compromising the ease of use. To this end, CASCADE provides a library that should be linked with the client application. This library aims at simplifying the interaction with the service for the application programmer. In particular, the library can piggyback input parameters on the request and strip output parameters from the reply transparently for the application. In principle, it is possible to port the application to CASCADE without using the library. However, it requires more modifications in the original application and greater effort on the side of the application developer.

To port an existing CORBA application to CASCADE, the original application server (i.e., the application service provider) has to register the object with the caching service and specify all of the policies to be used with this object. However, the application clients (i.e., the application service consumers) only need to indicate their interest in a cached copy and obtain a remote reference to a cached copy. Once such a remote reference is obtained, the client can invoke requests on the cached copy in exactly the same fashion it would be doing so on the original object. In most cases, the clients do not have to deal with policies. However, to allow for greater flexibility, each application client is permitted to override the default per-session and per-request policies specified at the object registration time. Of course, policy modifications that a client requests apply only to the session of this client.

3.3 Compliance with the CORBA Services Standard

The CORBA standard specifies a collection of object services, called CORBA services, that support basic functions for using and implementing objects [94]. These services are needed to support meaningful and productive communication at the application level and are useful for any CORBA applications regardless of their specialization. The most prominent examples of CORBA services implemented by most ORB vendors are the Naming Service, the Event Service, and the Transaction Service.

In this respect our design goal was twofold: to preserve the programming framework provided by the standard CORBA services, and to allow co-existence and cooperation with these services. To achieve this, we strictly followed the standard CORBA services design principles outlined in [94] (see below). Thus, our caching service can be viewed as another
useful object service aimed at improving responsiveness and availability of any CORBA-based application independent of application domain.

The following is a summary of the CASCADE features that place it in line with the standard CORBA services:

- The design of CASCADE uses and builds on CORBA concepts: separation of interface and implementation, object references are typed by interfaces, clients depend on interfaces, not implementations, etc.

- The service provided by CASCADE is generic with respect to cached object types.

- CASCADE allows local and remote implementations: The caching service is structured as a CORBA object with an OMG IDL interface and can be implemented as either an application library or a standalone server.

- CASCADE does not depend on any global identifier service or global id space in order to function. All internal CASCADE components that require some kind of identification rely on ids generated internally by CASCADE. These ids are unique only within the caching service scope and are invisible for the client applications.

- Finding the caching service is at a higher level and orthogonal to using the service. Since the caching service is structured as an object, all that is needed for accessing the service is its interoperable object reference (IOR). The latter can be found using any general purpose service (e.g., the Naming Service).

### 3.4 Choosing the Programming Framework

Our caching service imposes two principal requirements on the programming framework: (1) support for dynamic object loading, and (2) a possibility to pass run-time objects via the network. While the second requirement can be solved, e.g., by means of CORBA (once the CORBA pass-by-value standard matures), meeting the first requirement with many existing programming environments is platform dependent and often requires significant expertise on the application programmer side.

In this context, Java [24] seems an appealing programming tool because it has a built-in platform independent support for dynamic object loading and object serialization. Moreover, Java has recently become a de-facto standard for Internet-wide computing. Therefore, we implemented CASCADE in Java.

However, this choice also poses a few challenges: While a Java virtual machine constitutes a complete deployment environment similar to that of an operating system, it does not provide the same level of control to the running applications. For example, without instrumenting a JVM or changing the object implementation, it is impossible to save an image of a running Java thread on a disk and kill the thread in order to create another thread later and restore the execution. Furthermore, Java provides no efficient access to the address space of the running environment. The interaction with the memory management algorithms is limited: For example, an object image cannot be removed from the memory when desired. It is only possible to give a hint to the garbage collector to remove the object later. Moreover,
there is no efficient way to evaluate how much space an object takes nor how much CPU it consumes. While most of these limitations have been deliberately introduced into the Java design in order to enhance security and programming abstraction level, they severely hinder development of services that deal with object management.

3.5 CASCADE System Overview

![CASCADE System Overview Diagram]

Figure 3.2: The Caching Service Architecture

Our caching service is designed along the following lines (see Figure 3.2): The service is provided by a number of servers each of which is responsible for a specific logical domain. In practice, these domains can correspond to geographical areas. We call these servers Domain Caching Servers (DCSs). Cached copies of each object are organized into a hierarchy. A separate hierarchy is dynamically constructed for each object. The hierarchy construction is driven by client requests. The construction mechanism ensures that for each client, client’s local DCS (i.e., the DCS responsible for the client’s domain) obtains a copy of the object. In addition, this mechanism attempts to guarantee that the object copy is obtained from the nearest DCS having a copy of this object (a detailed explanation of how this is done appears in Section 5.1). Once the local DCS has an object copy, all client requests for object method invocations go to this DCS, so that the client does not have to communicate to a far server.

The DCS that holds an original object becomes the root for this object cache hierarchy. It plays a special role in building the hierarchy and in ensuring consistency of the cached copies, as described further in this thesis.

Hierarchies corresponding to each object are superimposed on the DCS infrastructure. Different object hierarchies may overlap or be completely disjoint. Also, overlapping object
Hierarchies do not necessarily have the same root. For example, in Figure 3.2 the original copy of the object $X$ is located in the DCS of domain $A.B$. This DCS is the root of the $X$'s hierarchy. The cached copies of $X$ are located in the DCSs of domains $A$, $A.E$, $A.D$ and $A.E,X$. Note that, in addition to being the holder of the cached copy of $X$, the DCS of domain $A$ also serves as the root of the object $Y$ hierarchy. Further, the $A.D$’s DCS contains only cached object copies and the $A.D,X$’s DCS does not contain objects at all.

An object hierarchy is used for disseminating information about this object (e.g., object updates) among the DCSs. Compared with other distributed architectures, using a hierarchy has the following advantages:

Conserved WAN bandwidth consumption: The bandwidth consumption of communication over a tree is low because each message is sent only $|V| - 1$ times, where $V$ is the number of nodes in the tree.

Improved scalability: The stateful communication over a hierarchical architecture is known to be scalable because of the low number of simultaneously opened node to node connections (i.e., small fan-in and fan-out degrees) and because of the small communication state kept at each node.

Reduced initial response time: Since an object copy is obtained from the nearest member of the object hierarchy, it takes less time on average to bring this copy to a local DCS than to obtain the copy from the original object holder. Thus, the client waits less for the result of its first request addressed to this object.

Easy management: It is easier to add or to remove a server in a tree than in other distributed architectures. In addition, a tree can be relatively easily reconfigured (a root of the tree can be moved etc.).

Easy consistency maintenance: If strong consistency among cached copies is required, a universally known root of the hierarchy can impose a total order on object updates.

The main disadvantage of a tree is its vulnerability to node failures. A single node failure can disconnect the whole branch of the tree. This problem can be solved, for example, by local replication of each DCS using a primary backup approach (see, e.g., [26]).

The algorithms running in CASCADE require two different modes of information dissemination: propagation to all DCSs (i.e., flooding) and delivery to some individual server, e.g., the root of the hierarchy. In both cases, messages are sent only between neighboring nodes in the hierarchy. Even for the second mode, when a pair of non-neighboring DCSs need to exchange some data, the message is not sent directly between them but rather propagated through the hierarchy nodes on the path from the first server to the second. For example, in Figure 3.2 if the DCS $A.D,Y$ has some data to disseminate to $A$, it is sent from $A.D,Y$ to $A.D$ first, and then $A.D$ sends it $A$. There are two reasons for this implementation: a) by restricting network communication to direct neighbors we keep the number of simultaneously opened network connections on each DCS low independently of the hierarchy size, thereby rendering the system more scalable, and b) it is known that the triangle inequality does not hold in the Internet, i.e., sending information through an intermediate node can be more efficient and achieve better latency than sending this information directly to the destination.
This also gives an explanation why deep hierarchies can sometimes perform better than flat hierarchies in which all DCSs are direct sons of the root server.

The communication between DCSs is also implemented via CORBA, so that each DCS provides a service for other DCSs and implements a well defined IDL interface for internal requests. Thus, the inter-DCS communication benefits from all of CORBA’s advantages: interoperability, portability, reliability guarantees for communication, and a wide spectrum of services which can be used, e.g., for secure communication and for name resolution. It is also worth mentioning that each DCS runs a single object implementing the interface for internal requests rather than multiple objects, one per each cached object. This reduces the number of stubs used for DCS to DCS communication to one, thus improving the system’s scalability.

### 3.6 CASCADE System Modules

#### 3.6.1 The Client Structure

![Client Structure Diagram]

Figure 3.3: The Client Structure

The client structure is depicted in Figure 3.3. It consists of the following elements:

**Interceptor:** ¹

This module is responsible for interception of client invocations of cached object methods and for altering the content of the request transparently for the client application. This module is used for passing implicit request parameters and invocation semantics (see Section A and 5.3). It can also be used for encryption, decryption and authentication of client requests (see Section 8.3.2).

**Client Library:** The client library is aimed at facilitating the interaction between the client application and the caching service. In particular, the library hides the interceptor from the application. Thanks to the client library, the application interacts with the cached object in the same way as it would do with an ordinary CORBA object.

---

¹Interceptors are a common way of gaining access to CORBA’s communication protocol; they are a part of the CORBA 2.3 standard [86]. While they are to undergo a technical revision in CORBA 3, no conceptual changes are anticipated.
3.6.2 The DCS Structure

Figure 3.4(a) shows the DCS breakdown into modules. The internal layout of these modules can be divided into four layers:

Object cache: This is where objects cached in the local DCS actually reside. Each cached object is wrapped into a proxy object whose structure is depicted in Figure 3.4(b) and discussed in Section 3.6.3 below.

Policy implementations: This is a collection of implementations of various per-object policies. Each policy implementation is stateless with respect to any individual object as it is shared among all cached objects. However, for each cached object a policy implementation is parameterized by the corresponding policy object. Policy objects along with the state of a policy implementation with respect to a particular object are located within the object proxy (see Figure 3.4(b) and Section 3.6.3).

Common task implementations: This part includes the implementations of the cache manager, the class loader and the thread manager. The cache manager controls insertion/deletion of objects to/from the object cache. Its implementation is based on a particular cache replacement policy implemented by the DCS (see Section 5.5). The class loader is responsible for loading the cached objects’ code into the Java virtual machine. Finally, the thread manager controls the assignment of threads and locks to various concurrent tasks as described in Section 5.8.

Communication management: This part comprises the application level routing and flow control modules. The application level routing module implements hierarchy management algorithms and determines how the information should be propagated. As we observed in Section 3.5, there are two data dissemination modes: data can be sent to all servers of a hierarchy or to a specific DCS. In both cases, the application level routing

(a) The DCS Modules

(b) The Proxy Modules

Figure 3.4: The CASCADE System Modules
module running on some DCS decides to what immediate neighbors of this DCS the information should be sent.

The flow control module regulates the flow of information between the servers. Specifically, it manages network connections, determines with what rate messages are sent, maintains various buffers etc. The full description of this module functionality can be found in Section 5.7.

In addition, there is an interceptor object underneath the DCS that intercepts client requests to cached objects. This is used to extract the implicit request parameters added by client side interceptors (see Section 3.6.1).

### 3.6.3 The Proxy Object Structure

Figure 3.4(b) shows the internal structure of the proxy of a cached object \( O_2 \). It consists of the following elements:

**The cached copy of \( O_2 \):** This includes the copy of \( O_2 \)'s state and code.

**\( O_2 \)'s info object:** This object consists of two parts: \( O_2 \)'s policy objects and \( O_2 \)'s accounting data. There is one policy object for each policy defined for \( O_2 \). The policies are configured when \( O_2 \) is first registered with the CASCADE system. \( O_2 \)'s accounting data includes various run-time statistics that are collected during its life-time, e.g., how many times each method has been called.

**\( O_2 \)'s hierarchy data and the state of policy implementations:** The knowledge of this proxy node about its location in the object hierarchy: its father node, its children nodes etc. In addition, it contains the state of consistency, persistence and other policy implementations.

**The method invocation control:** This module processes incoming method invocations based on the policies defined for the object: Whenever a method is invoked on the cached object, it is forwarded to the object proxy. The invocation is then processed by the method invocation control module that passes control to the appropriate policy implementations and supplies the requested invocation semantics (per-request policy) and the per-object policies as parameters.
Chapter 4

Consistency Model for Wide-Area Distributed Services

In this section, we introduce a framework of composable consistency guarantees that lends itself to general purpose services operating in wide area networks due to its flexibility and very fine granularity. We also investigate locality of consistency conditions since this property is deemed important for implementation scalability.

4.1 System Model

We generally adopt the model and definitions introduced in [59], but slightly adjust them to our needs. A system consists of a finite set of processes named $p_1, p_2, \ldots, p_n$ that communicate through shared objects. Each object has a name and a type. The type defines a set of primitive operations that provide the only means to manipulate that object; in this work we consider only single object operations. Each operation has a name, and input and output parameters.

Execution of an operation $op$ has a non-zero duration; this is modeled by an operation invocation event $inv(op)$ and a matching operation response event $resp(op)$. Execution of a system is modeled by a history which is a sequence (finite or infinite) of invocation and response events. Let $op$ be an operation on object $X$ performed at process $p$; $X$ and $p$ will be denoted as $obj(op)$ and $pr(op)$, respectively. Instead of the full notation $inv(op)$ on $X$ at $p$ and $resp(op)$ on $X$ at $p$ we will use $inv(op)$ and $resp(op)$ wherever operation’s object and process are not important.

A history $H$ is complete if for each $inv(op) \in H$, the matching $resp(op)$ also belongs to $H$. For the sake of presentation simplicity we consider only complete histories in this thesis; it can be shown that our results hold for incomplete histories as well but a more scrupulous case analysis would be required.

A history $H$ is sequential if (1) its first event is an invocation and (2) each invocation (response) event is immediately followed (preceded) by the matching response (invocation). A history $H$ is a serialization of a history $H'$ if $H$ is sequential and it consists of the same events as $H'$.

A process subhistory $H|_p$ ($H$ at $p$) of a history $H$ is the subsequence of all events in $H$ that occurred at $p$. An object subhistory is defined in a similar way for an object $X$.
it is denoted \( H|X \) (\( H \) on \( X \)). A history \( H \) is \textit{well-formed} if each process subhistory \( H|p \) is sequential. In the following we consider only well-formed histories. Such histories model sequential processes accessing concurrent objects. As some operations on the same object \( X \) may be concurrent, it is important to note that object subhistories of well-formed histories are not necessarily sequential.

A set \( S \) of histories is prefix-closed if, whenever \( H \) is in \( S \), every prefix of \( H \) is also in \( S \). A single-object history is one in which all events are associated with the same object. A \textit{sequential specification} for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history \( H \) is \textit{legal} if each object subhistory \( H|X \) belongs to the sequential specification for \( X \).

### 4.2 Consistency Conditions for Objects of General Type

#### 4.2.1 Refining Object Type System

In this section we further elaborate on the object type system to the extent necessary to show the results in the following sections. Each operation \( op \) has a name denoted \( \text{name}(op) \). An invocation event takes an input value \( \text{ival}(op) \) whereas a response event returns an output value \( \text{oval}(op) \). Loosely speaking, an operation can be seen as a trace of a function that possibly modifies an object state based on the input value and computes the output value based on the current object state. The domain and valid range of parameters are part of the type system.

An output value \( Ov \) is legal for a response event \( \text{resp}(op) \) in a sequential history \( H \) if \( Ov \) is in the valid range of output values for \( op \) and the sequential history generated from \( H \) by substituting the output value of \( \text{resp}(op) \) with \( Ov \) is legal. If an output value \( Ov \) is legal for a \( \text{resp}(op) \) in some sequential history in which the matching invocation \( \text{inv}(op) \) is passed an input value \( Iv \), then \( Ov \) is said to be legal for \( op \) with \( Iv \).

We now introduce few assumptions on the object type system and sequential specification. The following list is not intended to be complete in any sense; it just specifies the assumptions under which the results presented in Sections 4.2.4 and 4.3 hold.

The legality of a sequential history does not depend on processes at which history events were executed. Formally,

**Assumption 4.2.1:** If we take a pair of invocation and matching response events which occurred at process \( p_i \) in a legal sequential history and change their process name to \( p_j \), the resulting sequential history is also legal.

Informally speaking, functions modeled by operations are deterministic. This is captured by the following assumption:

**Assumption 4.2.2:** At most one output value can be legal for a response event in a sequential history.

Next, we assume that there are operations that model multiple return value functions, i.e., functions that may return different values in different executions:
Assumption 4.2.3: An operation op is called \emph{multiple return value} if there is some input value \( I_v \) of \( op \) such that there are at least two different output values that are legal for \( op \) with \( I_v \). We assume that each object type contains a multiple return value operation.

Intuitively, multiple return value operations model query functions that exist in all object types used in practice. For example, a \textsc{read} operation is multiple return value in the classical Read-Write type system. In the \textsc{Enq-Deq} type system for FIFO queues [59], \textsc{Deq} is also multiple return value operation. Note that a multiple return value operation may alter the state of an object (an example of such an operation is \textsc{read-modify-write}); this does not contradict the definition.

### 4.2.2 Definitions of Consistency Conditions

A history \( H \) induces an irreflexive partial order \( <_H \) on operations: \( op_1 <_H op_2 \) if \( \text{resp}(op_1) \) precedes \( \text{inv}(op_2) \) in \( H \). This order captures the notion of “real-time” precedence ordering. If \( H \) is sequential, \( <_H \) is a total order. Two operations are called non-concurrent if they are ordered by \( <_H \) and concurrent otherwise.

There are also widely known weaker operation orderings: According to process order \( \rightsquigarrow^\text{proc}_H \), \( op_1 \rightsquigarrow^\text{proc}_H op_2 \) if \( op_1 <_H op_2 \) and \( \text{pr}(op_1) = \text{pr}(op_2) \). An object order \( \rightsquigarrow^\text{obj}_H \) is defined similarly: \( op_1 \rightsquigarrow^\text{obj}_H op_2 \) if \( op_1 <_H op_2 \) and \( \text{obj}(op_1) = \text{obj}(op_2) \). [51] defines the normality order \( \rightsquigarrow^\text{norm}_H \): \( op_1 \rightsquigarrow^\text{norm}_H op_2 \) if \( op_1 <_H op_2 \), and either a) \( op_1 \rightsquigarrow^\text{proc}_H op_2 \), or b) \( op_1 \rightsquigarrow^\text{obj}_H op_2 \), or c) there is \( op_3 \) such that \( op_1 \rightsquigarrow^\text{norm}_H op_3 \) and \( op_3 \rightsquigarrow^\text{norm}_H op_2 \).

These orders lie in the basis of consistency condition definitions. Specifically, definitions of linearizability, sequential consistency and normality are based on \( <_H \), \( \rightsquigarrow^\text{proc}_H \), and \( \rightsquigarrow^\text{norm}_H \) respectively:

**Definition 4.2.1 (Linearizability):** A history \( H \) is linearizable if there exists a legal serialization \( S \) of \( H \) that preserves \( <_H \) (i.e., if \( op_1 <_H op_2 \), then \( \text{resp}(op_1) \) precedes \( \text{inv}(op_2) \) in \( S \)).

**Definition 4.2.2 (Sequential Consistency):** A history \( H \) is sequentially consistent if there exists a legal serialization \( S \) of \( H \) that preserves \( \rightsquigarrow^\text{proc}_H \).

**Definition 4.2.3 (Normality):** A history \( H \) is normal if there exists a legal serialization \( S \) of \( H \) that preserves \( \rightsquigarrow^\text{norm}_H \).

Note that these definitions differ only in the order that has to be preserved by \( S \). This common pattern of consistency condition definitions suggests a way of parameterizing a consistency condition definition by operation ordering.

Let \emph{consistency order generator} be a boolean predicate taking two processes, two objects, two operation names, two input values and two output values. Intuitively, consistency order generator determines for a pair of operations ordered by \( <_H \) in a history \( H \) whether this order should be preserved in the serialization of \( H \); the decision is based only

\[\text{The original definition of linearizability in [59] explicitly stipulates that } S \text{ preserve the process order of } H \text{ by using a stronger notion of history equivalence. It is easy to see that both definitions are equivalent because } <_H \text{ implies process order for well-formed histories.}\]
on the information about the operations themselves independently of the rest of the history. Formally, a consistency order $G_H$ generated by a consistency order generator $G$ is defined as follows: $e \sim_H f$ if $e \prec_H f$ and either $G(pr(e), pr(f), obj(e), obj(f), name(e), name(f), ival(e), ival(f), oval(e), oval(f))$ or there is an operation $g$ such that $e \sim_H g$ and $g \sim_H f$. For instance, the order generator $lin$ of $\prec_H$ always returns true. The order generator $proc$ of $G_{\sim_H}$ returns true if $pr(e) = pr(f)$; the order generators $obj$ of $\sim_H$ and $norm$ of $G_{\sim_H}$ are defined similarly.

**Definition 4.2.4 (Consistency Condition):** Consistency condition $CC(G)$ is a boolean predicate that takes a history $H$; $H$ satisfies $CC(G)$ if there exists a legal serialization $S$ of $H$ that preserves $\sim_H$. An implementation satisfies a consistency condition if every history generated by it satisfies this condition.

Note that this definition is given in terms of predicates rather than a specific set of serializations. We need this additional level of indirection in order to reason in an abstract way about the properties of consistency conditions (e.g., their strength) without limiting the scope of consideration to specific histories, or even particular type systems.

In addition to the most commonly used consistency conditions defined above, this definition covers many other conditions that may be used in specific systems. For example, consider the system that supports sequential consistency for normal updates and queries. In additional, it allows the client to query the most updated copy of the object by providing a strong query that should reflect the results of all preceding updates. Of course, strong query is more expensive compared to a normal query so clients should use it with discretion. A consistency order generator for this system would be $pr(e) = pr(f) \land \text{name}(e) = \text{UPDATE} \land \text{name}(f) = \text{STRONG QUERY}$. Another example is the system that provides sequential consistency for operations performed on most processes but has one master process whose operations are always done on the most updated copy.

Nevertheless, the above definition of a consistency condition is not general enough to encompass all consistency conditions existing in practice. First, it requires that a legal serialization of $H$ exist. This precludes weak consistency conditions (e.g., PRAM [74] and causal consistency [9, 10]) permitting different event sequences for each process. In Section 4.3, we discuss how the definition of a consistency condition can be generalized to include weak consistency conditions as well.

Second, not all consistency conditions are based on operation ordering. For example, much work on databases and distributed systems uses serializability [72] as the basic correctness condition for concurrent computations. In this model, a transaction is a thread of control that applies a finite sequence of primitive operations to a set of objects shared with other transactions. A history is serializable if it is equivalent to one in which transactions appear to execute atomically, i.e., without interleaving. This notion of atomicity, which implies that all events outside of a transaction should appear in a history either before the first event of after the last event of the transaction, cannot be captured by simple operation ordering. However, such consistency conditions are rare. In the rest of this thesis we consider only ordering based conditions unless stated otherwise.
4.2.3 On the Relative Strength of Consistency Conditions

Two consistency conditions $C_1$ and $C_2$ are equivalent if a history satisfies the first condition iff it satisfies the second condition. $C_1$ is stronger than or equivalent to $C_2$ if a history satisfies $C_2$ whenever it satisfies $C_1$; other comparative relations are defined likewise. If there is one history that satisfies $C_1$ but not $C_2$ and another history that satisfies $C_2$ but not $C_1$, then $C_1$ and $C_2$ are incomparable.

In a similar fashion we define comparative relations between consistency order generators: Two consistency order generators $G_1$ and $G_2$ are equally restrictive if for each history $H$, $\rightarrow H$ is equivalent to $\rightarrow H$ etc. Note that in all these restrictiveness relations it is sufficient to consider only sequential histories $H$. We prove this for the “equally or less restrictive” relation; the proofs for other relations are similar.

**Lemma 4.2.1:** Let $G_1$ and $G_2$ be two consistency order generators such that for every sequential history $S$, $\rightarrow S$ is weaker than or equivalent to $\rightarrow S$. Then $G_1$ is equally or less restrictive than $G_2$.

**Proof:** Assume, by way of contradiction, that there is a history $H$ for which $\rightarrow H$ is stronger than $\rightarrow H$, i.e., there is a serialization $S'$ of $H$ such that $S'$ preserves $\rightarrow H$ but not $\rightarrow H$. Then, there is a pair of operations $op_1$ and $op_2$ in $H$ such that $op_1 \rightarrow H op_2$ but $op_2$ appears before $op_1$ in $S'$. By the definition of consistency order, there is a sequence $S$ of operations in $H$ starting with $op_1$ and terminating with $op_2$ such that for each pair of consecutive operations $e$ and $f$ in $S$, $G_1(pr(e), pr(f), obj(e), obj(f), name(e), name(f), ival(e), ival(f), oval(e), oval(f)) = true$. Obviously, $S$ is a sequential history. Let us consider a subsequence $S''$ of $S'$ which consists only of events in $S$. It is easy to see that $S''$ is a serialization of $S$. By the definition of consistency order, $S''$ preserves $\rightarrow S$ but not $\rightarrow S$. This is a contradiction to the fact that $\rightarrow S$ is weaker than or equivalent to $\rightarrow S$. Q.E.D.

The notions of consistency condition strength and its generator restrictiveness are tightly related. It is easy to see that if $G_1$ and $G_2$ are equally restrictive, then $CC(G_1)$ and $CC(G_2)$ are equivalent. Furthermore, if $CC(G_1)$ is weaker than $CC(G_2)$, then $G_1$ is less restrictive than $G_2$. However, if $G_1$ is less restrictive than $G_2$, it is still possible that $CC(G_1)$ and $CC(G_2)$ are equivalent. For example, norm is less restrictive than lin but normality is equivalent to linearizability as shown in [59] and [51]. It is quite easy to see that lin is the most restrictive order generator which implies that linearizability is the strongest consistency condition.

Figure 4.1 provides an illustration of relations between various consistency conditions and their orders for the given history $H$ which is depicted in the left top corner. The filled circle represents “the universe” for considering consistency conditions; it contains all serializations of $H$. Some of those are legal while others are not; the bold curve in the middle of the circle represents the “border” of legality. The darkest sector of the circle contains those serializations that preserve $\rightarrow H$, which is itself contained in the set of serializations that preserve $\rightarrow H$. The depicted serializations $S_1$, $S_2$ and $S_3$ are members of these three sets, respectively. The white sector of the circle contains all other serializations, i.e., the serializations that preserve none of the above orders.
Figure 4.1: Various orders and serializations for the given history $H$.

Observe that there are legal serializations that preserve $\text{proc}_H$ (e.g., $S_3$) but all serializations that preserve $\text{norm}_H$ are not legal. Thus, the given $H$ is sequentially consistent but not normal or linearizable. This actually agrees with the known fact that sequential consistency (i.e., $CC(\text{proc})$) is weaker than normality and linearizability.

If $G_1$ imposes fewer constraints than $G_2$ but $CC(G_1)$ is equivalent to $CC(G_2)$, it might be easier to verify that a given implementation satisfies $CC(G_1)$ than $CC(G_2)$. For example, normality was introduced in [51] as a condition equivalent to linearizability but with less restrictive order generator. This work also suggests an interesting and important problem of finding analogs equivalent to commonly used consistency conditions which are as least restrictive as possible.

A general solution to this problem is complicated because different order generators might be incomparable and because not only the number of constraints is important but also the form in which they are expressed; the latter should be suitable for verification purposes. Such a form might greatly differ for various object type systems. However, considering the problem without assuming a specific type system in advance may lead to finding criteria that a candidate condition needs to satisfy in order to be equivalent to a particular well-known consistency condition. Such criteria might be easily verifiable for many commonly used type systems.

In this thesis, we propose a criterion that a consistency condition has to meet in order to be equivalent to linearizability. This criterion is based on the notion of non-legality preservation: A consistency order generator $G$ preserves non-legality if for any sequential history $S$ that is not legal, there is no legal serialization that preserves $\sim_S^G$.

**Theorem 4.2.2:** A consistency condition $CC(G)$ is equivalent to linearizability iff $G$ preserves non-legality.

**Proof:**
Assume by way of contradiction that $G$ does not preserve non- legality, i.e., there is a non-legal sequential history $S$ such that there is a legal serialization $S'$ of $S$ that preserves $\sim_S$, yet $CC(G)$ is equivalent to linearizability. The only serialization of $S$ that preserves $<_S$ is $S$ itself; thus $S$ is not linearizable. On the other hand, $S$ satisfies $CC(G)$ since it has a legal serialization $S'$. A contradiction.

As explained above, no consistency condition is stronger than linearizability. Thus, it is sufficient to show that $CC(G)$ is not weaker than linearizability. Suppose it is. Then, there is a history $H$ such that there is a legal serialization $S'$ of $H$ that preserves $\sim_{H}$, but there is no legal serialization of $H$ that preserves $<_H$. Let us take a serialization $S$ of $H$ which preserves $<_H$ and in which each pair of operations unordered by $<_H$ appear in the same order as in $S'$; $S$ is not legal by assumption. We now prove that $S'$ preserves $\sim_S$; this will conclude the proof by showing a contradiction to the fact that $G$ preserves non- legality. Suppose, not. Then there is a pair of operations ordered differently in $S$ and $S'$. According to the way in which $S$ is constructed, these operations are necessarily ordered by $<_H$. Thus, these operations appear in the same order in $H$ and $S$. By the definitions of a consistency order generator and a consistency ordering, if the order of these operations in $S'$ violates $\sim_S$, it also violates $\sim_H$. A contradiction.

Observe that the proof of the “only if” direction does not exploit any properties of ordering. As a matter of fact, we could have defined a general consistency condition in a more general way based on an existence of a legal serialization $S$ of the given history $H$ so that $S$ satisfies some predicate $P(H, S)$. Then the “only if” direction of Theorem 4.2.2 would hold for such general consistency conditions as well.

The non- legality preservation criterion may be easily verifiable for individual object type systems. For example, let us consider read-write objects with usual semantics. An order generator $G'$ that imposes an order on two operations if they are on the same object and at least one operation is write, preserves non- legality. Thus, $CC(G')$ (which is similar to conflict equivalence [98], a correctness condition used in many works on databases) is equivalent to linearizability. $G'$ is less restrictive than norm, and $CC(G')$ is more easily expressed than normality, thus being simpler to verify.

4.2.4 Local Consistency Conditions

A consistency condition $CC(G)$ is local if a history $H$ satisfies $CC(G)$ iff for each object $X$, $H|X$ satisfies $CC(G)$. As the following lemma shows, only the “if” part is interesting because the “only if” direction always holds no matter what the consistency condition is.

**Lemma 4.2.3:** If a history $H$ satisfies a consistency condition $CC(G)$, then for each object $X$, $H|X$ satisfies $CC(G)$.

**Proof:** Let us consider a legal serialization $S$ of $H$ that preserves $\sim_H$. By the definition of legality, $S|X$ is legal for each object $X$. Furthermore, for each object $X$, $S|X$ preserves $\sim_{H|X}$ by the definition of consistency order. ■

As we explained in the introduction to this thesis, locality is a desired property of a consistency conditions ([49], [51] and [59] also discuss motivation for local consistency conditions). Herlihy and Wing [59] showed that linearizability is a local condition. This obviously
implies that every condition equivalent to linearizability is local. Furthermore, it is easy to
see that if a consistency order generator \( G \) is less or equally restrictive compared to \( \text{obj} \), then
\( CC(G) \) is local. Thus, both the strongest consistency condition and very weak consistency
conditions are local. The following theorem minimizes the uncertainty in the remaining gap:

**Theorem 4.2.4:** If a consistency condition \( CC(G) \) is equivalent to or stronger than sequen-
tial consistency but weaker than linearizability, then it is not local.

**Proof:** Assume that such consistency condition \( CC(G) \) exists and it is local. In order to
prove the theorem, we introduce the notion of separating sequence. A separating sequence
is a finite single object sequential history \( Sep \) containing at least two operations such that
\( Sep \) is legal but no serialization of \( Sep \) in which the last operation of \( Sep \) precedes the first
operation of \( Sep \) is legal.

**Lemma 4.2.5:** For an object type as defined in Section 4.2.1, a separating sequence exists.

**Lemma Proof:** Let us take a multiple return value operation \( op \) (such an operation must
exist by Assumption 4.2.3). By definition, there exists an input value \( Iv \) of \( op \) such that
there are two different output values \( Ov_1 \) and \( Ov_2 \) that are legal for \( op \) with \( Iv \). Look at
some history \( H \) consisting only of an invocation of \( op \) with the input value \( Iv \). According to
Assumption 4.2.2, at least one of \( Ov_1 \) and \( Ov_2 \) is not legal for \( resp(op) \) in \( H \). Assume w.l.o.g.
that \( Ov_1 \) is not legal for \( resp(op) \) in \( H \). Let us look at the shortest legal sequential history
\( S \) which contains an invocation of \( op \) with the input value \( Iv \) and output value \( Ov_1 \) (if there
are several histories of the same length, we can take any one of them). \( S \) is finite and it ends
with \( resp(op) \) because the set of legal histories is prefix closed. Furthermore, it follows from
the reasoning above that \( S \) contains at least two operations. Finally, any serialization of \( S \n
in which the last operation of \( S \) precedes the first operation of \( S \) is not legal. Therefore, \( S \) is
a separating sequence. ■

We now proceed with the proof of the theorem. Since \( CC(G) \) is weaker than linearizabil-
ity, it does not preserve non-legality by Theorem 4.2.2. Thus, there is a non-legal sequential
history \( S = \{ \text{inv}(o_1), \text{resp}(o_1), \text{inv}(o_2), \text{resp}(o_2), \ldots \} \) and its legal serialization \( S' \) such that
\( S' \) preserves \( \sim_S \) (we can assume w.l.o.g. that \( S \) is a single object history). We now construct
a new history \( H \) by expanding \( S \) as follows: for each pair of consecutive operations \( o_i, o_{i+1} \)
in \( S \) we introduce a new object instance \( X_i \) and take a separating sequence \( Sep_i \) for \( X_i \) such
that the first event in the sequence occurs at \( pr(o_i) \) and the last event occurs at \( pr(o_{i+1}) \)
(recall that we can replace process names of operations in a legal sequential history according
to Assumption 4.2.1). \( Sep_i \) is inserted between \( resp(o_i) \) and \( inv(o_{i+1}) \).

It is easy to see that \( H|X \) satisfies \( CC(G) \) for each object \( X \). We are going to prove that
\( H \) does not satisfy sequential consistency, i.e., \( CC(proc) \). This will imply that \( H \) does not
satisfy \( CC(G) \) thus showing a contradiction to the fact that \( CC(G) \) is local.

Suppose there is a legal serialization \( S'' \) of \( H \) which preserves \( \preceq_{H} \). Then there is a
pair \( o_k, o_{k+1} \) of consecutive operations in \( S \) whose order in \( S'' \) is reversed. Let us look at
the separating sequence \( Sep_k \) which appears between \( resp(o_k) \) and \( inv(o_{k+1}) \) in \( H \). Since
\( S'' \) preserves \( \preceq_{H} \), the last operation of \( Sep_k \) precedes the first operation of \( Sep_k \) in \( S'' \).
Therefore \( S''|X_k \) is not legal by the definition of separating sequence. ■

32
Note that in order to build a counterexample to locality of $CC(G)$, construction in the proof may introduce many new object instances. However, in many systems that exist in practice, locality violation can be observed with just few objects because the shortest non-legal sequential history $S$ whose legal serialization satisfies $\mathcal{S}S$ is usually very short and because the same object instance can be "reused" for multiple separating sequences.

[59] established that sequential consistency is not local for a particular object type (the Enq-Deq type system for FIFO queues). One important corollary from Theorem 4.2.4 is that sequential consistency is not local for any object type system with the assumptions of Section 4.2.1. However, this theorem is more general: For example, in Section 4.2.2 we discussed several non-standard consistency conditions that can be useful for specific systems (such as the system with strong queries or the system with a master process). Theorem 4.2.4 establishes that all those conditions are not local.

### 4.3 Weak Consistency Conditions

In this section we weaken the definition of a consistency condition in order to consider consistency conditions that do not imply existence of a single legal serialization; PRAM and causal memory belong to this category. Since these conditions are defined only for read-write objects, we restrict our attention to this type system. We will further denote $Hw$ the subsequence of all events in $H$ which are invocations or responses of write operations; $H[w]$ is the subsequence of all events in $H$ which are either in $Hw$ or in $H[w]$.

#### 4.3.1 Serialization Set and Definition of a Weak Consistency Condition

Recall that the definition of a consistency condition in Section 4.2.2 requires the existence of some special legal serialization of the history. This serialization represents the logical order by which all processes view the operations. On the other hand, definitions of weaker conditions, such as PRAM, only require the existence of one special serialization for each process. In other words, such definitions allow each process to view the operations in a different logical order. In order to relate these definitions in a single framework, we introduce the following definition:

**Definition 4.3.1 (A):** serialization set of $H$ is a set of serializations of $H[p_i + w]$ one for each process $p_i$.

A serialization set is legal if all the serializations it contains are legal. We now define that a history preserves a consistency condition by requiring the existence of a serialization set in which all serializations must obey certain ordering restrictions. Yet, the restrictions for different serializations in a serialization set may be different. This sets the framework for defining very weak consistency conditions, in which the view of each process may be different in a very fundamental way. This framework allows us to reason about locality of such consistency conditions. In addition, it proves useful for defining composable consistency conditions in Section 4.4.

While the definitions of a consistency order generator and a consistency order remain the same as in the previous sections, the definition of a consistency condition is modified as follows: Similarly to a consistency order generator, let total order constraint be a boolean
predicate taking two processes, two objects, two operation names, two input values and two output values. Essentially, total order constraint determines for a pair of write operations in a history $H$ whether they should appear in the same order in all the serializations of the serialization set of $H$. Formally, a serialization set $SS$ of a history $H$ satisfies a total order constraint $TO$ if for every pair of write operations $e$ and $f$ in $H$ such that $TO(pr(e), pr(f), obj(e), obj(f), name(e), name(f), ival(e), ival(f), oval(e), oval(f))$ and for every pair $S$ and $S'$ of serializations in $SS$, $f <_S e \implies f <_{S'} e$.

$H$ satisfies a weak consistency condition $WCC(G, TO)$ if there exists a legal serialization set $SS$ of $H$ such that $SS$ satisfies $TO$ and each serialization in $SS$ preserves $\cong_H$. This definition enables us to express virtually all commonly used weak consistency conditions. For example, by setting the $TO$ predicate to false, we can express PRAM and causal consistency: $H$ is PRAM if there is a legal serialization set $SS$ of $H$ such that each serialization in $S$ preserves $\cong_H$. Causal consistency can be rephrased as a weak consistency condition based on the following order generator: $pr(e) = pr(f) \lor name(e) = write \land name(f) = read \land oval(e) = ival(f)$. Furthermore, cache consistency is $WCC(proc, obj(e) = obj(f))$, whereas the results of Theorem 4.4.4 below imply that sequential consistency is equivalent to $WCC(proc, true)$. Finally, it is also possible to express the weak consistency condition introduced in [41]. Like release consistency, this condition is based on synchronization variables but it does not imply any form of atomicity.

### 4.3.2 Locality of Weak Consistency Conditions

We now present a complementary result to Theorem 4.2.4 for read-write objects. As special cases, this theorem implies that PRAM, causal and cache consistencies are not local.

**Theorem 4.3.1:** If a consistency condition $WCC(G, TO)$ is equivalent to or stronger than PRAM but equivalent to or weaker than sequential consistency, then $WCC(G, TO)$ is not local.

**Proof:** For proving this theorem, we need a weaker definition of non-legality preservation: A consistency order generator $G$ weakly preserves non-legality if for any sequential history $S$ such that $S$ is not legal and all read operations in $S$ occurred on the same process, there is no legal serialization that preserves $\cong^*_S$.

**Lemma 4.3.2:** If a consistency condition $WCC(G, TO)$ is equivalent to or weaker than sequential consistency, then $G$ does not weakly preserve non-legality.

**Lemma Proof:** Assume by way of contradiction that there is such a consistency condition $WCC(G, TO)$, and $G$ weakly preserves non-legality. Let us consider the following sequential history $S = \{R(X) = 1 \text{ at } p, W(X, 1) \text{ at } q\}$. Since $G$ weakly preserves non-legality, there is no legal serialization of $S$ that preserves $\cong^*_S$. Therefore, there is no legal serialization set of $S$ in which the serialization of $p$ preserves $\cong^*_S$. Thus, $S$ does not satisfy $WCC(G, TO)$. However, $S$ obviously satisfies sequential consistency. A contradiction.

Let us assume by contradiction that there exists a local consistency condition $WCC(G, TO)$ that is equivalent to or stronger than PRAM but equivalent to or weaker than sequential consistency. By the lemma above, there exists a non-legal sequential history
$S$ such that all read operations in $S$ occurred on the same process (denote it $p$), and a legal serialization of $S$ that preserves $\simeq_S$. Again, we can assume w.l.o.g. that $S$ is a single object history.

We now construct a new history $H$ by expanding $S$. For each pair of consecutive operations $o_i, o_{i+1}$ in $S$ we introduce a new object instance $X_i$ and a sequence $Sep_i$ of the following form: $\{W(X_i,1) \text{ at } pr(o_i), R(X_i) = 1 \text{ at } p, W(X_i,2) \text{ at } pr(o_{i+1}), R(X_i) = 2 \text{ at } p\}$. $Sep_i$ is inserted between $resp(o_i)$ and $inv(o_{i+1})$.

**Lemma 4.3.3:** Each $Sep_i$ is a legal sequential history. However, there is no legal serialization of $Sep_i$ preserving $\simeq^{proc}_{\prec} Sep_i$, in which the last write operation of $Sep_i$ precedes the first write operation of $Sep_i$.

**Lemma Proof:** The lemma directly follows from $Sep_i$ construction. ■

Observe that for each object $X$, there is a legal serialization of $H|X$ that preserves $\simeq^{H|X}_{\prec}$. This is because each $Sep_i$ is legal by itself, and there exists a legal serialization of $S$ that preserves $\simeq_S$. Furthermore, note that in any serialization set of $H$, the serialization of $p$ consists of the same set of operations as $H$ itself while all other serializations contain only write operations. Thus, a serialization set $SS$ of $H$ is legal iff the serialization of $p$ in $SS$ is legal. The same is true for $H|X$, for any object $X$. Therefore, it is easy to show that $H|X$ satisfies $WCC(G,TO)$ for each object $X$. The required serialization set $SS$ can be constructed as follows: the serialization $SS_p$ of $p$ can be a legal serialization of $H|X$ while all other serializations will consist of the write operations of $H|X$ ordered as in $SS_p$.

We are going to prove that $H$ does not satisfy PRAM, i.e., $WCC(proc, false)$. This will imply that $H$ does not satisfy $WCC(G,TO)$, thus showing a contradiction to the fact that $WCC(G,TO)$ is local.

Suppose there is a legal serialization set $SS'$ of $H$ such that each serialization in $SS'$ preserves $\simeq^{H}_{\prec}$. Let us consider the serialization $S'$ of $p$ in $SS'$. By the above, $S'$ consists of the same operations as $H$ itself. Since $S'$ is legal but $S$ is not, there is a pair $o_k, o_{k+1}$ of consecutive operations in $S$ whose order in $S'$ is reversed. Let us look at the separating sequence $Sep_k$ that appears between $resp(o_k)$ and $inv(o_{k+1})$ in $H$. Since $S'$ preserves $\simeq^{H}_{\prec}$, the last write operation of $Sep_k$ precedes the first write operation of $Sep_k$ in $S'$. Therefore $S'|X_k$ is not legal by Lemma 4.3.3. ■

### 4.4 Composable Consistency Conditions for Wide-Area Distributed Services

#### 4.4.1 Composability Framework

In this section, we define a formal framework that allows us to reason about composing consistency guarantees using elementary consistency conditions as basic building blocks. We start by defining the most basic consistency condition that is required in order for shared objects to be a meaningful tool for communication between processes: every write operation by any process must be seen by all other processes. This is captured by the following property:
**Eventual Propagation:** For every process $p_i$ and history $H$, there exists a legal serialization $S_{p_i}$ of $H|p_i + w$.

This requirement essentially expresses liveness of update propagation: For a given history and a given write operation in this history, if some process invokes an infinite number of queries, it will eventually see the result of this write. We therefore assume that Eventual Propagation holds for all histories considered in the rest of this thesis. Furthermore, Eventual Propagation guarantees that at least one legal serialization set exists for any given history, thus laying the ground for further description of the framework.

The definition of weak consistency conditions in Section 4.3.1 stipulates that every serialization in a serialization set preserve the required ordering. However, it would be useful for many distributed applications to use *per-session guarantees* in which ordering is imposed on individual serializations of a serialization set. The motivation for using such guarantees can be found, e.g., in [118]. To allow for per-session conditions, we say that for a given history $H$ and a process $p_i$, a serialization set $S = \{S_{p_i}\}$ preserves a condition $X$ for the local history $H|p_i$ if $S_{p_i}$ satisfies the ordering required by $X$. When all serializations in the set satisfy this ordering (as required by the definition in Section 4.3.1), we say that $S$ globally preserves $X$.

For a given history $H$, a serialization set $S = \{S_{p_i}\}$ globally preserves some set of consistency conditions (i.e., a combination of one or more conditions) if $S$ globally preserves each condition in this set. Now we can state the main idea of the composability framework: a history $H$ is consistent with respect to a condition set $X$ if there exists a legal serialization set $S$ of $H$ such that $S$ globally preserves $X$. Note that this is stronger than the fact that $H$ is consistent with respect to each individual condition in the set separately because the same single serialization set must preserve all conditions in the combination. Finally, we say that an implementation $A$ obeys a condition set (i.e., a combination of one or more conditions) $X$ if every history generated by $A$ is consistent with respect to $X$.

Since we do not need to discuss different kinds of order generators in this part of the work, we will use a simpler and more standard notation of $o_1 \overset{H}{\rightarrow} o_2$, instead of $o_1 <_H o_2$, to denote that $o_1$ precedes $o_2$ in a history $H$. Furthermore, we will use the following shortcut notation when discussing causal memory: Given a history $H$, an operation $o_1$ directly precedes $o_2$ (denoted $o_1 \overset{H}{\rightarrow} o_2$) if either $o_1 \overset{H}{\rightarrow} o_2$ or $\text{name}(o_1) = \text{WRITE}, \text{name}(o_2) = \text{READ}$, and $o_2$ reads a result written by $o_1$. Let $\rightarrow$ denote the transitive closure of $\overset{H}{\rightarrow}$.

### 4.4.2 Proposed Set of Elementary Consistency Conditions

**Read Your Writes:** For a given history $H$ and a process $p_k$, a serialization set $S = \{S_{p_k}\}$ preserves *Read Your Writes for the local history $H|p_k$* if for every two operations $o_1$ and $o_2$ in $H|p_k$ such that $\text{name}(o_1) = \text{WRITE}, \text{name}(o_2) = \text{READ}$, and $o_1 \overset{H|p_k}{\rightarrow} o_2$, it holds that $o_1 \overset{S_{p_k}}{\rightarrow} o_2$.

**FIFO of Reads:** For a given history $H$ and a process $p_k$, a serialization set $S = \{S_{p_k}\}$ preserves *FIFO of Reads for the local history $H|p_k$* if for every two operations $o_1$ and $o_2$ in $H|p_k$ such that $\text{name}(o_1) = \text{READ}, \text{name}(o_2) = \text{READ}$, and $o_1 \overset{H|p_k}{\rightarrow} o_2$, it holds that $o_1 \overset{S_{p_k}}{\rightarrow} o_2$.  

36
FIFO of Writes: For a given history $H$ and a process $p_i$, a serialization set $S = \{S_{pj}\}$ preserves FIFO of Writes for the local history $H|p_i$ if for every two operations $o_1$ and $o_2$ in $H|p_i$ such that $name(o_1) = \text{WRITE}$, $name(o_2) = \text{WRITE}$ and $o_1 \xrightarrow{H|p_i} o_2$, holds $\forall j: o_1 \xrightarrow{S_{pj}} o_2$.

Reads Before Writes: For a given history $H$ and a process $p_i$, a serialization set $S = \{S_{pj}\}$ preserves Reads Before Writes for the local history $H|p_i$ if for every two operations $o_1$ and $o_2$ in $H|p_i$ such that $name(o_1) = \text{READ}$, $name(o_2) = \text{WRITE}$, and $o_1 \xrightarrow{H|p_i} o_2$, it holds that $o_1 \xrightarrow{S_{pj}} o_2$.

Local Causality:\footnote{This is similar to the condition called \textit{Writes Follow Reads} in \cite{118}.} For a given history $H$ and a process $p_i$, a serialization set $S = \{S_{pj}\}$ preserves Local Causality for the local history $H|p_i$ if for every three operations $o_1$, $o_2$ and $o_3$ such that $o_2$ and $o_3$ are in $H|p_i$, $name(o_1) = \text{WRITE}$, $name(o_2) = \text{READ}$, $o_3 = \text{WRITE}$, $o_2$ reads a result written by $o_1$ and $o_2 \xrightarrow{H|p_i} o_3$, it holds that $\forall j: o_1 \xrightarrow{S_{pj}} o_3$.

As noticed in \cite{118}, Read Your Writes, FIFO of Reads and Reads Before Writes only affect the local histories for which they are provided. On the other hand, Local Causality and FIFO of Writes contain guarantees w.r.t. the local histories of other processes.

Total Order: For a given history $H$, a serialization set $S = \{S_{pj}\}$ globally preserves Total Order if for every two serializations $S_{pi}$ and $S_{pj}$ in $S$, $S_{pi} \xrightarrow{w} S_{pj} \xrightarrow{w}$.

Unlike other elementary conditions, it does not make sense to define a per-session Total Order. In fact, this condition differs from all the others: While all other elementary conditions can be expressed as $WCC(G, false)$ for different consistency order generators $G$, the Total Order property is equivalent to $WCC(false, true)$.

Any single condition that relates two events of the same type is trivial by itself. For example, if we only require FIFO of Reads, then naturally we can always find legal serializations in which all reads are ordered in FIFO order. This is because we have not placed any requirements on writes, and therefore we have the freedom to order the writes in the serialization so all the reads are legal. This applies similarly to FIFO of Writes, Local Causality and Total Order. Thus, these guarantees become meaningful only in combinations that contain several guarantees of different types. The only guarantees that are not trivial by themselves are Read Your Writes and Reads Before Writes.

4.4.3 Examples of Useful Compositions

Expressing Standard Consistency Conditions

We now present several theorems that show how some combinations of the basic consistency conditions relate to each other and to other known consistency conditions.

**Theorem 4.4.1:** Any history that is consistent w.r.t. Total Order and Reads Before Writes is also consistent w.r.t. Local Causality.
Proof: Assume, by way of contradiction, that there is a history $H$, which is consistent w.r.t. Total Order and Reads Before Writes, but is not consistent w.r.t. Local Causality. Thus, there are three operations $name(o_1) = \text{write}$, $name(o_2) = \text{read}$, and $name(o_3) = \text{write}$, such that $o_1$ was issued by $p_i$, $o_2$ and $o_3$ issued by $p_j$, and $o_2$ reads the result of $o_1$, and $o_2 \rightarrow o_3$, yet there is a serialization set $S$ of $H$ that globally preserves Total Order and Reads Before Writes in which $o_3 \rightarrow o_1$ for some process $p_k$. By assumption, for the serialization $s_{p_j} \in S$, $o_2 \rightarrow o_3$, and $o_1 \rightarrow o_2$. However, since $S$ obeys Total Order, $\forall k$: $o_1 \rightarrow o_3$. A contradiction. ■

The order in which operations of some process $p_i$ appear in $H|p_i$ is also known in the literature as process order. The conditions FIFO of Writes, FIFO of Reads, Read Your Writes, and Reads Before Writes can be seen as limitations of process order to the corresponding operations. For example, FIFO of reads only requires preserving process order for reads. Given this observation, the following theorem is not surprising.

Theorem 4.4.2: Any history that is consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes is also PRAM consistent.

Proof: We need to show that there is a serialization set such that in each serialization $S_{p_j}, H|p_j = S_{p_j}|p_j$, and all write operations of any other process $p_i$ are ordered in the same order as in $H|p_i$. Consider a given serialization set $S$ that is consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes. Fix any serialization $S_{p_j} \in S$, and any process $p_i$. By FIFO of writes, every two write operations of $p_i$ are ordered in $S_{p_j}$ in the same order as in $H|p_i$. Thus, we only need to show that every pair of operations by $p_j$ of which at least one is a read are ordered in $S_{p_j}$ according to their order in $H|p_j$. However, this trivially holds for $S_{p_j}$ due to FIFO of Reads, Read Your Writes and Reads Before Writes. ■

Theorem 4.4.3: Any history that is PRAM consistent and is consistent w.r.t. Local Causality is also causally consistent.

Proof: Let $H$ be a history that is PRAM consistent and consistent w.r.t. Local Causality, and let $S$ be the serialization set that obeys all the requirements of PRAM and Local Causality. We claim that $S$ also obeys the requirements of the serialization set of causal consistency. Assume, by way of contradiction, that it does not. Thus, there must exist at least one serialization $S_{p_i}$ in $S$ for which there are two operations $o_1$ and $o_2$ such that $o_1 \rightarrow o_2$, but $o_2 \rightarrow o_1$. By the PRAM guarantees, $o_1$ and $o_2$ are operations of two different processes $p_j$ and $p_k$, such that $k \neq j, k \neq i$, and $j \neq i$. By the definition of $S_{p_i}$, both $o_1$ and $o_2$ are write operations. Given that $o_1 \rightarrow o_2$, there is a sequence of operations $o_{p_1}, \ldots, o_{p_t}$ such that $o_1 = o_{p_1}, o_2 = o_{p_t}$, and $\forall q: 0 \leq q \leq \frac{t}{2}$, $name(o_{p_2-q+1}) = \text{write}, name(o_{p_2-q}) = \text{read}, o_{p_2-q}$ reads the result of $o_{p_2-q-1}$, and both $o_{p_2-q}$ and $o_{p_2-q+1}$ occur in the same process and in that order. Thus, for each couple of writes $o_{p_2-q}$ and $o_{p_2-q+2}$, Local Causality guarantees that they are ordered in this order on $S_{p_i}$. By transitivity, $o_1 \rightarrow o_2$. A contradiction. ■
**Theorem 4.4.4:**
Any history that is PRAM consistent and is consistent w.r.t. Total Order is also sequentially consistent.

**Proof:** Let $H$ be a history that is PRAM consistent and consistent w.r.t. Total Order. Thus, there exists a serialization set $SS$ of $H$ that obeys both the requirements of PRAM and Total Order. We now show how to construct a legal serialization $S$ of $H$ such that for every process $p_i$, $H|_{p_i} = S|_{p_i}$. Due to Total Order, all the writes are ordered in the same order in all serializations in $SS$. Thus, we start by creating a serialization $S$ of all writes in $H$ ordered in the order they appear in all serializations. Next, we extend $S$ by adding the read operations in the following manner, performed iteratively for all processes $p_i \in \{p_1, \ldots, p_n\}$: for each two write operations $o_1$ and $o_2$, we add all read operations by $p_i$ (if any exist) that were ordered between $o_1$ and $o_2$ in $S_0$ and order them in $S$ between $o_1$ and $o_2$. Also, if there are already some read operations by other processes between $o_1$ and $o_2$, we place the reads of $p_i$ immediately after $o_1$. In a similar manner, we add to $S$ all reads that are placed in any $S_{p_i}$ before the first write, and place them before the first write in $S$, and add all reads that are placed in any $S_{p_i}$ after the last write, and place them after the last write in $S$. Note that $S$ now includes all operations of $H$, and is thus a serialization of $H$. Moreover, since all the operations of each process $p_i$ are ordered in $S$ in the same relative order as in $S_{p_i}$, they are also ordered in the same order as in $H|_{p_i}$. Finally, since each read by any process $p_i$ is placed in $S$ between the same writes it was placed in $S_{p_i}$, and since $S_{p_i}$ is legal, then $S$ is also legal. Thus, $S$ obeys all the requirements of sequential consistency, and hence, $H$ is sequentially consistent. 

**Other Useful Combinations**

When an applications obey a known programming convention, it is often possible to run it on top of an implementation that provides a weak consistency condition, yet the result will be as if the application was run with a strong condition. The most prominent example of this is that executions of data-race-free programs on a release consistent distributed shared memory are in fact sequentially consistent [8]. Similarly, by exploiting the semantics of the application and specific operations, it may be possible to obtain meaningful correct behavior with weak ordering guarantees, as proposed in [71]. The benefit of this is that since the implementation only guarantees a weak condition, it can be implemented more efficiently. Below we give a couple of examples that demonstrate the usefulness of the basic conditions, even in combinations that involve only a few of them, and perhaps even provide different guarantees to each serialization.

Consider an application in which there is only a single writer. In this case, it is enough to require FIFO of Writes, and the result will be as if Total Ordering was used. In particular, when there is a single writer, PRAM is equivalent to sequential consistency. However, the equivalence of FIFO of Writes to Total Ordering might be useful even on weaker combinations than PRAM.

Another interesting application that can benefit from a condition that is weaker than PRAM is a bulletin board. Here, a client is only interested in other client’s postings, and thus does not need Read Your Writes.
As a final example, consider an application in which there are several simple clients and a few supervisor clients. Each simple client reads and writes to different objects than the other simple clients. Yet, supervisors can read all objects. In this case, the serializations of simple clients should obey FIFO of Writes and Read Your Writes, while supervisors need FIFO of Reads.
Chapter 5

Implementation and Algorithms

5.1 Hierarchy Construction

A hierarchy construction is started when a client calls \texttt{register\_object} in order to create a new hierarchy for the object (or for the group of objects). This call registers the object (or the group of objects) with the local client’s DCS, which becomes the root of the new hierarchy. In the following description we call it a \textit{root DCS} for this hierarchy. It keeps the knowledge of the whole hierarchy and is responsible for the hierarchy construction. The root DCS also plays a special role in achieving consistency among the cached copies of the object, as we explain in Section 5.3. In addition, the root DCS registers itself with the naming service as a caching service provider for its cached object (or the group of objects).

Figure 5.1 shows the sequence of operations executed when a new DCS joins the hierarchy. When a client wishes to start working with a cached object, it calls \texttt{copy\_object} on its local DCS. Unless the local DCS has this object already cached, it finds the root DCS for this object with the aid of a location service. Then it contacts the root DCS with a request to join the hierarchy for this object. This request also contains the domain of the new DCS.

When the root DCS receives such a join request from a DCS of domain $D$, it finds a domain $D'$ in the existing hierarchy such that $D'$ is the closest to $D$. If the hierarchy includes several domains at about the same distance from $D$, we chose the domain whose node is at the highest level in the hierarchy and has fewer child nodes in order to create more flat and balanced hierarchies. Then, the root DCS sends a reply specifying the location of $D'$s DCS to the local client’s DCS. Upon receiving this reply, the latter sends a request to its designated father node in order to register as its son in the hierarchy and in order to obtain a cached copy of the object. The father node registers the new child and sends its cached copy, thus, completing the join protocol.

The hierarchy construction protocol guarantees that for each client there is a local DCS that has a cached copy of the object; the local DCS handles the client’s requests. It is this fact and the hierarchical architecture of the system that allow to significantly reduce the response time, to distribute the load on DCSs and to render the caching service scalable.

To explain the process of hierarchy construction in greater detail, we need to elaborate on several issues. First of all, the root of the hierarchy should be able to determine which domain in the hierarchy is the closest to the domain of the joining DCS. It should be noted that the notion of a distance between different domains is an interesting and non-trivial question
that arises in many different contexts. For example, many works on peer-to-peer computing choose the number of network hops as the major metric. However, the primary goal of a caching service is to minimize latencies. Furthermore, the triangle inequality does not hold in the Internet as shown by our work and a number of previous studies. In particular, in many cases it makes sense for a server to delegate all communication to another nearby server that has a better link to the outside world. Therefore, we chose a direct latency value as the distance metric.

In some cases, the distance between domains can be estimated in a simple way by considering IP addresses or symbolic domain names. For example, if a DCS attempts to join the hierarchy and this hierarchy already includes another DCS in the same network or in the same small domain of symbolic names (e.g., the domain of academic institutions in Israel \( .ac.il \)), then it will make sense to attach the second server to the hierarchy in proximity of the first one. However, the general solution to this problem has no alternative but to run some kind of topology discovery protocols. In CASCADE, we are currently running simple round trip time measurements but incorporating more complex protocols is part of our future work (see Section 8.3.1). In would be also interesting to take available bandwidth on the inter-DCS links into account in addition to latencies when constructing hierarchies.
Another important issue is how we find the root of the hierarchy for a given object. First of all, this requires some kind of a unique global object identifier that is used as a search key for locating the object hierarchy and its root. In many cases, such an identifier existed in the original CORBA application prior to introducing caching: it is under this identifier that the service provider was advertising its service and service clients were obtaining an object reference, e.g., through a Naming Service\(^1\). To port this scheme to CASCADE, the service provider needs to pass this object identifier when registering the object with CASCADE. The root of the hierarchy registers itself under this name in some location service. When an application client invokes a \texttt{copy\_object} request on its local DCS, it has to specify the identifier of the object whose copy it requests to put into the cache. The local DCS uses the location service to translate the service identifier into the root of the hierarchy.

Thus, we need a location service suitable for our purposes, that is, a scalable and highly available one because CASCADE servers can be distributed all over the world. Building such a location service is a fundamental problem that arises in many other contexts such as mobile computing. Many solutions have been already developed so it makes sense to use an existing implementation rather than to implement another one for our caching service. Currently, we are in the process of integrating the location service of \cite{149} with CASCADE. This service appears particularly appealing because of its highly scalable hierarchical architecture. Furthermore, this service has nice locality awareness: a resolution request is handled faster if it originates from a domain that is close to the domain where the object was registered.

Yet another issue the hierarchy construction algorithm has to deal with is how to prevent the root of the hierarchy from being overloaded with join requests. It should be noted that this problem is not very severe because join requests are not as frequent as, e.g., requests for method execution. A typical technique for addressing this problem is replicating the root of the hierarchy itself, that is, using a cluster of computers instead of a single host at the root node. We also use a slightly more complex solution than the construction protocol described above in order to minimize the amount of information the root needs to keep. The idea is to distribute the knowledge of the hierarchy among multiple hierarchy nodes. Specifically, the root of the hierarchy knows the top \(n\) levels of the hierarchy and each node at the \(n\)-th level is responsible for its own sub-branch of the hierarchy. When a join request arrives, the root DCS determines which of the tree branches is the closest to the domain that the request comes from. Then, it redirects the requesting DCS to one of the servers at the \(n\)-th level, and the latter decides upon the exact place in its sub-branch where the requesting DCS should join the hierarchy.

Since we do not know in advance what DCSs are going to join the hierarchy, the dynamic process of hierarchy construction may result in non-optimal hierarchies. Therefore, it would be useful to reconstruct already existing hierarchies to make them more balanced and reduce the distances between the nodes. Dynamic hierarchy reconstruction would also help to cope with failed nodes, dynamic changes in the network topology, uneven distribution of load etc. Various techniques for such dynamic reconstruction could be borrowed, e.g., from

\(^1\)It was possible that the original application was using a different scheme, e.g., a Trading Service that locates a service by properties instead of a global identifier. However, it would be infeasible for CASCADE to support all existing ways of identifying a service. Furthermore, it is reasonable to assume that the service provider can assign its service a widely known global name. For example, developers of Java classes conventionally give them the name based on the title of their company.
peer-to-peer and publish-subscribe platforms (see Section 8.3.1). Unfortunately, it appears very challenging to preserve semantic guarantees, e.g., of consistency conditions in face of dynamic changes in a hierarchy. Furthermore, CASCADE has a complex thread and send buffer management (see Sections 5.7.2 and 5.8). Making a synchronized snapshot of various threads, buffer states, and messages in transition between the nodes of the hierarchy not only leads to very complex and elaborate distributed algorithms but it also requires to introduce a lot of Java monitors. Contention on those monitors would have a noticeable negative impact on the service processing time even when no hierarchy reconstruction is required.

Disconnection from the hierarchy is currently supported only for leaves. If some object is no longer used, and some intermediate DCS node wants to leave the hierarchy for this object, this DCS should wait for its children to disconnect first.

When a leaf DCS node $D$ wishes to disconnect from the object hierarchy, it informs the root about this, so that the root updates its knowledge about the hierarchy. Then $D$ sends its parent a request to detach it from the hierarchy. Only when $D$ receives a reply, it can safely disconnect.

It should be emphasized that CASCADE takes care of all synchronization problems that arise during joining and leaving the hierarchy (see also Section 5.8). CASCADE also supports an unconditional hierarchy destruction. Once a request for a hierarchy destruction is invoked on one of the DCSs, it is propagated to all hierarchy nodes and every node removes the object copy along with the state of policy implementation, hierarchy knowledge and other meta-information about the object (though some accounting information about the object can be saved onto the disk for future use as explained in Section 5.6.2). It is the responsibility of the application to make sure that the object is no longer in use prior to invoking a hierarchy destruction request.

### 5.2 Failure Model and Persistence

There is a well known tradeoff between achieving better performance and greater resilience to failures in distributed applications. In particular, most high performance systems such as distributed shared memory implementations do not consider any type of failures. On the other hand, services that are highly fault tolerant (e.g., services that handle byzantine failures) have to pay a high price in performance and scalability compared with less resilient services.

Since CASCADE operates in a hostile WAN environment it has to be fault tolerant to some extent. However, CASCADE is mostly intended for efficient electronic information services and collaborative systems (see Section 6) rather than mission critical applications. Therefore, currently we consider only benign failures.

While we account for transient communication link failures between DCSs, we assume that all the links are eventually operational. In other words, we do not consider permanent network partitions and disconnected operation of servers. As described in Section 3.5, we use CORBA for internal DCS-to-DCS communication. Since the main CORBA protocol is IIOP, which is based on TCP, we enjoy all of the usual TCP guarantees including FIFO and recovery from lost messages within a session. However, if absence of connectivity lasts for some significant time, TCP drops a connection in which case CORBA throws an exception indicating that a remote object is inaccessible. When this occurs, we use the standard
technique of sending the same request periodically until its transfer succeeds. Of course, it may happen that the same request is received twice so we should be able to deal with duplications.

Client crashes can affect only the execution of the failed client. In particular, all policy guarantees such as consistency conditions are preserved in face of client failures. In fact, servers are almost stateless with respect to individual clients. Furthermore, any per-client state data (such as lock information mentioned in Section 5.4) a server may have is deleted after some timeout. This makes the caching service less vulnerable to application programmer’s mistakes.

Server crashes present a more difficult problem for CASCADE. It is particularly challenging to define meaningful semantics when a server fails in the middle of a request execution. Using distributed transactions can help addressing this problem to some extent (see Section 5.4). CASCADE also support object persistence by periodically saving cached objects to a stable storage. Specifically, CASCADE provides a choice between several persistence policies:

**updates-driven**: Upon this policy a cached version of the object is saved each \( n \)-th time it is updated, where \( n \) is a parameter of the policy.

**timer-driven**: Upon this policy a cached version of the object is saved each given period of time, where this period is a parameter of the policy.

**client-driven**: The API of CASCADE allows the application to initiate saving the cache content on disk. Upon this policy, a cached object is saved when the corresponding caching service API method is called.

However, it is insufficient for a server to save only the currently cached objects in order to be able to recover from a crash. The state of a server comprise states of policy implementations, content of numerous queues and buffers, acquired Java monitor etc. Making a full snapshot, or even saving state changes incrementally after every small operation can result in severe performance degradation. Thus, from performance standpoint it is better for the application to not require CASCADE to save its cached content very frequently. Of course, this implies that the most recent changes can be lost when a server crashes.

As explained in the previous section, CASCADE does not currently have a dynamic hierarchy reconstruction mechanism. Therefore, when a server fails, a new DCS should be started on the same computer in order for the hierarchy to continue operation. This can be done either manually or through the CORBA automatic activation mechanism supported by most ORB vendors. When a server is restarted, old objects are read from the disk and their old state is restored. Furthermore, the objects are created in such a way that old object references are still valid\(^2\). Thus, a server crash and restart is transparent for other DCSs in the object hierarchy.

\(^2\)This is achieved by using the CORBA Portable Object Adaptor policy of a user defined object key.
5.3 Implementation of Cached Copies Consistency

Consistency implementation in CASCADE closely follows the model presented in Section 4.4.2. However, we do not discuss the implementation of the Reads Before Writes guarantee in this section: Since the invocation of queries is mainly synchronous in CORBA, the execution of the application is blocked until a result is returned. Thus, a later update cannot affect the returned value. Therefore, Reads Before Writes trivially holds in any natural implementation. It should be noted that for multithreaded clients, each thread of control is treated as a separate process from the standpoint of consistency guarantees. In order words, per-session consistency conditions are applied to each thread separately.

5.3.1 Assumptions on the Application

Before we proceed to describing the implementation of consistency conditions in CASCADE we need to explain the rationale behind choosing a particular way of implementing the guarantees. First of all, we divide all interface methods of a cached object into updates that modify the object state and queries that do not. Updates and queries are handled differently by CASCADE: We assume that the application invokes queries more frequently than updates. Therefore, the implementation in CASCADE propagates update requests to all DCSs in the object hierarchy whereas queries are executed locally, i.e., on the local DCS of the client that invokes the request. This makes executing updates much more expensive compared to queries.

Note that a division of object methods into queries and updates is a semantic rather than a syntactic one: For example, a query can prefetch some data into an object state variable in order to serve further queries more efficiently. While this operation alters the object state, it is still considered a query because it does not make the object “more updated”. Since a caching service has no knowledge about the semantic state of a cached object, it cannot determine that some method is a query by, e.g., parsing its bytecode. If a query method is mistakenly categorized as update, requests for this method execution will be uselessly propagated between the servers wasting bandwidth and other resources. Therefore, we believe that this classification has to be done by the application developer and not by the service. In CASCADE, it is part of the policy specified at the object registration time.

Another observation we make is that in object oriented client-server middlewares, an object state typically takes much more space than method arguments. Therefore, propagating update requests themselves consumes much less bandwidth compared to propagating object states that result from update invocations. When requests are propagated, they need to be executed at every DCS, whereas propagation of states requires object serialization and deserialization operations. For the prevailing majority of applications, the latter operations consume at least as much server computational resources as executing the requests. Therefore, push based update propagation lends itself to such settings better than state invalidation schemes. In other words, it is more efficient to push update requests to all servers in a hierarchy than to retrieve the entire object state later.

However, in certain cases a request execution can be non-deterministic, i.e., it can produce different results each time the same request is invoked. In particular, it may have some side effects. Such a request should be executed exactly once on one of the servers and it is the
resulting state that should be propagated instead of the request. Therefore, CASCADE supports both propagating requests and object states and the application has to specify which propagation should be done as part of the policy.

Finally, while we allow the client application to communicate to and invoke requests on different servers, we assume that normally the client works with the same server (typically, the server of its domain) most of the time. Therefore, the implementation should be optimized for this “static” case while still accounting for the possibility that the client switch to a different server in the future.

5.3.2 Implementation of Eventual Propagation and Total Order

CASCADE always guarantees Eventual Update Propagation while the use of other conditions can be controlled by the application. To guarantee Eventual Update Propagation, queries are always locally executed at the DCS a client communicates to and updates are propagated through the hierarchy. However, the way updates propagate and the order in which they are being applied depend on whether Total Order is required.

If Total Order is not required by the application, Eventual Propagation is implemented as follows: A DCS that receives an update request from a client applies it locally and sends it to all its neighbors in the hierarchy in parallel. A DCS that receives an update request from a neighbor DCS X applies the update and performs flooding, i.e., sends the request to all its neighbors but X. Note that this propagation protocol preserves per-DCS FIFO of updates because all the links are FIFO (as explained in Section 5.2) and because there is only one path in the hierarchy between any pair of nodes. Furthermore, per-DCS Session Causality also holds: If a DCS receives and applies an update, and then some client queries the object state and issues another update at this DCS, then the second update will be broadcast to the neighbors of this DCS after the first one. We will show later in this section how these facts can be exploited in order to provide an efficient implementation of session guarantees.

The Totally Ordered Eventual Propagation (i.e., the Total Order + Eventual Propagation conditions) is implemented as follows: Updates first ascend through the hierarchy towards the root. The root of the hierarchy orders the updates in a sequence, applies them and propagates ordered updates through the hierarchy downwards towards the leaves.

Note that this implementation of Total Order is not affected by presence or absence of application demand for other consistency conditions. Moreover, this implementation is entirely based on the DCS algorithm and inter-DCS protocol, and does not require any client involvement.

Since our goal is to address Internet applications, where extremely long delays are common, we have made the design choice that update requests can return before the update has traversed the entire object hierarchy. The result, however, is that the implementation of session conditions requires client cooperation in most cases.

The need for client cooperation could be eliminated or at least reduced, if an update request (or even a query request) invoked on a DCS by the client returned only after the update (query) traversed the whole object hierarchy. However, this would result in extremely long delays so we do not consider it as a viable solution for a service like CASCADE.

Also, the implementation of the session conditions adapts itself to the set of consistency requirements chosen by the application. In particular, their implementation is significantly
affected by presence or absence of Total Order. Therefore, we discuss their implementation with and without Total Order separately.

5.3.3 Implementing Session Guarantees in Presence of Total Order

The implementation of the session guarantees is greatly simplified by the presence of the Total Order implementation. First, Session Causality is achieved for free, as Theorem 4.4.1 implies. Second, the root of the hierarchy can assign each update a global update identifier that serves as a version number of the object. Hence, an object version can be identified by a single number. As a result, the implementation of session guarantees becomes simpler, less information needs to be stored at both clients and DCSs, and most important, less consistency related data needs to be transferred between a client and a DCS per method invocation.

Specifically, with each query result, a DCS returns to the client the number of the object version this query sees. It would be more complicated to handle updates in a similar way because updates have to be propagated first to the root DCS which assigns them an update identifier. In principle, a DCS that received an update request from a client can block the client until the update identifier is received from the root DCS and then pass this version number to the client. This way, the only consistency information to be transferred between a client and a DCS would be a single global version number. However, since CASCADE is intended to operate in a WAN environment and the propagation latency between a client DCS and a root DCS may be quite significant, this solution may block the client for prohibitively long time.

Therefore, CASCADE adopts an alternative identifier scheme for updates: Each DCS maintains a counter of updates originated at this DCS and each update is assigned a local update identifier consisting of the DCS identifier and a counter value. In contrast to the global version numbers, two local update identifiers assigned by different DCSs are incomparable. When a client invokes an update request on a DCS, the DCS immediately produces a new local update identifier and returns it to the client.

For implementation of some session guarantees we need to maintain a version vector for an object with one entry per DCS in the hierarchy; each entry in this vector corresponds to the last local update identifier received from the corresponding DCS\(^3\). Version vectors are handled in the following way: When an update ascends through the hierarchy towards the root, the local update identifier is piggybacked on the update message. When this update is propagated from the root towards the leaves, its global version number and local update identifier are both piggybacked. Upon receiving and applying this update, a DCS updates its current object version number and version vector.

We now describe the individual implementation of the three session guarantees that require a non-trivial implementation:

**FIFO of Reads:** As previously explained, with each query result, a DCS returns to the client the object’s version number that this query sees. The client passes this number to a (possibly different) DCS upon its next query. This DCS does not apply the query

\(^3\)The technique of using version vectors is extensively described in literature; see, e.g., [81] where version vectors are used for maintaining logical time.
and blocks the client until it receives and applies the update referred to by the version number (in other words, the DCS synchronizes the query with the version number).

**FIFO of Writes:** For implementing FIFO of Writes, the root DCS should maintain a version vector which contains the last local update identifier received from each DCS. Keeping only the last update identifier is sufficient because Total Order preserves per-DCS FIFO of updates: Two updates issued at the same DCS reach the root where they are ordered in the order of their issuance.

As previously explained, when a client invokes an update request on a DCS, the DCS transfers a local update identifier back to the client. The client only remembers the last local identifier it received from some DCS and forgets all previous local identifiers. This is sufficient because FIFO of Writes is a transitive relation and it is enough to remember only the last predecessor.

The client passes the last known local identifier to a (possibly different) DCS upon the invocation of its next update request. The DCS piggybacks this identifier on the update message that traverses the hierarchy towards the root. The root DCS compares this identifier against the version vector and blocks the message until the update referred to by the identifier is received and applied.

**Read Your Writes:** For implementing Read Your Writes, each DCS maintains a version vector. When a client invokes a query request on a DCS, it passes the local update identifier(s) of the last update(s) it initiated. The DCS synchronizes the query with these identifiers based on the information stored in its version vector.

If Read Your Writes is provided along with FIFO of Writes, one last local update identifier is sufficient to be synchronized with because FIFO is a transitive relation. Otherwise, for each DCS the client sent an update request to, it should remember the last local update identifier received from this DCS. In this case, the query must be synchronized with the entire set of identifiers. However, since we assume in the model that a client only communicates with a small subset of all existing DCSs in the object hierarchy, the set of identifiers is also small and its transfer between a client and a DCS is not an expensive operation.

If Read Your Writes is provided along with FIFO of Reads, the amount of information to transfer and store at a client can be optimized in a different way: The client should only remember the local update identifiers it received since the last query. If the client first issues several updates and then two queries, the first query will be synchronized with the updates and the second query will be synchronized with the first one. Therefore, no explicit synchronization of the second query with the updates is necessary in this case.

Even if Read Your Writes is provided without FIFO of Reads and FIFO of Writes, other optimizations are possible based only on Total Order. Specifically, when the client sends a set $S$ of local update identifiers upon invoking a request on a DCS, this DCS checks which one of identifiers in $S$ corresponds to a later update in the global sequence. Then, it replies with the global identifier of this update along with the return value of the request. If the same client invokes another query at some later point, it will send this global identifier instead of $S$.  

49
In summary, if Total Order is provided, the implementation of the session guarantees introduces an insignificant extra overhead: The amount of consistency information that needs to be stored at clients and transferred between clients and DCSs is small and does not depend on the number of clients and DCSs in the system.

5.3.4 Implementing Session Guarantees Without Total Order

When the Total Order implementation is not employed, an object does not have a single version number. In this case its state can only be characterized by the version vector that has to be maintained by each DCS. While this does not affect the implementation of Read Your Writes and the implementation of FIFO of Writes remains almost as simple as in the case of Total Order, the implementation of FIFO of Reads becomes more complicated and expensive. In addition, an implementation of Session Causality should now be provided. We elaborate on the changes in the implementations below:

**FIFO of Writes:** As with Total Order, a DCS returns a local update identifier to the client that initiates the update, a client remembers only the last local identifier and forgets the previous one, and this local identifier is transferred to a DCS upon the next update request. The only change is that now the DCS blocks the client and does not assign the update request a local identifier until it receives the referred update. If the DCS immediately produced a local update identifier, released the client and left the update request in a pending state, then all later (unrelated) update requests with higher local identifiers would have to wait until this update would be applied. This is a shortcoming of the version vector method which assumes that updates originated at the same DCS are applied in the order of their local identifiers.

An appealing alternative to blocking the client is to use a vector of sliding windows instead of just a vector of update identifiers. Version vectors are so commonly used due to being a compact data structure that allows the service to efficiently determine what updates from a given DCS have been already applied. However, they induce unnecessary FIFO dependencies that are not needed for preserving the required consistency guarantees. On the other hand, a sliding window is a less compact structure than a single version number because it contains a lower window bound and an ordered list of update identifiers inside the window up to the upper bound. The lower window bound plays the same role as a version number. For instance, a value of \( \{ A, 5 \} \) means that all updates of a DCS \( A \) up to \( \{ A, 5 \} \) have been already executed but \( \{ A, 6 \} \) has not. In this example, the list of update identifiers will contain updates with identifiers greater than \( \{ A, 6 \} \) that have already been applied. Note that this list does not have to be continuous. The advantage of using sliding windows is in providing greater flexibility: An update \( U \) can be applied even if some previous updates from the same DCS have not been executed yet, as long as the execution of \( U \) will not violate the required consistency guarantees and will not cause the sliding window to grow beyond the allowed size.

In the context of FIFO of Writes implementation, this enables to assign a local identifier to a client’s update immediately upon its reception and release the client. The downside of using a vector of sliding windows, however, is that a DCS has to remem-
ber the identifiers of the updates applied out of order. Therefore, while eliminating unnecessary delays, this solution requires more space and more complicated version management. Moreover, transferring sliding windows over the network requires more bandwidth compared to version numbers. In particular, this makes the implementation of FIFO of Reads complicated and inefficient. Hence, sliding windows are not currently used in CASCADE.

**FIFO of Reads:** Without Total Order, the simplest implementation of this condition is that a DCS transfers the entire version vector to a client along with the results of a query. The client remembers the version vector it received the last time and forgets the previous vector. This vector is passed to a (possibly different) DCS upon the next client query, and the DCS synchronizes the query with each local identifier in the vector.

This implementation is inefficient because the entire version vector whose length is the number of DCSs in the object hierarchy is sent twice per each query. Below we introduce optimizations that allow us to reduce the average amount of transferred information.

**Session Causality:** Again, the simplest implementation is that a DCS transfers the entire version vector to a client along with the query results. However, unless FIFO of Reads is also provided, it is not sufficient that a client remembers only the version vector it received in the previous interaction with the DCS. Actually, the client must merge all the vectors it received during the execution by computing their maximum. This merged vector is passed to a DCS upon the next client update. Furthermore, since every DCS has to synchronize this update with this vector, the DCS piggybacks the entire vector on the update message sent to other DCSs. A possible optimization here is to use the *causal separators* technique [107] in order to reduce the amount of piggybacked information. This technique appears especially appealing due to the hierarchical architecture employed by CASCADE in which each intermediate node can act as a causal separator.

As we see, when Total Order is not employed, the straightforward implementations of FIFO of Reads and Session Causality are quite expensive in terms of the amount of information to be transferred over the network. Fortunately, the implementation of FIFO of Reads can be significantly improved by using the optimization that is explained below.

**Efficient FIFO of Reads Implementation**

First, rather than sending the entire version vector to a client as part of the response to queries, a DCS can send the difference between its current version vector and the vector received from the client for the purpose of synchronization\(^4\). This difference is usually shorter than the entire version vector. For example, the difference of \(<\langle A, 1 \rangle, \langle B, 3 \rangle\>\) and \(<\langle A, 1 \rangle, \langle B, 1 \rangle\>\) is \(<\langle B, 3 \rangle\>\). Upon receiving such a vector difference the client can add it to the vector it sent and restore the entire version vector of the DCS in its local memory. However, the client should still send its entire version vector for synchronization.

Another optimization is based on the following observation: If a client does not switch DCSs (in other words, it invokes all updates and queries on the same DCS), then FIFO of

\(^4\)This optimization is similar to Singhal-Kshemkalyani technique [112] for implementing vector clocks.
Reads always holds in a trivial way and does not need to be implemented at all. Furthermore, FIFO of Writes and Session Causality also trivially hold due to per-DCS FIFO of Writes and per-DCS Session Causality, respectively. This situation is summarized in Table 5.1 that clearly shows the cost of client mobility.

<table>
<thead>
<tr>
<th>Session Guarantee</th>
<th>with TO</th>
<th>w/o TO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mobile Clients</td>
<td>Not</td>
</tr>
<tr>
<td>FIFO of Reads</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Session Causality</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FIFO of Writes</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Read Your Writes</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

✓ – trivially holds
× – adds extra cost
×× – requires costly communication

Unfortunately, if the consistency implementation is unaware that the client continues to work with the same DCS, it transfers the same high amount of information as if the client switched DCSs. This observation calls for optimizing the implementation for the most usual and frequent case when a client communicates with a single DCS. The client can just verify that it invokes a current request on the same DCS as the previous one. If this is true, the client does not need to send any information for synchronization.

However, a DCS still has to return its version vector along with the query results in order to account for the possibility that a client invokes the next query on another DCS. Furthermore, if a client sends no synchronization information, we can no longer use the differential optimization described above because a DCS has no reference point to compute the difference of vectors.

Thus, there is a need for synchronization information shorter than just an entire version vector. To this end, we introduce a notion of local DCS history which is a numbered sequence of update identifiers of all the updates applied at the DCS during the execution. A local history pointer is just an index to local DCS history. A DCS can return this pointer to a client, and a client can transfer it back to the DCS for synchronization at some later point. As a result, only local history pointers and vector differences are transferred over the network instead of entire version vectors.

As part of this optimization, a DCS should be able to compute the difference between its current version vector and a pointer to some past point of its local history. An important question is how this can be done efficiently without keeping the whole local history. Fortunately, it is sufficient to remember small part of the history, i.e., only the updates in the version vector ordered by their local history pointers. When a query request is invoked and the value of the local history pointer attached to this request is k, the difference sent by the DCS in reply will contain all entries in the version vector whose value of the local history pointer is greater than k. For example, consider a hierarchy with four DCSs A, B, C, and D and the local history \{B,1\}, \{D,1\}, \{D,2\}, \{C,1\}, \{D,3\}, \{C,2\}, \{A,1\}, \{A,2\}, \{A,3\}, \{A,4\}, \{B,2\}, \{B,3\}, \{B,4\}, \{A,5\}, \{B,5\}, \{A,6\}, \{A,7\}, \{A,8\}, \{C,3\}, \{C,4\}. Figure 5.2 shows the structure CASCADE would produce for this history. If a client invokes a query at this moment that is accompanied with a local history pointer 12, the difference between the current version vector and the version vector at the moment when the update number 12 was executed would be \{B,5\}, \{A,8\}, \{C,4\}. Note that this difference does not
contain the last local update identifier of $D$ because that update has a local history pointer smaller than 12.

![Figure 5.2: A structure required for efficient computation of version vector differences](image)

This optimization is generalized for the case when a client communicates with several DCSs in the following way: For each DCS $D$ a client invoked a query on, it remembers the last local history pointer $LHP_D$ it received from this DCS and the version vector $VV_D$ corresponding to this pointer. (As we have already seen, this vector can be restored in the client memory without being transferred over the network). In other words, a client remembers how advanced each DCS is. Note that since FIFO of Reads is a transitive relation, the vector corresponding to the last query is always more advanced than all other vectors. Therefore, this vector represents the knowledge of the client itself about the last object version. Let us denote it $VV_{client}$. When a client invokes a query on a DCS $D$, it transfers $LHP_D$ along with the difference between $VV_{client}$ and $VV_D$. When the client receives a new local history pointer and a vector difference piggybacked on query results, it updates both $VV_D$ and $VV_{client}$ to the same value.

While this optimization requires that a client keeps a version vector for each DCS it communicates with, it is still very efficient because even a client that switches between different servers usually works with a small number of DCSs as noted in Section 5.3.1. Also, recall that all these complex algorithms of version vectors management are implemented in the library; they are transparent for the application.

### 5.3.5 Object Group-Based Consistency

The above mentioned consistency policies apply to an individual object. However, some applications might wish to impose sequential consistency across several objects. To address this, we introduce object group-based consistency policy. With this policy, a group of objects, each one having its own hierarchy, are to be maintained in a strongly consistent manner. We impose a restriction for this policy that these hierarchies must have a common root. Without such a limitation the algorithms for achieving group consistency become prohibitively expensive.

This policy is implemented in the following way: The common root introduces a total order on the updates of all the objects in the group. Then, updates of each object are propagated through its own hierarchy. However, if some DCS has cached copies of more than one object in the group, this DCS does not apply updates to any object before it applies all other preceding updates, including updates of other objects.
5.4 Support for Atomic Operations and Locking

Sometimes it is required to execute several operations \textit{atomically}, without being interrupted by the execution of other requests. CASCADE provides the client a possibility to specify several update operations in one request. To achieve this, the client should invoke an \textit{update\_object} request that is part of the CASCADE API (see Section A). Each DCS applying this request processes the specified sequence of update operations atomically. Furthermore, when an object group-based consistency is used and the client’s local DCS has cached copies of two objects $X$ and $Y$ belonging to the same group, it is possible to issue a request that comprise both operations on $X$ and operations on $Y$.

Locking can be very useful for applications that occasionally need to access the most updated object copy. CASCADE provides two-phase locking for cached objects that have the Total Order consistency condition as part of the object policy. When a client needs a lock, it issues a \textit{lock\_object} request on its local DCS. This request is treated as an update in the sense that it is propagated to the root of the hierarchy and inserted into the global sequence of updates. When the turn of a locking request for execution arrives, the root transfers its object copy to the local DCS of the client that requested the lock and blocks all further updates. When the client releases the lock by issuing a \textit{unlock\_object} request, the local DCS sends an updated object copy back to the root server that resumes processing of updates.

However, it may occur that the client which possesses the lock crashes (see Section 5.2), or the application programmer forgets to add an unlock request. Since CASCADE cannot depend on the reliability of clients, special care should be taken so that the object would not remain locked forever. To prevent this, CASCADE employs the technique of lease based locking [55]. Specifically, the DCS that obtained the lock releases it automatically after a pre-defined timeout unless it receives a request from the same client for another lease period. Thus, if the client intends to continue updating the object, it should issue another lock request before the timeout elapses.

CASCADE supports three types of object locks defined in the specifications of CORBA Concurrency Control Service [94]: \textit{Write} lock conflicts with any other lock, \textit{read} lock conflicts only with a write lock, and \textit{upgrade} lock conflicts with a write lock and with other upgrade locks.

As part of fault-tolerance support, it is important to provide transactional semantics. In CASCADE, the root of the hierarchy can be a natural coordinator of commitment protocols such as two-phase commit. Note that transactions can span multiple objects in the case of an object group-based consistency when the root is shared between several hierarchies.

5.5 Overview of Cache Management in CASCADE

Since the size of the cache is limited, a DCS can hold only a limited number of cached copies. When a cached copy is to be brought to a DCS but there is no more space in the cache, some other object must be evacuated from the cache first. Here we only outline the cache management basics in CASCADE while a detailed description can be found in [18] and [19].

First of all, not all cached objects are suitable for evacuation. In particular, objects locked by this DCS and/or objects whose methods are currently being executed are never
replaced. In addition, since each object registered with CASCADE should remain available until it is explicitly unregistered, objects for which this DCS is the root of the hierarchy are also never evacuated from the cache.

In order to prevent uncontrolled growth of the total size of registered objects, CASCADE imposes two limits: (1) on the maximal number of objects registered at each DCS, and (2) on the maximal size of each registered object. If the total number of registered objects reaches the limit, new register requests are rejected. In addition, whenever the size of some registered object grows beyond the allowed limit, it is unregistered and its hierarchy is destroyed.

When an object copy is evacuated from the cache, the DCS keeps a small record needed for information dissemination along the hierarchy. This record is finally removed only when (1) no client has issued a request for the object during a pre-defined timeout, and (2) this DCS becomes a leaf of the object hierarchy. When this occurs, the DCS undertakes the steps detailed in Section 5.1 in order to disconnect from the hierarchy.

If some previously evacuated object is required later by a client, it will be acquired again from the father node transparently for this client. In turn, if the father node does not have a copy of the requested object, it will try to acquire it from its father node. This way the request ascends all the way up along the hierarchy until the DCS that has a copy of the object is reached. Note that the request chain always terminates because the object is never evacuated from the root of the hierarchy. The object copy then descends along the hierarchy back to the request originator.

A general cached object typically consumes much more resources than a simple web document. While a web document is usually kept on a disk, objects have to be in the main memory in order to execute operations on them. Furthermore, since objects are running, they consume CPU and possibly other resources. Therefore, the choice of replacement policy can have significant influence on performance.

CASCADE supports a wide range of cache replacement policies, including LRU, LFU, LFU with aging, Size, Inverse Size, Greedy Dual Size Frequency and others. In addition CASCADE introduces policies that take into account parameters related to the knowledge of the hierarchy such as the number of descendant nodes this DCS has in the examined object’s hierarchy. Implementation of these policies as well as their analysis are presented in [18] and [19].

5.5.1 Calculation of Object’s Size

A basic accounting operation of any cache management is calculating an object size. Unfortunately, there is no direct way in Java to obtain an object’s size at runtime. We used reflection in order to calculate the size. Reflection is a feature of Java that allows an executing Java program to examine, or “introspect”, itself and manipulate internal properties of the program. For example, it is possible for a Java class to obtain the names of all its members and display them. We used reflection to get all members of the class representing the object. For each member, we checked its type: If it is a primitive type, then its size is known a priori and is added to the total size; if it is a reference to another class, then reflection is performed recursively on that class. During this process, we must avoid entering loops, i.e., referencing an object that was already referenced before.

55
5.6 Mobile Code

For many objects, code is longer than data. Therefore, proper code caching is essential for efficient service operation. In particular, it is important to take into account that not all pieces of an object code are necessary for a successful object deployment: while some parts of the code can be executed quite often, others can be called infrequently or not used at all. Such unused parts of the code do not need to be cached and transferred over the network. It should be noted, however, that developing optimizations for code management requires in-depth understanding of the programming language features as well as of the compilation and loading process.

Typically, a CORBA object is coarse grain: it incorporates complex application logic and it is composed of many programming language objects. In particular, CORBA object code comprises multiple programming language code classes. This partition determines the granularity of code caching in CASCADE: each individual class can be cached as a unit. In principle, more effective optimizations can be obtained when dividing the classes into smaller units, e.g., methods and caching the code at a smaller granularity. However, this would require instrumenting the object code (as done, e.g., in [69]) which we are trying to avoid.

The two main operations in dealing with mobile code are a) code fetching and loading, and b) code unloading and evacuation. Below we consider these operations in greater detail.

5.6.1 Code Fetching and Loading

In Java, there are few ways in which code classes can be related to each other:

**Inheritance:** Every Java class except for `java.lang.Object` has exactly one superclass. However, an interface may inherit from multiple interfaces. Furthermore, when a class implements an interface, we also consider this to be an instance of the inheritance relation. Obviously, a transitive closure of the inheritance relation cannot contain cycles, i.e., a class cannot be an ancestor of itself.

**Containment:** An object class can contain a non-primitive field whose type is defined by another class. Such a field can be either static (i.e., a class field) or non-static (i.e., an instance field). Note that in the implementation, an instance of A will contain a reference to an instance of B rather than the instance itself. Thus, cyclic containment relations are possible.

**Reference:** A class may refer to another class from its code, either from a method or from a field initializer. Note that every reference to a non-static field or method of an object instance is always preceded by the instance creation in the context of a virtual machine execution.

Let us consider a linking process in Java [73]. Before a class can be deployed, it has to be loaded, linked and initialized. Linking involves three different activities: verification, preparation (i.e., allocation of tables and other resources) and resolution of references. Of those activities only resolution requires traversing references to other classes and might provoke recursive loading of other classes (deep resolution). However, the specifications in [73]
only stipulate resolving the reference to the superclass as mandatory. Implementation has
the freedom to choose between early and lazy resolution with regard to all other references.
In practice, we are aware of no JVM implementation that resolves all references recursively;
most existing JVMs use lazy resolution. This implies that in order to deploy a class, only
the class itself and its ancestor classes need to be loaded and linked. As the object code
executes, further references to other classes due to containment and reference relations may
need to be resolved. Specifically, a class is loaded when either its instance is created, or a
static field or method of the class is accessed.

This delayed code loading allows for various optimizations when fetching the code from
a remote location. For each class C, we define C's code bundle to be the code of C itself
plus the code of all C's ancestor classes and interfaces that C implements. It is clear from
the discussion above that the C's code bundle is the minimal and necessary amount of
code required to deploy C. Thus, the simplest way of transferring an object code is by code
bundles: In order to deploy a CORBA object at a new location (in the context of CASCADE,
this would be a new DCS joining the hierarchy), the main code bundle, i.e., the code bundle
of the main servant class for this object, has to be transferred and loaded. Once the object
is deployed, its methods can be invoked. At some future point, executing an object code can
result in further reference resolution and demand for another class D. When this happens,
the DCS requests D from its father node in the object hierarchy and the father DCS ships
the whole D's code bundle. Of course, the father DCS should keep track of the classes that
it sent in the past so that the same class will not be sent twice.

We can make this code transfer method slightly more efficient by refining the definition of
a code bundle: there is no need to include standard Java and CORBA classes because those
classes exist initially at each location and they do not need to be distributed. Furthermore,
standard Java and CORBA classes can reference only other Java and CORBA classes so
using such classes cannot result in a request for a class which does not exist locally. We
ignore references to such classes in the rest of the description.

Aside from simplicity of shipping the code by bundles, the main advantage of this transfer
method is maximal economy of both bandwidth and computer memory resources: a class is
transferred and loaded only if it is actually used. The downside, however, is long latencies
required to download an object code: When an unresolved class is encountered, the linking
process is blocked until a code bundle is received. Immediately after linking is resumed,
another class can be requested causing the process to block again. This might be repeated
many times before a single object method can be executed. Furthermore, classes may need to
be resolved as soon as they reach the class initialization phase because static field initializers
may refer to other classes.

Code transfer time can be reduced by using pipelining when sending code bundles. However,
the sending DCS has to know what bundles to send and in what order. Shipping all
of the object code in an arbitrary order wastes both network bandwidth and computer re-
sources of the receiving DCS because some code classes may only be needed in the distant
future, or not at all. Moreover, if classes required for object initialization are transferred
after those that are never used, there would hardly be any gain in object deployment time.
Usage of Object Class Graph

In order to determine what classes to send, CASCADE introduces a notion of directed object class graph. The nodes in this graph are classes and the edges are relations between the classes. There are three types of edges according to the three kinds of relations specified above. Figure 5.3 provides an illustration of an object class graph. Inheritance, containment and reference relations are represented by solid, dashed and dotted arrows, respectively. One particular application of a class graph is for computing code bundles efficiently. Note that the bundles do not necessarily share a root because java.lang.Object and other standard Java and CORBA classes are not included in the graph.

Next, we introduce a metric on a class graph. That is, we associate non-negative distances with every edge in this graph. A zero distance on the edge from class $C$ to class $D$ means that $C$ cannot be loaded without $D$. The greater is the distance, the less important for class deployment this edge is. As explained below, distances are used for computing the set of classes to be sent over the network.

It might be possible to achieve a more precise heuristic by associating with an edge the probability of traversing this edge during the object execution. When a request for class $C$ arrives, we could compute the probability for each node to be reached from $C$ and ship the nodes whose probability exceeds some predefined threshold. However, this would be computationally expensive. Furthermore, it is only possible to approximate the probability values with some precision. It should be noted that slight changes in the input probabilities can have a strong impact on the output probabilities. Therefore, small estimation errors would diminish the effectiveness of this “theoretically perfect” heuristic compared with other computationally cheaper methods.

The algorithm in CASCADE simply considers all classes in the class graph whose distance from $C$ does not exceed some predefined threshold, i.e., a circle with a certain radius around $C$. More precisely, it considers all classes in a small radius $\alpha_1$ and some of the classes
in a bigger radius \( \alpha_2 \). In addition to the significance of relations between the classes, the algorithm takes into account the sizes of those classes. Intuitively, if a short class and a long class have the same probability of being used, it is cheaper to transfer and load the short class. The algorithm uses two size-related parameters: \( \text{MaxClassSize} \) and \( \text{SizeThreshold} \). Classes bigger than \( \text{MaxClassSize} \), which are at a distance between \( \alpha_1 \) and \( \alpha_2 \) from \( C \), have their distance multiplied by the ratio of their size to \( \text{MaxClassSize} \). These classes can be transferred only if the resulting distance is still smaller than \( \alpha_2 \). Furthermore, classes at a distance between \( \alpha_1 \) and \( \alpha_2 \) from \( C \) will be transferred only as long as the cumulative size of the classes to be sent does not exceed \( \text{SizeThreshold} \).

The pseudocode of the algorithm is presented in Figure 5.4. The COMPUTE-SEND-LIST function takes the requested class \( s \) and the son DCS, called \( \text{sonDCS} \), which sent the request, and returns an ordered list of classes to send. The distance on edge \((u,v)\) is denoted \( d(u,v) \) and the size of node \( v \) is denoted \( v.size \). The algorithm is based on Dijkstra’s algorithm for finding distances to each graph vertex from a given source node [37]. Like Dijkstra’s algorithm, our implementation is greedy: it proceeds by iterations, each time taking the next graph node closest to \( v \) that has not been taken yet. Similarly, the algorithm maintains an array \( \text{dist} \), which contains the shortest currently known distance from the given source node (i.e., the requested class) to each node in the graph. The main differences are that our algorithm takes sizes of classes into account, it produces a different output, and it stops computing distances and terminates when encountering the \( \alpha_2 \) limit. Furthermore, each DCS holds a variable \( \text{SentClasses}_{\text{sonDCS}} \) for each \( \text{sonDCS} \) to remember the set of classes sent to \( \text{sonDCS} \) in the past. Previously transferred classes are not sent again but their nodes need to be traversed because edges from such classes may lead to classes that have not been previously sent but should be shipped this time.

Note that the order of classes in the returned list is important: this is the order in which the classes should be sent. The classes in the beginning of the list are required immediately for object deployment while the classes at the end of the list might be needed only at some later point during the object execution. Of course, pipelining is used when transferring the classes.

The operation of the algorithm is illustrated in Figure 5.3. The numbers on the edges denote distances. In this example, the class \( C \) was requested by a son DCS. The values of \( \alpha_1 \) and \( \alpha_2 \) were set to 1 and 3. The bold solid line surrounds the area of radius \( \alpha_1 \) around \( C \) and the bold dashed line encloses the area of radius \( \alpha_2 \) around \( C \). The size of class \( F \) is three times \( \text{MaxClassSize} \) while all other classes are shorter than this limit. Assuming that no class has been transferred to the requesting son DCS prior to this call, the algorithm will produce the following list: \( C, B, A, D, E, G, H \). \( F \) and \( I \) will not be transferred because the weighted distance from \( C \) to \( F \) will be \( 1.5 \times 3 = 4.5 > \alpha_2 \).

**Construction of Object Class Graph**

Frequently, an application puts into a cache several objects that comprise the same code but different data. In such cases, the same class graph can be used for multiple objects. When a CORBA object is registered at some DCS, this DCS looks up if it already has information about the main object class. If there is no information about the code of the object that is being registered, a new class graph is created for this object.
RELAX \( (u, v) \)
\begin{verbatim}
begin
  newdist := dist[u] + d(u, v)
  if newdist ≤ α₁ then
    UPDATE-DIST(v, newdist)
  else if newdist ≤ α₂ then
    if v.size > MaxClassSize then
      newdist := newdist * (v.size / MaxClassSize)
    UPDATE-DIST(v, newdist)
end
\end{verbatim}

UPDATE-DIST \( (v, newdist) \)
\begin{verbatim}
begin
  if newdist < dist[v] then
    dist[v] := newdist
end
\end{verbatim}

COMPUTE-SEND-LIST \( (s, sonDCS) \)
\begin{verbatim}
begin
  initialize all entries in dist to ∞
  dist[s] := 0

  SendList := ∅
  SendSize := 0

  while there exists an unmarked vertex do
    let u be an unmarked vertex such that dist[u] is minimal
    if dist[u] > α₁ then
      break from the loop
    if u ∉ SentClasses_{sonDCS} then
      if dist[u] > α₁ and SendSize + u.size > SizeThreshold then
        break from the loop
      append u to the end of SendList
      SendSize := SendSize + u.size
    mark u
    for each edge \( (u, v) \) such that v is unmarked do
      RELAX \( (u, v) \)
  return SendList
end
\end{verbatim}

Figure 5.4: The algorithm computing a list of classes to transfer

We assume that unlike CASCADE, a CASCADE application itself does not generate or obtain Java bytecode on the fly. Another assumption is that the application does not use Java objects collocated with it but created externally. Furthermore, CASCADE does not
currently support dynamic class upgrades after the object has been already deployed. In these settings, the entire object class graph can be constructed upon an object registration by recursively parsing the bytecode of the main CORBA object class. Moreover, an object graph does not change throughout an object execution. This graph is considered part of the object meta-information and it is transferred along with the object data and some code classes when a new DCS joins the object hierarchy. Note that the amount of data needed to encode a class graph is significantly smaller than the size of classes so it makes sense to send the whole graph in order to save in class transfer.

The only class graph data that changes over time is distances on the edges. All inheritance edge distances are permanently set to zero because a class cannot be deployed without its ancestor classes. Therefore, if a certain class is within some radius from the requested class, all its code bundle is within this radius. At the same time, containment and reference edges may have varying distances. Some initial values are assigned to these edges when the graph is created by parsing the class structure. In particular, if a class field or a reference to another class is used in a constructor or in a field initializer, a distance to this class will be smaller than a distance to a class used in one of the non-constructor methods. Furthermore, the number of times the class is referenced and the number of methods from which the class is referenced also affect initial distance values.\(^5\)

Non-zero distance values are updated dynamically during an object execution. When an attempt to resolve a reference from class \(C\) to class \(D\) results in a request for \(D\), then both the requesting DCS and its father DCS supplying the requested class reduce the distance on the edge from \(C\) to \(D\). However, it is not trivial in Java to determine the source of the class graph edge, i.e., the class from which a reference is being resolved because the Java runtime system does not expose this information directly to the application. The process of class loading and resolution in Java is done through user defined class loaders. Class loaders do not participate in the resolution process which is done solely by the JVM itself. They are rather responsible for supplying a class bytecode when a JVM encounters a reference to a class it cannot find.

The idea used in CASCADE is that the application can create multiple class loaders. When a JVM needs to resolve a reference from class \(C\) to a missing class \(D\), the request for \(D\) is issued on the class loader that supplied \(C\) as specified in [73]. At this point, the class loader may create another class loader and forward the request to it. This way, each class will be defined by its own class loader and it will be straightforward to derive the source of the class graph edge from the class loader on which the request for a missing class is issued. However, creating a different class loader for each individual class is wasteful. To explain how this problem is handled efficiently in CASCADE, let us observe that loading two classes with the same loader may cause later uncertainty (dubbed a “conflict” in this context) only if these two classes have a reference to a third class that has not been loaded yet. For example, in Figure 5.3 only classes \(C\) and \(B\) have to be loaded with different class loaders because they both point to \(J\). Furthermore, “conflicts” are pairwise and non-transitive, i.e., if a class conflicts with two other classes but those classes do not conflict with each other, then these classes can be loaded by the same class loader. Thus, when we bring a class in, we need

---

\(^5\)In principle, finer flow analysis techniques can be used to obtain more precise distance value. However, such techniques can be quite complex and computationally intensive, which is why they are not used in the current version of CASCADE.
to create a new class loader for it only if some reference from this class points to a class \( D \) which has not been loaded yet and each existing class loader has already defined a class that has a reference to \( D \). In practice, this means that very few class loaders suffice for loading all classes in a class graph.

Obviously, different methods can be invoked on various cached object copies and different pieces of an object code can be used at various locations. As a result, distances associated with the same object class graph at various DCSs may diverge. Whether distances should be consistent over all of the object hierarchy depends on the behavior pattern of the application: if the same methods are called and the same parts of code are executed at all locations, it would make sense to propagate updates of distances among all DCSs. On the other hand, if the application behavior differs from one location to another, each DCS should only take into account invocation requests initiated by its local clients.

Currently, the choice in CASCADE is determined by a policy parameter among the following options:

- The default algorithm is that each DCS periodically sends the distance updates it collected to its parent node, until all of the updates reach the root of the hierarchy. Then, the root distributes the collected information down the tree to all DCSs.

- No information exchange at all.

- A regional based behavior. In this policy, all DCSs are divided into groups and only DCSs that belong to the same group (e.g., DCSs in the same country) will exchange distance updates. This policy assumes that there is a direct relation between users’ geographical location and the way they use an object.

Other approaches can be employed as well. For example, it is possible to build a global statistics data storage that is shared by all DCSs. All DCSs will periodically send statistical data about an object execution to this storage. Off-line data mining and flow analysis tools can be used to extract interesting information from the statistics data and predict future object usage and methods invocations. Alternatively, it is possible to use a statistics collecting agent that visits all hierarchy nodes periodically and aggregates the collected statistics.

**Summary of Code Fetching Approaches**

To summarize, the following code transfer methods are supported by CASCADE: (again, the choice is governed by the policy)

**Minimal transfer:** When a class is requested, its code bundle is transferred. This method saves bandwidth and computer memory resources but it may incur long latencies in object deployment.

**Transfer by object class graph:** This method is based on the algorithm 5.4 described above. It attains a good compromise between consumed resources and object deployment latency. However, the method itself is quite complex and it requires maintaining the class graph and sending the graph and graph updates over the network.
**Complete transfer:** In this method, all classes in the class graph are transferred. The graph is created only once at the object registration stage to obtain a list of classes that are required to deploy the object. Initial distances are assigned and the COMPUTE-SEND-LIST function is invoked in order to set the order in which classes should be sent. However, the class graph is not transferred over the network, it does not need to be maintained, and the distances are not updated. This method achieves the best latency but may require a lot of resources.

**Transfer by simple statistics:** Techniques based on an object class graph are complex and they require additional communication. This method employs much simpler statistics: for each class, the DCS remembers the number of son DCSs that requested this class. When a class is requested by a DCS, its code bundle is sent along with all other classes that were sent to at least $\beta$ percent of all son DCSs but not to this DCS. Additional experiments need to be conducted to establish the optimal value of $\beta$ and to compare this approach with those listed above.

### 5.6.2 Code Evacuation

When an object hierarchy is destroyed, or an object copy is temporarily evacuated, its code should be unloaded and removed as well, unless it is used by another object. According to the last version of the JVM specifications [73], a class is unloaded when there are no instances of this class and the class loader that defined the class is no longer reachable so it will be collected by a garbage collector. Thus, when an object is removed from the cache in CASCADE and no part of its class graph is shared with other objects in cache, all references to the class loaders that defined the classes of this class graph are nullified, effectively removing the object code.

In principle, it would be useful to implement a partial code evacuation, that is, a removal of some classes in the object class graph. This would make sense when some classes of an object graph have not been used for a long time. However, there is no efficient way to trace method invocations of a Java object so it is difficult to figure out when a class was used for the last time without employing some kind of profiling, which can severely slow down an object execution. There is even no simple way to verify if a given class is being used, e.g., if its instance currently exists. This is because the Java standard provides no efficient way to intercept creation of new class instances. While it is possible to traverse all currently existing objects in the same manner it is done by a garbage collector, doing this at the application level can hardly be considered a practical approach.

A partial code evacuation would also be useful when a part of a class graph is shared between two cached CORBA objects, and one of those CORBA objects is being evacuated. In this case, it would make sense to remove the classes in the non-shared portion of the graph. Unfortunately, the Java language does not provide as much control over the unloading process as it does with loading the code. In order to evacuate an individual class, we need to nullify the reference to the class loader that loaded this class. However, this class loader may have loaded other classes as well. The idea of defining classes by different class loaders that we suggested in Section 5.6.1 can solve this problem as well. Unfortunately, unlike in the solution discussed in Section 5.6.1, here we can do only marginally better than defining each class by its own class loader. The only possible optimization here is to define classes that
belong to the same strongly connected component of a class graph by the same loader. This will not create a problem because if a node \( v \) is shared between multiple class graphs, then all nodes that belong to the same strongly connected component as \( v \) are shared between those graphs. However, the number of strongly connected components in a class graph can still be prohibitively large. For example, in Figure 5.3 there is only one non-trivial strongly connected component: \( \{ B, C, D \} \). Thus, each class of \( A, E, F, G, H, I, \) and \( J \) has to be defined by a class loader of its own.

Therefore, CASCADE supports only removal of all classes in a class graph. When a class is removed, its bytecode is saved on a disk. If the class is required later for the same or another object, it will be loaded from the disk instead of being transferred over the network. Moreover, the object class graph including distance values is stored on a disk as well. As we outlined in Section 5.6.1, multiple application objects placed into a cache are frequently based on partially or completely the same code. Therefore, statistics collected about a code execution can turn out to be useful for the objects that would be placed into a cache in the future.

### 5.7 Flow Control

A substantial amount of data can be exchanged between DCSs, mainly due to propagation of updates and update results. The internal communication between DCSs is also done through CORBA, which uses IIOP/GIOP as the main communication protocol. IIOP is built atop TCP, which has an elaborate flow control mechanism that resulted from many years of research. However, this mechanism is tailored mainly for “classical” (Ethernet, token ring, and similar) LANs and for interactive applications like Telnet. It has serious drawbacks for most other applications as suggested by studies of persistent HTTP [57, 89], Sun RPC [38], ATM networks [85] and X Windows System [110]. The results of this dissertation in Section 7 confirm some of these problems. Furthermore, most currently existing ORBs do not take into account these TCP performance issues when running over WANs and do not attempt to cope with them. We start this section with a discussion about TCP and its interaction with ORBs. Then, we describe the flow control mechanisms used in CASCADE to address these issues.

#### 5.7.1 Discussion of TCP Performance

**Nagle Related Issues**

According to the Nagle buffering algorithm in BSD style implementations [113], a packet is not sent until one of the following occurs: a) a full-size segment can be sent, b) half of the client’s advertised window can be sent, or c) the sender does not expect an ACK on a previous packet. Thus, if the application message is short (few hundreds of bytes), it might be delayed until the ACK on the previous packet arrives. In wide area networks with long delays, this means that sending the message might be delayed by hundreds of milliseconds. The situation is even worse when the ACK on the previous packet is lost, or when the ACK is delayed due to the TCP delayed acknowledgement algorithm.

While the Nagle algorithm is enabled by default, most implementations allow for disabling
it through a corresponding operating system call, and this option is included in the Java TCP socket API. However, if there is no buffering mechanism and the application sends frequent short requests, each request will result in a packet at the network level. This is wasteful because of the packet header overhead and excessive accesses to the network. Furthermore, it may lead to an excessive load on routers. Thus, the application itself needs to provide a buffering mechanism. As the performance study in the above mentioned papers and Section 7 shows, a properly implemented buffering mechanism at the application level drastically reduces the latencies while sending only marginally more packets compared with the Nagle algorithm.

**TCP Retransmission Policy**

Standard TCP implementations have several performance problems when used over high latency or high bandwidth connections. Recently, several performance extensions like TCP-Vegas [27], selective acknowledgements [46] and NewReno [44] have been introduced. Out of those, only NewReno became a de-facto standard so it is supported and enabled by default in most implementations. Unfortunately, even with NewReno, TCP tends to be very conservative when dealing with retransmission timeouts. For example, we observed that a very short request that normally took about 250ms to propagate in both directions and get executed could take as long as 1200ms in the case of a single packet loss. Since most Internet links experience frequent periods of high lossiness (5 to 20%), this has a dramatic impact on latencies. When packets are lost during the slow start phase and the congestion window is still small, this may lead to remarkably long latencies, up to dozens of seconds to transfer a 4Kb request.

While TCP-Vegas [27] alleviates much of the problem according to several studies, it is still unsupported by many implementations. Furthermore, the Java socket API does not provide the application with any control over performance extensions. Thus, very little can be done at the application level to cope with long latencies due to inefficient loss discovery mechanism. One idea is to open an auxiliary TCP connection between a pair of DCSs in addition to the main one and detect losses at the application level: if a request sent over the main connection does not return within a timeout based on the RTT value, it is assumed to be lost so the application clones the request and reinvokes the copy over the auxiliary connection. Of course, the application must maintain RTT in order to employ this method and it must be able to deal with duplicate request invocations. While not being a real solution, this technique appears to smooth the peak latencies in the extreme cases.

**Socket Buffer Size**

The TCP socket buffer size is known to affect the performance: a larger buffer allows for higher concurrency of copying operations but requires more memory and may lead to long backlogs. Since the optimal buffer size depends on the communication induced by the application, the optimal decision w.r.t. the TCP buffer size should be done at the application level rather than the network one. The standard Java TCP socket API allows for changing the buffer size. However, the default length of TCP socket buffers is operating system dependent so the degree of inefficiency highly depends on a particular TCP implementation. In most platforms where we ran CASCADE the size of the buffer was large enough to accommodate
all of the traffic being received or sent so changing this parameter led only to a marginal improvement. In contrast, old BSD-derived systems have this parameter set to 4096 bytes which is insufficient for most kinds of communication.

5.7.2 Flow Control Implementation in CASCADE

All internal communication between DCSs in CASCADE is passed through the layer of connectors. Connector is an endpoint of DCS-to-DCS connection at the level of CASCADE. Consider Figure 3.2 again. In this example, DCS A has 4 connectors to communicate with DCSs A.B, A.C, A.D and A.E. Note that connectors are not per-object: if two DCSs are neighbor nodes in multiple object hierarchies, each DCS has only a single connector to the other. Thus, connectors multiplex the communication intended for different hierarchies into a single stream.

Essentially, the connector functionality can be summarized as a) buffering and rate control, b) data merge and replacement, c) TCP connection management, and d) thread management. We describe each of these mechanisms in a separate section below.

Buffering and rate control

As we pointed out above, CASCADE disables the Nagle algorithm and implements its own buffering mechanism. Data are buffered by connectors due to several factors:

- It is well known that bursty communication increases loss percentage. Therefore, connectors employ a leaky bucket style [116] scheme to limit the rate of sending packages. In addition, connectors attempt to avoid long bursts by introducing a longer delay after sending a sequence of messages in a sequel.

- If the receiving DCS is unable to process incoming requests at the speed of their arrivals and its backlog grows, it asks the sender to slow down. This causes the sender to reduce the sending rate. In such a case, the receiving DCS is responsible for notifying the sender when the queue of incoming requests shrinks again so that the sender can send messages at a higher rate again.

- When a new request needs to be sent, it might happen that all senders’ threads are busy and there would be no available thread to send the request. In this case, the request is buffered until some senders’ thread becomes available. Availability of threads depends on the threading scheme, which is described in Section 5.7.2.

It should be emphasized that buffered data should be sent in chunks that do not exceed the maximal segment size. Furthermore, if a single long request can be sent immediately without any buffering, connectors will still break it into fragments, send a few initial fragments, but defer sending the rest. We found sending long messages extremely harmful based on our experiments:

- Sending a long, multiple segment message as a single method invocation produces bursts in the communication pattern. On the other hand, breaking the message into single segment fragments gives more flexibility in the sending rate control. Furthermore, it is possible to send the fragments through different connections as described below.
When using the POSIX TCP socket API, bytes are available to the application immediately after they arrive. However, with the CORBA RPC semantics, if a long request/reply consists of multiple segments, it will only be available to the application once all segments have arrived. Thus, if few update requests are buffered and sent as a single CORBA request, the receiver would not be able to deliver the first update request as soon as it would be available. It would rather have to wait for all of them to arrive before delivering them. Thus, if the segment containing the second request is lost, this will substantially affect the latency of the first request as well.

To see how drastic this difference can be, let us consider some realistic scenario: Assume that a packet is lost on some link with probability \((1 - \alpha)\), the average latency of delivering a packet without losses is \(c_1\), and the average latency of delivering a packet that requires at least one retransmission is \(c_2\) (both latencies are under the assumption that there are no other packets in transit). Suppose the application has two requests in a send buffer, each request taking one segment. Moreover, the TCP congestion window is large enough and the segments are sent back to back. Since we are interested in a qualitative assessment rather than an exact quantitative analysis, we will also make a very rough assumption that the loss detection and retransmission mechanisms work independently for both segments. If both requests are sent as separate invocations, the average latency for the first one will be \(\alpha \cdot c_1 + (1 - \alpha) \cdot c_2\), while the average latency for the second one will be \(\alpha^2 \cdot c_1 + (1 - \alpha^2) \cdot c_2\). On the other hand, if the requests are sent in a single invocation, the average latency for the second request will be the same but the latency for the first request will increase to \(\alpha^2 \cdot c_1 + (1 - \alpha^2) \cdot c_2\). The difference between the latencies in the two cases is \((\alpha - \alpha^2)(c_2 - c_1)\). By substituting \(\alpha = 90\%\), \(c_1 = 200\) ms and \(c_2 = 1100\) ms (these are the real values from one of the links we used in our experiments), we get a difference of 81 ms.

In most WANs, the maximal segment size is set to 512 bytes. In CASCADE, connectors never send messages longer than 450 bytes as a single invocation in order to provide space for CORBA and TCP/IP headers.

**Data merge and replacement**

Due to the buffering mechanism, the data may remain in buffers for a significant time before being sent. Meanwhile, further update requests for the same object may be produced. These later requests may be merged with the previous ones, thus saving the space in request headers. Even better, in the case of propagating objects as opposed to propagating updates, each update invocation results in a message that contains a serialized object version so a later invocation result can in fact replace the previous one. Under heavy update loads, this optimization was observed to reduce both the traffic volume and latencies by as much as 50 percent.

**TCP connection management**

In addition to the performance issues related to TCP itself, an application like CASCADE has to deal with TCP connection management in CORBA and ORB implementations: Let us assume that a couple of DCSs hold cached copies of the same few objects and extensively
exchange update requests on these objects. An important question is how many TCP connections should be open between the DCSs for this communication. This issue arises due to the TCP FIFO semantics: If all messages are sent through the same connection, then unnecessary communication induced FIFO dependencies appear. Thus, an update request may not be delivered to the application until the previously sent requests on other objects arrive. In particular, if a previously sent request is lost, it will lead to unnecessary long delays. On the other hand, it may be too costly to open a new TCP connection for each of the objects since a DCS can hold many objects in its cache.

Probably the smartest way to manage connections for an ORB would be to maintain a small connection pool for a group of objects residing on the same host and accepting requests in such a way that each of the objects can be reached through any of the connections in the pool. In this scheme, when an ORB needs to send a request, it attempts to pick a connection

---

Figure 5.5: Connection management in CASCADE
on which all sent requests have been already delivered. However, this is not implemented in
any ORB we are aware of. Furthermore, this scheme does not provide a full solution to the
problem: if an object (like the main DCS object) accepts requests of several unrelated types
(e.g., different methods), the FIFO semantics might need to be preserved between requests
of the same type but not between requests of different types. In this case, the application
might desire to send requests of each type on a different connection. At the same time, there
is no way for the ORB to understand the request semantics and choose the best connection
management scheme. Thus, the best an ORB can do in the most general case is to expose
the control over the connection management to the application.

Nevertheless, before the introduction of Portable Object Adaptor (POA), ORBs provided
very little means for the application to control connection management, and the way it could
be done was highly non-portable. The POA standard allows for creating POA managers,
each of which listens on a different port. This way an object can be attached to multiple
POA managers (through several POAs) and have multiple IORs so it can be reached via
more than a single connection, as depicted in Figure 5.5.

This was used in CASCADE in the following way: All internal DCS-to-DCS communication
is done by means of method invocations on the main DCS object. This DCS object can
be attached to multiple POAs as described above. A connector for a neighbor DCS main-
tains a pool of IORs for the neighbor DCS object. While all update requests for the same
object are usually sent through the same connection (i.e., as invocations on the same IOR),
the first update request on a newly cached object is sent via the least loaded connection.
If the neighbor DCS senses that the connections become overloaded (i.e., requests for too
many objects are multiplexed through the same connection), it may create another POA
manager, attach the DCS object to it, create a new IOR, and notify the neighbor nodes.
This effectively enlarges the connection pool for this DCS.

Thread management

The most common CORBA invocation semantics are synchronous RPC. In other words,
an invoking thread is blocked until the answer (which may be a single reply status value)
arrives. Thus, \( n \) threads are required to send \( n \) requests in parallel. We explained in
the description of the buffering mechanism why the length of a single request should not exceed
the maximum segment size. Therefore, if the congestion window is large enough, then the
number of threads required for request invocations increases as fast as the number of data
segments to be sent.

CORBA also provides for two other modes of invocation: oneway and deferred syn-
chronous. In the oneway mode, the requesting application does not receive any reply so
that the invocation does not block. However, there is also no way to get a feedback about
invocation success or failure. Furthermore, the oneway mode was underspecified so that a
complying implementation can still be blocking (see [106] and [58] for a detailed discussion
on this issue). The deferred synchronous mode allows for invoking a request asynchronously,
and for polling or receiving a reply later.

The default threading scheme implemented in CASCADE makes use of the deferred
synchronous mode in the following way as depicted in Figure 5.6: There is a single global
receiver thread polling and receiving replies for all of the connectors. A single thread suffices
here because all replies consist of just few bytes. Moreover, if a reply is not processed immediately after its arrival, this will not affect the operation of the system in any significant way. At the same time, there is a sender thread per each connector that invokes the requests in the deferred synchronous mode as the buffering mechanism releases them for sending. Making a single thread service all of the connections is a poor idea because it substantially reduces the concurrency of request invocations: Sending a message is a costly operation because of the copy operations, and it can block due to the lack of buffer space. On the other hand, making multiple threads service a single connector does not make sense as long as there is only one TCP connection opened for this application level connection. Of course, if the connection pool scheme described above is employed, using multiple sender threads for a single connector might improve concurrency. Therefore, CASCADE implements an alternative scheme in which threads are taken from the thread pool described in Section 5.8 to service the connectors.

The downside of using the deferred synchronous mode of invocation is that this mode is only enabled for dynamic invocation, which is more costly than static invocation because of the extra marshaling overhead. However, our experiments with wide area networks clearly show that the marshaling times are negligible compared with network latencies, even for dynamic invocation. Yet, for a network with lower latencies, using a separate thread for each request and invoking requests in the synchronous mode can be better than the above scheme based on the deferred synchronous invocation. Therefore, CASCADE supports this alternative scheme as well. When a separate thread is used for each request, these sender threads are taken from the thread pool and not pre-assigned as in the default scheme.

5.8 Concurrency and Queues Management

In CASCADE, many tasks may need to be executed concurrently: update requests and their results need to be propagated, methods of multiple cached objects have to be executed in parallel, objects can be serialized and deserialized, cache manager might perform some background tasks, etc. The multitasking environment in CASCADE is supported by Java threads. Some of the tasks are assigned a dedicated thread, e.g., for processing in connectors as described in Section 5.7.2. However, most of the tasks are served by a global thread pool on a first come first served basis. The thread pool is highly optimized to minimize the amount of necessary synchronization and contention on monitors. It is configurable with respect to the number of threads it may create and to the way the threads are allocated to various tasks.

Note that within the context of a JVM, the thread pool of CASCADE typically co-exists with other threads that are maintained by the ORB implementation. The latter are used for servicing CORBA request invocations. All blocking application requests (e.g., queries and blocking updates) are performed solely by ORB threads. However, some invocation requests have to be non-blocking, e.g., non-blocking application updates or update propagations between DCSs. When a DCS A transfers an update to another DCS B, it would be inefficient to block the requesting thread of A until B completes an update execution because this thread could be used for other tasks. Unfortunately, the current version of CORBA does not
support an asynchronous reliable messaging mechanism\textsuperscript{6}. Thus, an incoming update request is inserted into a queue, the invocation completes and control is returned to the invoking thread. Queued requests are serviced later by the thread pool of CASCADE.

Each cached object has its own processing queue as depicted in Figure 5.6. The type of requests depends on the consistency policy used with this object: Without the Total Order consistency guarantee, each object queue merely contains requests for a method execution. When the total order implementation is deployed, the queue for the root of the hierarchy contains requests for ordering and execution while queues for all other hierarchy nodes contain execution or deserialization requests that have been already ordered. In the case of

\textsuperscript{6}Specifically, a oneway invocation has underspecified semantics, event service does not provide a reliable delivery, and notification service is not portable and it is not supported by most ORB implementations. However, the next version of CORBA is supposed to include an asynchronous messaging mechanism.
Total Order, the queue is sequential: processing of the next request starts only after the previous request has been completely handled. This is done to preclude a concurrent execution of updates on a single object. Furthermore, a read-write lock is used to prevent executing updates and queries concurrently but to allow execution of concurrent queries.

Note that an incoming request may need to be put both in the processing queue of an object and in the queues of connectors to be propagated further through the object hierarchy. The queues are not synchronized because most of the time we would not like to delay a request execution until this request is propagated to the son DCSs and vice versa. However, in absence of such a synchronization, we need to provide more complex algorithms for those infrequent occasions when a new DCS joins the hierarchy and obtains an object copy, or when a previously evacuated object is brought back. For example, if an update execution takes significantly more time than a request propagation, it might happen that some update request $U$ has been already propagated to all of the neighbors but it is still in the processing queue. In this case, it is not reflected by the object state yet. If a request for an object copy arrives at this point, we cannot simply send the current object state because the requesting DCS will never receive $U$. Delaying the transfer of an object copy until $U$ is executed does not solve the problem since further updates can be propagated and inserted into the processing queue at that time. Furthermore, it does not make a lot of sense to temporarily prevent further update propagation by blocking the queue in a connector. This is because a) a connector does not maintain a queue of update requests for each individual object so we would need to block update requests for all of the objects, and b) a request for an object copy may travel through the hierarchy up to the root (see Section 5.5), so the requesting DCS is not necessarily a neighbor of the DCS which transfers an object copy.

To solve this problem in CASCADE, we send some of the processing queue content (i.e., update requests) along with an object state in reply to an object copy request. Thus, the requesting DCS will need to invoke the received updates on the received object copy before it becomes ready to process further requests on this object.
Chapter 6

Applications

In this chapter we describe several applications that can benefit from our caching service. We have designed the prototypes of these applications and implemented them to assess the effort required to build applications with CASCADE as well as to analyze the requirements of these applications from a caching service. In some cases, we were also able to evaluate the performance gain coming from using caching.

6.1 Yellow Pages Service

If we had to name a single class of applications that would gain a great benefit from using CASCADE, it would be electronic information services. A typical representative of this wide application class is a yellow pages service. The object to be cached here is a catalogue of named information items. Clients can register their names along with the information they would like to publish about themselves. They can update information about themselves and they can also query information about some particular name or search the catalogue for particular information.

The performance of this application can be boosted by using CASCADE because its consistency guarantees are weak and because queries are typically far more prevalent than updates. Under the realistic assumption that no two clients will try to register under the same name, updates from different clients are not interrelated. While synchronous catalogue update provides better fairness, it will usually be permissible for two clients to temporarily see different views of the catalogue. After all, even the phone company does not deliver a new yellow pages book at the same time to all houses. This eliminates the need for the Total Order guarantee. In addition, if the DCS hierarchy reflects the geographical location of the servers, then according to the principle of locality, if some client registers information at some DCS, it is more probable that queries about this information will be issued to the same DCS. Therefore, additional consistency guarantees (such as Session Causality) are not required either. Thus, this application requires only Update Propagation and FIFO of Writes out of all elementary consistency conditions.

Furthermore, this is not a mission critical application so that it does not have very strong reliability and security requirements. For example, it is acceptable if a user has to reissue the same query request more than once because of a DCS failure as long as it happens very rarely. There is no need to encrypt updates and queries because the published information
is normally not confidential. However, perhaps it would be desirable to authenticate update requests so that updating a record would be restricted to the record’s owner.

6.2 Distributed Bulletin Board

In a distributed bulletin board, users can post events, or poll the board for recently posted events. Also, it is possible to define topics, in which case a user may look for postings in a specific category, or get a listing of all postings. Similarly to the yellow pages application, a distributed bulletin board does not have strong reliability and security requirements. However, its consistency requirements are stronger because a bulletin board should preserve Local Causality of event postings so that a follow-up posting is always seen after the original posting it is referring to. Yet, this application does not require the most expensive Total Order and Read Your Writes consistency guarantees as explained in Section 4.4.3.

6.3 Tickets Reservation Service

In contrast to the Yellow Pages Service, this service requires classical strong consistency [72]. The object to be cached here is a ticket reservations database. Clients can book a ticket and they can also query information about ticket availability. The service should not allow two clients to book the same ticket. This implies that all reservation requests should be totally ordered.

It should be noticed that queries in a ticket reservation service can benefit from dirty copy consistency [12]. Loosely speaking, a dirty copy consistency query returns a value resulted from all locally known updates, even if some of these updates were not ordered in the global sequence yet. Consider the following situation: Some client reserves a ticket at some DCS, and then another client attempts to reserve the same ticket at the same DCS before the previous update has been propagated. Then a dirty query can already show that this ticket is not available before the actual update propagation. Note that the probability that the second client contacts the same DCS is relatively high because of the above mentioned principle of locality.

This application also has strong security requirements. Since financial information such as credit card data is involved, all communication should be encrypted and the access to the service should be authenticated. This is a typical example of an application that can deploy a distributed transaction for booking purposes.

Note also that our caching service can be used for both airline ticket reservations and theater reservations. Here, again, the fact that we cache active objects is instrumental as the rules for performing reservations in both cases are different. For example, airlines allow for over-booking, but over-booking might not be acceptable in theaters.

6.4 Distributed Scorched Earth Game

While working on the CASCADE system, we took an already existing application and made an attempt to port it to CASCADE. The application was a Distributed Scorched Earth game [5]. In this combat game, each of the players owns a tank and the goal is to destroy
the tanks of all other players by shooting at them. The game is turn-based: When the game
starts, the implementation sets a random order on the participants. Then, the first player is
given the right to shoot. Each time, the turn passes to the next player whose tank has not
been destroyed yet. The game continues until there is only one tank left.

In the original implementation of [5] there is a single server holding the state of a game.
Each time a turn is made, a corresponding GUI based application client sends a message to
the server that is responsible for distributing this state update to all other clients participat-
ing in the game.

To port this game to CASCADE, we registered the state of a game with CASCADE as
the object to be cached. The design challenge was to choose the way in which the clients
would receive the information about a game progress. One possibility was to introduce a
support for pushing information to clients into CASCADE. To this end, a client would need
to register with its DCS and indicate its interest in receiving game updates, or notifications
about them. However, this would make DCSs stateful with respect to application clients
and limit the scalability of the caching service. Instead, in the solution we choose the client
polls its local DCS periodically in order to verify whether a game update has been received.
While this puts a burden of transferring frequent query requests on the local links between
a DCS and its clients, a DCS remains stateless with respect to the clients.

Note that the game is turn-based and a participant is allowed to make her turn only
when she receives a message that the previous turn has been completed. Thus, this applica-
tion requires only Local Causality, FIFO of Reads and Reads Before Writes consistency
guarantees. In particular, it does not require the Total Order guarantee. In fact, all turns
are totally ordered by the application itself.

This order induced by the application may create an impression that some of the per-
formance gain due to the weak consistency requirements is lost. However, the extent to
which it is correct is strongly affected by the distribution of game participants over DCSs.
This is because passing a turn between two players communicating with the same DCS is
efficient whereas passing a turn between two remote clients is not. Thus, if multiple game
participants are in the same DCS domain, the efficiency of using caching depends on the
order in which the turns are passed among them. Specifically, much better performance will
be obtained if they are ordered consecutively and not interleaved with players from other
domains.

To compare the performance of the ported application with the application using a cen-
tralized server, we implemented an automatic client that plays according to some predefined
deterministic algorithm and does not require human interaction. Thus, if the same clients
participate in a game and they are ordered in the same way, the game is deterministic from
the beginning and its result (including the number of turns it takes) is always the same,
whether run with CASCADE or a centralized server. Therefore, we can run the same game
once with CASCADE and another time with a centralized server and obtain a fair comparison
of performance.

Note, however, that the number of turns the game takes is not strictly proportional to the
number of participants. Sometimes, adding participants to the game can make it shorter.
Therefore, we tried to analyze the dependency of the total time the game takes on the number
of turns rather than the number of clients. Figure 6.1 compares the results for a centralized
server and a system consisting of three DCSs. We can see three dependencies here: the
upper curve is for the centralized server, the lowest curve is the “best” case for CASCADE when the participants were ordered in such a way that all players in the same domain were ordered consecutively. The middle curve corresponds to the case in which players from all domains were randomly interleaved in their turn order. We can see that even in the case of random interleaving, the gain from using CASCADE is significant.

6.5 Caching Multimedia Objects With CASCADE

Traditionally, multimedia systems were implemented in an ancient, monolithic style. However, distributed multimedia applications have recently started adopting object and component-based philosophies. In such applications, the multimedia data is wrapped in a middleware object, e.g., a CORBA object. In this paper, we propose using CASCADE for caching such multimedia objects.

Since different multimedia objects may include various combinations of video, audio, still images, and text, and different audio and video streams might be coded under different codecs, they might also expose different interfaces. Moreover, the code for implementing the interfaces is likely to be different. An important feature of CASCADE is that it can cache objects that include both data and code. This makes CASCADE a good candidate for caching multimedia objects, since it can serve as a unified framework for caching objects of various types, without imposing restrictions on the codecs or accessing code of these objects.

Typically, multimedia objects are quite large so that a cache does not have enough space
to hold the entire object. This calls for developing novel cache management schemes that are able to support partial caching (i.e., caching part of the object) and partial evacuation. For example, it is possible to remove the end of the movie from a cache, while keeping the start of the movie [56, 96, 111]. The rational for this is two fold. First, by having the beginning of the movie in the cache, one can guarantee fast startup time. While the movie starts playing, it is possible to fetch the rest of it. Also, often users start watching a film, then realize it is not what they wanted, and terminate. Thus, the start of a movie is used more often than its end.

Another approach common in MPEG-1 and MPEG-2 [1] is to compress the movie more eagerly, at the expense of reduced quality. The rational here is similar to the above. In order to browse movies, it is enough to keep a low quality version, and only when the movie is really selected, it is fetched in high quality. MPEG-4 and MPEG-7 [1] offer even greater partial evacuation capabilities [103]. Specifically, MPEG-4 and MPEG-7 allow coding a movie as multiple objects, e.g., background vs. the rest, etc. Each of these objects is compressed separately, thus resulting in a much better compression ratio for the same video quality. This can be similarly used for partial evacuation. For example, one might evacuate objects that appear less important, or ones that are changing often, while keeping the others. MPEG-7 would be in particular useful here, as it allows tagging and prioritizing such movie objects in a standard way.

CASCADE supports pluggable replacement algorithms, and in particular it can accommodate algorithms that are specifically optimized for multimedia objects. In particular, multimedia objects can implement the partial evacuation interface. In this case, when the cache runs out of space, it can use a combination of partial evacuation of objects that support this interface with evacuation of other objects to make room for newly needed objects\(^1\). Note that the caching algorithm can learn which objects allow partial evacuations by looking at the interface repository. Example of protocols dealing with caching partial contents can be found, e.g., in [56, 84, 111].

CASCADE takes care of automatic update propagation to keep cached copies in a consistent state. This is done in a manner which is fully transparent for the application. It should be noted that while in web caching only a single copy is updated, CASCADE supports concurrent updates on several cached copies. This can be exploited by applications such as video conferencing.

Other issues that must be dealt with in creating a caching mechanism for the Internet include scalability, security, and manageability. More precisely, the main scalability issues arise from the need to support a large number of clients, spread in a large geographical area. Thus, the main problem here is to reduce the number of requests a server needs to address, and placing copies of objects close to clients. On the other hand, one cannot afford to place copies of objects in places where they are not needed. Recall that CASCADE dynamically builds a separate caching hierarchy for each object. Also, if an object is not used for a long period, and there is space shortage in the DCS it was cached on, the object will be replaced.

As for security, one of the main issues is access control for intellectual property protection, and preventing objects from interfering with each other. We are currently carrying out a work to address these issues in CASCADE as described in Section 8.3.2.

\(^1\)However, ensuring fairness in such systems is a challenging problem.
6.6 Utilizing CASCADE for Efficient Implementation of Other CORBA Services and Facilities

The CORBA Event Notification Service (ENS) provides a convenient paradigm for developing distributed applications, in which the producers of information are completely decoupled from its consumers. More specifically, information producers post objects/events on designated channels. In return, this invokes listeners that have registered to receive these types of events on the corresponding channels. The naming service is another important CORBA service, as it allows for locating an object implementations based on its name. We believe that both the ENS and naming service can greatly benefit from the caching service, in terms of scalability and responsiveness, as discussed in the introduction. Also, due to the nature of these services, it is sufficient to require weak consistency for them.
Chapter 7

Performance Measurements

In this chapter, we describe our experiments with CASCADE. We tested the system in a real wide area environment to assess the speedup in terms of service response times (i.e., duration of a remote request invocation) compared with a centralized solution. Furthermore, these tests allowed us to gain insight about network level properties such as throughput. We also ran simulations in a local area network to evaluate hit rate and other cache properties. The results of these simulations and cache management tests are presented in [18]. That study is complementary to the results described in this chapter. Furthermore, running simulations is essential for estimating other characteristics that require a massive computation (e.g., scalability and hierarchical properties). In this chapter, we present some preliminary results of such simulations.

7.1 Running CASCADE in a Wide Area Network

7.1.1 The Testing Environment

We ran CASCADE on nine hosts at the following geographically disperse locations over the Internet: MIT, at the Massachusetts Institute of Technology, Cambridge, MA; UCSD, at the University of California San Diego; AM, at Vrije University of Amsterdam, Netherlands; PT, at the University of Lisbon in Portugal; TECH, at the Technion, Haifa, Israel; HUJI, at the Hebrew University, Jerusalem, Israel; SU, at the Sydney University in Australia; BR, at the Federal University of Paraiba in Brasil; and TW, at National Taiwan University in Taiwan. The host in Amsterdam runs SunOS operating system on the Sun Sparc architecture. All other hosts run Linux on the Intelx86 architecture, ranging from Pentium II with 128Mb of memory to multiprocessor Pentium 4 with 1Gb of memory.

We used the Visibroker ORB in all the tests described in this thesis. Recently, we started testing CASCADE with ORBacus and other ORBs. While the absolute values obtained in the new measurements slightly differ, all effects we observed in the tests done with Visibroker apply to other ORBs as well. Furthermore, we run all Java programs with the Sun’s implementation of JDK, ranging from JDK 1.2 to JDK 1.4. In Section 7.1.3 we briefly discuss the impact of choosing a particular Java implementation on the caching service performance.

In order to track the latency and loss rate of the links connecting these sites, we peri-
odically ran ping from each host to all other hosts, sending a sequence of ping probes once every five minutes. The ping process was monitored by a crontab. The tests were run for a total of about 150 hours spread over 4 months, during different days of a week and different hours of a day. While ICMP-based ping probes differ in many ways from TCP-based CORBA communication, these measurements provide some intuition about the behavior of the underlying network.

<table>
<thead>
<tr>
<th>From To</th>
<th>AM</th>
<th>PT</th>
<th>MIT</th>
<th>UCSD</th>
<th>SU</th>
<th>BR</th>
<th>TECH</th>
<th>HUJI</th>
<th>TW</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>54(97)</td>
<td>141</td>
<td>170</td>
<td>310(376)</td>
<td>367(470)</td>
<td>86(170)</td>
<td>87(170)</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>PT</td>
<td>55(98)</td>
<td>142</td>
<td>209</td>
<td>311(413)</td>
<td>221</td>
<td>122(193)</td>
<td>124(198)</td>
<td>347(389)</td>
<td></td>
</tr>
<tr>
<td>MIT</td>
<td>145</td>
<td>141</td>
<td>83</td>
<td>227</td>
<td>198(287)</td>
<td>180</td>
<td>190</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>UCSD</td>
<td>167</td>
<td>203</td>
<td>82</td>
<td>178</td>
<td>294</td>
<td>251</td>
<td>255</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td>308(376)</td>
<td>242</td>
<td>180</td>
<td>449</td>
<td>396</td>
<td>400</td>
<td>296</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TW</td>
<td>315</td>
<td>341</td>
<td>220</td>
<td>176</td>
<td>293</td>
<td>460</td>
<td>385</td>
<td>391</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Round trip times and loss percentage of the links

The results are presented in Table 7.1. This table is not complete: We were unable to run ping measurements between some hosts because of firewalls at some of the locations that filtered out all ICMP communications. In some cases, we were pinging a different host at the same University instead of the host behind the firewall on which CASCADE was tested.

The table shows both average round trip times (RTT) in milliseconds and average loss rate in per cent. It should be noted that due to the great variability in both characteristics, the average values are not very indicative. This is particularly true for loss rates whose distribution was continuous, i.e., without multiple prominent peaks and base regions. For some links, indicated by a star next to the average loss rate, the distribution was almost uniform, that is, each time we were running a sequence of pings we had a high chance of getting a different value. This effect is especially noticeable for the link from SU to BR where the loss percentage ranged from 10 to 48 depending on the time of the day, day of the week etc. Furthermore, we observed several transient losses of connectivity for short periods of few minutes.

The RTT values were somewhat more persistent. Here, most of the time we observed similar values within a deviation of 3 to 20 percent, depending on the link. However, sometimes the values jumped up drastically and remained high for the duration of few days, most probably because of router or link upgrades. We did not count such infrequent but significant deviations when computing the average but rather wrote them in parenthesis next to the average values in the table. Furthermore, we did not count the periods of these changes when evaluating the response times of an object cached with CASCADE, as presented in Section 7.1.3 below.

This comes in line with research of end-to-end Internet path properties, e.g., in [47, 109, 126]. These works show that end-to-end Internet performance is extremely hard to analyze,
predict, and simulate. Another distinguished feature in the Internet is that message latencies on different communication paths often do not preserve the triangle inequality because the routing policies of Internet routers often do not choose an optimal path between two hosts. For example, it follows from the above table that communication from AM to BR would be significantly faster if routed through PT. This phenomenon, in particular, provides a motivation for creating deeper hierarchies because it may prove better to connect new DCSs to intermediate nodes rather than directly to the root of the hierarchy.

7.1.2 The Experimental Hierarchy

In all our tests, we ran a single DCS on each host. All experiments comprised two steps: a) building the hierarchy, and b) executing requests on the objects. We did not consider object evacuations from the cache; it was the subject of a separate study in [18]. While we ran our experiments with several hierarchies that included the hosts listed above, we observed no significant differences in network level communication properties or in the speedup of object response times compared with the case of a centralized single object\(^1\). Thus, we chose to present results for one particular hierarchy depicted in Figure 7.1.

We chose the root of the hierarchy to be at AM because the sum of the distances from AM to all other hosts is smaller than the sum of the distances from most other sites (e.g., from BR to all other hosts). Furthermore, the host in AM is a strong machine that can handle the extra burden of request ordering and processing incurred on a root node. The hosts in the same geographical locations were made connected to each other, e.g., HUJI was a child node of TECH. Note that the hierarchy is four level deep, SU being at the 4th level of the hierarchy. Thus, we were able to analyze the impact of a hierarchy depth on latencies and other communication properties.

\(^1\)However, a considerable number of sufficiently large hierarchies should be considered to evaluate the hierarchy construction mechanism and scalability properties. Since coordinating between a large number of hosts in the Internet and managing large hierarchies is difficult, these properties should rather be tested in simulations.

Figure 7.1: The experimental hierarchy
7.1.3 Invocation Time Measurements

The main goal of response time tests was to compare request invocation times of an application using a centralized object with the same application using CASCADE. Since the root of a hierarchy corresponds to the centralized object location in the original application, we first deployed a centralized object in AM, ran a set of application clients at all other locations on a set of workloads, and collected the results. Then we constructed the hierarchy described in Section 7.1.2 with the root in AM and ran the same application clients on the same set of workloads. This way, we were able to make a fair comparison of invocation times for the centralized object and for CASCADE.

The Test Application and Workload

The implementation of the testbed involved three parts:

- A sample object whose interface has both update and query methods, the size of an object being a parameter;
- A mechanism for generating a synthetic workload, i.e., a sequence of update and query requests to be run from all locations. A workload was generated off-line prior to starting each test and made publicly accessible through the web;
- A test client application that can be run at any location. This application downloads a workload from the web in the beginning of the test and then simulates it by invoking requests on either a centralized object or a cached copy.

The application also logs response times on a disk. Furthermore, DCSs also kept a log for their operations including processing times. To reduce the amount of time spent on disk writes and to make it negligible compared to invocation times, only operations that took more than 5 seconds and average values over a considerable period of time were recorded.

The most challenging part was generating adequate workloads because there are very few publicly available traces of CORBA, or other object-based distributed systems operating in wide area networks. Note that numerous web traces and traces for various file systems are of no help since typical sequences of requests for method invocations can fundamentally differ from requests for web document or file data retrieval. When we started our experiments, we were trying to distribute all requests between different domains by using various distributions (e.g., Pareto, Zipf etc.). Fortunately, we found out that the exact type of distribution has very little impact on performance. Furthermore, the speedup of using CASCADE compared with the centralized case is almost linear: If there are \( a \) requests originating from domain \( A \) and \( b \) requests originating from domain \( B \), we can run a test with the same schedule in which all \( a + b \) requests originate from \( A \) and measure a speedup \( S_{PA} \). Then, we can run another test in which all \( a + b \) requests originate from \( B \) and measure a speedup \( S_{PB} \). The experiments we have performed show that the speedup of the system with requests originating from both \( A \) and \( B \), will be \( (S_{PA} \times a + S_{PB} \times b)/(a + b) \). This indicates that the bandwidth on the links was adequate for our tests, at least when the request production rate was not high. Thus, in all our runs described below, requests were distributed uniformly over the nine origin domains.
Based on our initial experiments, we concluded that among all workload parameters it was a) percentage of updates, b) request length, and c) request production rate, that had the strongest influence on invocation times. The percentage of updates and request length were ranging from 10% to 20% and from 10b to about 15Kb, respectively. Since we wanted to separate the study of network latencies and behavior from the study of a load on CASCADE servers, we ran these tests under a moderate request production rate (10 requests per second from each domain). Moreover, our object implementation was relatively lightweight: An execution of each method took few milliseconds, which matches the request execution times for a web server, a file server, or information retrieval systems but does not consider objects providing computational services. The size of the object was ranging from few bytes to about 100Kb.

Factors Contributing to Invocation Times

Since CASCADE is a complex multilayered system, there are many factors that affect the service operation. In addition to the environmental settings, cached object implementation and application workload discussed above, there are also internal CASCADE parameters (e.g., thread pool tuning) and policies defined by the application for the cached object.

In particular, consistency policies described in Sections 4.4.2 and 5.3 substantially affect invocation times. For weak consistency policies (i.e., when the Total Order condition is not part of the policy), all operations are executed on a local DCS without being blocked on costly request transfer between DCSs. In our experiments, each invocation took 5 to 20 milliseconds in this case, depending on the host where the DCS runs and the exact environment settings. On the other hand, a remote invocation on a single centralized object can take up to 400-500 milliseconds for slow WAN links as can be seen from Table 7.1 and results presented below in the next section. Thus, for applications with weak consistency requirements, the service time can be reduced by a factor of 20-100 by employing caching solutions such as CASCADE. This speedup does not depend on the percentage of updates in the workload. Furthermore, it increases for long requests that enlarge request transfer and service response times for a centralized object but have much less effect when requests are invoked on and replies are returned from a nearby cached copy.

Note that in the test application, clients do not switch between different DCSs. Recall from Section 5.3 that in this case, FIFO of Reads, FIFO of Writes, and Reads Before Writes implementations do not block on request transfer between DCSs. Thus, if the application does not require the Read Your Writes guarantee, the same speedup of 20-100 times will be achieved even when the Total Order implementation is employed. Again, this order of magnitude holds for any workload as explained above. Thus, the most interesting situation is when both the Total Order and Read Your Writes guarantees are required by the application. Below we consider only this case.

Another object policy that strongly affects invocation times is the type of propagation done by the consistency implementation, by requests or by objects. Since the object implementation used in the test was not computationally heavy and the state of the object was longer than update requests, which is consistent with most object based distributed applications, propagating requests proved to be considerably more lightweight than propagating object states. The difference was especially prominent for large objects: We found out that
the object size plays a very important role for object propagation while having almost no significance when requests are propagated.

In many aspects, propagating object states for large objects is similar to propagating update requests for long requests because both lead to intensive communication between DCSs. However, propagation by objects requires object serialization and deserialization that may be relatively heavy operations in Java, depending on the platform, Java implementation etc. On the other hand, when objects are propagated, connectors perform data merges and replacements as described in Section 5.7.2, substantially reducing the amount of propagated data. This tradeoff becomes particularly interesting for higher request production rates when both effects have strong impact on the service performance. Our experiments suggest that when object states and requests are of comparable size, propagating objects can turn slightly more efficient when DCSs run on powerful server machines. However, the difference was not very significant. While all performance results that appear below mention only the effect of request lengths, they can also indicate the influence of an object size for object state propagation.

It should be noted that the length of request results (i.e., the return values of an invocation) is much less important than the length of the request itself (i.e., the size of input parameters). This is because requests themselves are disseminated between all DCSs in a
hierarchy while invocation results are only returned to the client that invoked the request.\(^2\)

Figure 7.2 shows a typical distribution of time between the operations that constitute a request invocation. This distribution corresponds to the case when an update request is invoked on a local DCS, propagated to the root, ordered and applied by the root, and the results are propagated back to the local DCS. Another situation covered by this distribution is when a query request is invoked on a local DCS, and this query request has to block until a previously issued update request arrives. All operations performed by the ORB (i.e., marshaling, request interception and dispatching, etc.) were taking less than one percent of the total time in all our tests. Processing the request by CASCADE (that includes the time when the request is pending in various queues) and the proper object execution were typically taking 3 to 8 percent, depending on the factors we described above throughout this chapter. The network latency constituted most of the overall invocation times, just as we expected.

Typically, request processing time at a DCS was 2 to 25 milliseconds. Most of this time went to serialization/deserialization operations for object propagation, garbage collection, synchronization between Java threads (in particular, monitor contention), and waiting while these operations are performed for preceding requests. The distribution of time between these operations as well as the absolute duration values strongly depended both on the computational resources of the DCS host (i.e., the amount of memory and CPU speed) and the particular implementation of JVM. For example, serializing a 10Kb object takes about one millisecond with JDK 1.4 on Pentium IV with 1Gb of memory but up to few dozens of milliseconds with JDK 1.2 on Pentium II with 128Kb of memory. It is well known that the garbage collection mechanism was substantially improved in JDK 1.3. At the same time, it is somewhat surprising that the implementation of a JVM has at least the same effect on efficiency of serialization and monitor contention as the platform settings. For example, we initially ran a DCS at MIT with JDK 1.2 and contention on Java monitors for some workload was about 10 milliseconds per request, which was more than on other hosts with exactly the same JDK implementation. When we replaced JDK 1.2 with 1.3, the monitor contention times for the same workload dropped to just a couple of milliseconds per request. In some cases, we observed an improvement in monitor contention times following a Linux upgrade to a newer version.

On one of the hosts, logging onto a disk turned out to contribute few milliseconds to the processing times. This was because the log was written on a non-local disk through NFS. We eliminated this factor by changing the log location to a local directory.

**Results of Response Time Tests**

We now present the results of running over a hundred of tests, each test containing about 100,000 invocation requests. These tests were conducted at different hours of the day and different days of the week during a period of about four months.

As described above, each test included a workload consisting of both update and query requests that originated from all locations. Figure 7.3 presents a typical situation corresponding to a sample test period for one of the locations. It shows how invocation times

\(^2\)To be precise, when a request is executed only once on the root DCS, returned values are also transferred between the root DCS and the DCS that accepted the client request.
measured by the client that was running at this location evolve in time. While update and query requests were interleaved, the graph shows only invocation times of update requests that included propagation to the root of the hierarchy and back to the local DCS. The times are given in milliseconds, and the requests issued in this test were 7Kb long. We can see how big the standard deviation of response times was (about 30 percent). We can also see that the first half of the period was quite unstable with longer response times and deviations. This was probably due to a load on some of the intermediate routers in this Internet path. During the second half of the period, the situation became more stable and deviations appeared to be more moderate.

Figure 7.4 shows five distributions of response times as measured at MIT for the tests with short requests (about 100b long). Recall that the MIT node is at the second level in the hierarchy and it was directly connected to the root at AM. As in the test of Figure 7.3, update requests in this experiment were blocking, i.e., response to the client was delayed until the request is propagated to the root, ordered and received back. The first distribution refers to the case of a centralized object, when there is no difference for update and query requests. All other distributions describe response times when using CASCADE. The second and the third distributions refer to the workload in which 10 percent of all requests were updates. They describe invocation times for queries and updates, respectively. The fourth and fifth distributions also refer to queries and updates but when update requests constitute 20 percent of the workload.

We can see that virtually 100 percent of query requests have invocation times of 0 to 20 milliseconds, independently of the number of update requests in the workload. The distributions for update requests and for the centralized object were slightly more heavy...
tained but still more than 90 percent of invocations fall in the same 20 millisecond time interval. These three distributions appear almost identical. Thus, we can conclude that for short requests, each individual update invocation takes about the same time independently of the percent of updates in the workload. Furthermore, the response time for an update is always the same, whether for a centralized object or for an object cached with CASCADE.

Figure 7.5 shows analogous distributions measured at BR. BR is at the same level in the hierarchy as MIT but the link from BR to AM is more lossy. Here only distributions for a centralized object and for update requests with CASCADE are presented. Again, the distributions are almost identical but the distribution for a cached object is slightly more heavytailed. This indicates that the loss rate on the link has a bit stronger negative effect in the case of CASCADE than in the case of a centralized object.

Figure 7.6 presents several distributions of response times for query requests. The workloads on which these distributions were received varied in the length of requests and percentage of updates. While this graph presents results for the link from MIT to AM, results for all other links were very similar. We can see that percentage of updates had no effect on the average invocation time of a query. On the other hand, longer requests caused an increase in the average invocation time by 22 milliseconds. Furthermore, the distributions for longer requests is noticeably more heavytailed.

Response times for long update requests are presented in Figure 7.7. The two distributions are for workloads with 10 and 20 percent of update requests, respectively. These distributions are fundamentally different from the corresponding ones for short requests.
only they are quite heavytailed but we can clearly see two distinctive peaks (in fact, there is another peak in the interval above 1 second but this peak is small and not so prominent as the first two). This is due to message losses on the link: for a long request, there is a high probability that at least one TCP packet belonging to this request gets lost. Such a packet loss causes a severe increase in the invocation time for the request. The second peak corresponds to the case when a request execution is delayed by a single segment retransmission. The third peak is due to multiple packet losses. Note that if two consecutive packets are lost, TCP can detect this and retransmit the packets at the same time, so that the invocation time increases by only one packet retransmission period. This explains why the second peak is almost as big as the first one while the third peak is substantially smaller.

In addition, we can see that percentage of updates in the workload has small but noticeable effect for long requests, unlike for short ones. Specifically, the distribution for 20 percent is slightly more heavytailed and its average is 20 millisecond higher compared with the distribution for 10 percent. The reason for this is an increased amount of communication, which causes more packet losses and out of order arrivals as shown in Section 7.1.4.

We also tried to investigate how invocation times with CASCADE are affected by the location of the DCS in the hierarchy. Figure 7.8 depicts the histogram of update invocation times for SU, which is at the deepest (fourth) level in the hierarchy, being connected to AM by two intermediate nodes. The three presented distributions correspond to the cases of a centralized object, short update requests with CASCADE, and long update requests, respectively. We can see that the distribution for short requests has a very distinct peak interval into which over 90 percent of invocations fall. This is only a bit less than the percent of invocations that fall into the peak interval for a centralized object. The exact location

Figure 7.5: Invocation time histogram for the Brasil to Amsterdam link
of these two peaks relatively to each other is determined by how efficient the constructed hierarchy is. For our hierarchy, the path to the root through UCSB and MIT was slightly longer than the path of a direct connection to AM. In addition, processing at each node adds some 5 to 20 milliseconds to the overall invocation time, once for the propagation to the root once for the backward propagation.

However, the distribution for long requests was drastically different: It was almost uniform. As in the case of other distributions, the most influential component of an invocation time was the network latency. The network latency for each of the three links on the path from SU to AM had a distribution similar to the one presented in Figure 7.7 but with different peak locations. When these three distributions are superimposed on each other to contribute to the invocation times at SU, the resulting distribution is almost uniform.

All results presented so far describe tests in which the Nagle algorithm was disabled. This is the default CASCADE behavior as explained in Section 5.7.1. Figure 7.9 compares the performance of CASCADE at MIT with and without this algorithm, both for short and long update requests. It can be clearly seen how strong the negative effect of the Nagle algorithm is. Sections 5.7.1 provides the explanation for this phenomenon. For short requests, the average response time becomes almost twice longer when the Nagle algorithm is enabled. For long requests the difference is less drastic but it is still about 100 milliseconds. Furthermore, the distributions with the Nagle algorithm are considerably more heavytailed.

It is reasonable to ask whether CASCADE consumes significantly more bandwidth by disabling the algorithm and implementing its own buffering mechanism. This issue is discussed in Section 7.1.4.
Figure 7.7: Response times for long update requests

Figure 7.8: Effect of the hierarchy depth
Figure 7.9: Impact of the Nagle algorithm for a non-lossy link (MIT to AM)

Figure 7.10: Impact of the Nagle algorithm for a lossy link (BR to AM)
Figure 7.11: Average invocation time as a function of update request size

Figure 7.12: Average invocation time for deeper hierarchies
Figure 7.10 shows the impact of the Nagle algorithm for the link from BR to AM, which has a higher loss rate compared to the link from MIT to AM. Here, the loss in performance for short requests is even more prominent. However, for long requests, the difference in invocation times with and without Nagle deteriorates.

Figure 7.11 shows the dependency of the average request invocation time with CASCADE on the update request size. The presented results are for a workload with 20 percent of updates and for the link from MIT to AM. Note that unlike all graphs presented earlier in this thesis, this figure makes no distinction between update and query requests, i.e., each point here represents an average time over all requests in the test.

The two displayed curves correspond to blocking and non-blocking updates. For blocking updates, response to the client is delayed until the request is propagated to the root of the hierarchy and back. A non-blocking update request returns immediately but a subsequent query will have to block in order to satisfy the Read Your Writes consistency guarantee. In addition, the figure shows the average invocation time for a centralized object.

As we would expect, using non-blocking update requests is more efficient than blocking because it allows for a pipelining between propagating a previously issued update request and handling of a subsequent query. The dependency on the size of update requests is almost linear in the both cases. This dependency turned out very similar for a workload with 10 percent of updates but the slope of the curve was about twice smaller.

Clearly, the curves are quite steep. This steepness is due to the fact that requests are pushed through the hierarchy resulting in a lot of packets being sent in a bursty fashion. The communication for a centralized object is much more regular as described in Section 7.1.4. Therefore, the curve for the centralized case is much less steep. While the advantage of using CASCADE is very prominent for short requests, the curves intersect at some point. Thus, starting from a certain request size, the invocation time for CASCADE can become worse than the invocation time for a centralized object. The location of the intersection point depends on many factors but especially on the percentage of update requests in the workload.

Fortunately, in many applications update requests are short and most data is transferred in response to query requests. For example, in the Yellow Pages application (see Section 6.1), an update request typically contains the new values of only few fields. At the same time, a query request may retrieve a lot of information. This gives a definite advantage to the caching solution of CASCADE over a centralized object. Yet, we continue exploring the end-to-end TCP behavior and various flow control techniques to make the dependency curves less steep.

Finally, Figure 7.12 presents analogous dependencies for the SU location, which is at the fourth level in the hierarchy. The results here are very similar to those for the MIT location. The advantage of CASCADE for short update requests is even more prominent. However, the curves for CASCADE are even more steep because of the way in which latency distributions for intermediate links are superimposed on each other.

7.1.4 Assessing Network Level Properties of the Communication

To gain deeper insight about the pattern and properties of DCS-to-DCS communication in the Internet, we monitored the TCP packets sent at the network level. Since such moni-
monitoring requires administrative privileges, we could do it only at TECH where we had such permissions. Specifically, we ran the `tcpdump` program to collect information about all packets that were sent or received by the DCS running at TECH. Then, we used the `tcptrace` program [97] to process the collected statistics and present it in a comprehensible form.

Figure 7.13: A TCP segment size graph

Figure 7.13 depicts the graph of a TCP segment size for the communication between the DCSs at TECH and AM. The zigzag curve shows the segment size for each individual packet, whereas the straight horizontal line shows the average segment size up to the current point of the connection. We can see that both the average and maximal allowed segment sizes were about 1420b.

Figures 7.14, 7.15, 7.16, and 7.17 describe all “activity” on the connection, i.e., all sent and received packets. These graphs refer to the experiments in which requests were 7Kb long. The X-axis shows the time when events (i.e., sending or receiving a packet) occur, and the Y-axis places these events with respect to the sequence number space of the connection.

Figures 7.14 and 7.15 describe the communication with a centralized object, the latter being a closeup of the former. Figure 7.14 depicts two curves: The lower curve corresponds to sent and received packets whereas the upper curve represents the receive window as advertised by the receiver. Vertical sections of the lower curve indicate the sent packets. The upper and lower limit of the sections gives the sequence number of the first and last byte in the segment. The diamond-topped sections show packets that were sent with the PUSH bit set (in particular, packets are sent with this bit when the Nagle algorithm is disabled). Horizontal sections of the lower curve track the ACKs returned by the receiver and point to the beginning of the sender’s window.

Small ticks above the upper curve indicate a receive window advertisement that is the same as the previous advertisement. Since the distance between the upper and lower curves is quite big, we can conclude that the buffer space on the side of the receiver was not a problem in these tests.
Figure 7.14: A time sequence graph for the centralized object.

Figure 7.15: A time sequence graph for the centralized object (a closeup).
Figure 7.16: A time sequence graph for CASCADE communication

Figure 7.17: A time sequence graph for CASCADE communication (a close-up)
We can see that the communication pattern is quite regular: update requests are sent by a single client once at a time, after a response to the previous request arrives. There is very little “unusual” activity such as retransmissions, packets arriving out of order, etc.

This makes a contrast with Figures 7.16 and 7.17 that describe the communication between DCS servers in CASCADE, the latter being a zoom into a part of the former. Little ticks just below the horizontal sections (e.g., the long horizontal section in Figure 7.17) indicate duplicate ACKs. For CASCADE communication, we can see a lot of packets arriving out of order (that are indicated by “O”), selective ACKs (denoted by “S”), and triple duplicate ACKs (denoted by “3”). Such an irregular communication affects performance for long requests and explains the steepness of the curves in Figures 7.11 and 7.12. At the same time, the upper and lower curves are quite distant from each other just as they are in Figure 7.14 for the centralized object. Thus, there was no problem with TCP buffers on the receiver’s side.

Some approximate idea of the congestion window (as estimated by TCP) can be derived from Figures 7.18 and 7.19. More precisely, these figures show the amount of outstanding data, i.e., the number of unacknowledged bytes. The measurements were taken for a centralized object (Figure 7.18) and CASCADE communication (Figure 7.19). The zigzag curve represents the instantaneous outstanding data samples, that is, the number of bytes that are unacknowledged at a given point. The upper smooth line is the average of the outstanding data up to that time. The lower smooth line represents the weighted average of the outstanding data up to that time. The samples are weighted by the time for which they exist.

Again, we can see that the amount of the outstanding data was bigger and with much stronger variation for the communication in CASCADE compared with the centralized object. This goes in line with irregularity of the communication pattern in CASCADE that renders the communication more prone to packet losses and packets arriving out of order. Also, note that that amount of outstanding data does not necessarily indicate the real available capacity of the links: If the communication is not very intensive, the congestion window will not grow beyond the required size. Even if the communication is intensive, the congestion window still remains an internal TCP heuristic. This approximation may be imprecise, e.g., because of TCP’s conservatism w.r.t. packets losses.

Finally, Figures 7.20, 7.21 and 7.22 show throughput measurements for the centralized object, a CASCADE communication when Nagle is enabled, and for CASCADE communication without Nagle, respectively. Of course, these measurements were done on the same workload (20 percent of update requests that were 7Kb long). Note that while the DCS at TECH has a child DCS at HUJI, these results take into account only the communication between TECH and AM. This makes a fair comparison because we also do not take into account the requests sent from HUJI to the centralized object.

The dots in these graphs represent “instantaneous throughput”, which is defined as the size of the segment divided by the time elapsed since the previous segment arrived. Due to clock granularity, there tends to be a lot of banding in these points. The zigzag line is an average of the previous ten dots. The smoother line is the average throughput over the life of the connection to that point (i.e., the total number of bytes divided by the total number of seconds).
Figure 7.18: An outstanding data graph for the centralized object

Figure 7.19: An outstanding data graph for CASCADE communication
Figure 7.20: A throughput graph for the centralized object

Figure 7.21: A throughput graph for CASCADE communication when Nagle is disabled
Figure 7.22: A throughput graph for CASCADE communication when Nagle is enabled

We can see that the throughput for the centralized object and for CASCADE communication is almost the same in these particular experiments. However, in general the comparison is not quite straightforward. For a centralized object, all requests travel forth and back over WAN links, whereas only update requests are propagated by CASCADE. On the other hand, every request in CASCADE is flooded through the hierarchy, i.e., sent over multiple links compared with a single link for a centralized object. This flooding propagation may result in an intensive communication, especially for large hierarchies with high fan-out number of nodes. Note, however, that this effect is mitigated by using flow control optimizations in connectors (see Section 5.7.2). Yet, its impact may be significant when the workload contains a high percent of update requests that are produced at high rate.

We can assess the influence of the Nagle algorithm on throughput by comparing Figures 7.21 and 7.22. The throughput appears to be slightly (about 10 percent) higher when Nagle is disabled. As we saw in Section 7.1.3, disabling the Nagle algorithm leads to a substantial improvement in response times. Furthermore, we found out that the total duration of the same tests with the same workload was a bit shorter in this case. This explains the effect of a higher byte throughput without Nagle.

In addition to byte throughput, it is important to consider packet throughput because it is the number of packets that creates a load, e.g., on routers. As we expected, CASCADE sends a higher number of short packets when Nagle is disabled because requests are sent earlier, without waiting until a full-size segment can be sent. For short requests (significantly smaller than one TCP segment) that are produced by the application at low rate the difference could be up to a factor of two or three because CASCADE does not wait sufficient time for the requests to accumulate in buffers prior to sending them. However, the communication is lightweight in this case and the throughput is low anyway. On the other hand, for more intensive communication that is induced by longer requests that are produced at a higher rate, the observed difference was not so significant. In particular, for the test presented in
Figures 7.21 and 7.22, the CASCADE implementation was sending only 15 percent more packets when the Nagle algorithm was disabled.

7.2 Running Simulations in a Local Area Network

In this section, we describe our experiments of running simulations with CASCADE in a local area network. Most of these simulations were for the purpose of evaluating various cache management policies as presented in [18]. However, we have also conducted few experiments to understand the extent to which the system is scalable. Here, we present some preliminary results in this direction.

The experiments were carried out on a system consisting of 50 DCSs. These DCCSs were uniformly distributed over 10 machines, all of which were Windows 2000 powered Intel based Dual 1.7 GHz Pentium 4 Xeon with 1Gb RDRAM. The network connecting the DCSs was a 100 Mbps Fast Switched Ethernet. To emulate the setting of a wide area network, we inserted artificial delays on invocations between DCSs. While in general it is extremely difficult to reproduce the network behavior of the Internet by using such an emulation (see [47]), it is deemed sufficient for our specific goal of testing the scalability of CASCADE under high loads.

All tests included 1500 objects and started from building the hierarchies. All DCSs participated in all of the hierarchies in such a way that each DCS was a root for 150 hierarchies. The testing application was the same as for WAN experiments described in Section 7.1.3. However, the request production rate was much higher (about 100 requests per a second on each of the DCS). Each workload contained 100,000 requests. We used the request propagation policy throughout the experiments (as opposed to the object state propagation policy).

Of course, these experiments created a heavy load on the machines and exploited 100 percent of the CPU resources. However, we observed that the system remained stable and performed quite well under these conditions. Furthermore, the simulated network latency still contributed a considerable percent to the overall invocation time (30 to 70 percent, depending on many factors). In fact, the most constraining limit we were hitting was the amount of virtual memory on the machines where the application clients and DCSs were deployed. Since a virtual machine is a heavy process that consumes a lot of memory, we could not afford running more DCSs and constructing bigger hierarchies on these machines. Furthermore, we could not run experiments with object state propagation because object serialization and deserialization require a lot of memory. In the future, we would need to deploy the system on 50 to 150 machines in order to test the service scalability and resilience to load in simulations more approximated to practical settings.
Chapter 8

Conclusions and Future Work

8.1 Summary of The Results

Designing and deploying distributed applications that operate in wide area networks is inherently a difficult task. While modern object oriented middlewares pose a potential for the construction of truly ubiquitous distributed services, the long and unpredictable latencies of the Internet as well as its unreliability, complicate the realization of this potential.

In this thesis, we have presented CASCADE, a general purpose caching service for distributed CORBA objects. As explained in Chapter 1, we believe that a CORBA caching service is necessary in order to build scalable, highly available distributed CORBA services, that can be accessed from anywhere in the Internet. This has served as a motivation for defining the features of CASCADE such as preserving the original application programming model, portability, and configurability with a variety of policies. Furthermore, it has been the reason that CASCADE allows caching of active objects that include both data and code.

As we described in this thesis, CASCADE uses a separate dynamically built hierarchy for each object. The use of a hierarchy conserves bandwidth, contributes to the scalability of the caching service, and facilitates update ordering. Having a separate hierarchy for each object contributes to even better scalability and allows for more efficient consistency maintenance protocols. However, the real challenge lies in reconstructing already existing hierarchies while continuing to support the required semantics such as consistency guarantees. This dynamic hierarchy reconstruction would help to make the hierarchies more balanced and reduce the distances between the nodes.

To address the specific needs of the applications operating in wide area networks, we have introduced a framework of composable consistency guarantees. The main idea behind composable consistency conditions is the ability to express the application consistency requirements in a simpler and more precise way so that the application does not have to pay the overhead of unnecessarily strong consistency maintenance. We have identified a list of elementary consistency conditions, and proved that their different compositions yield most well known consistency conditions as well as many other conditions that we have shown to be useful for commonly used distributed applications. We have also discussed the implementation of this framework of consistency conditions in CASCADE.

We have shown that consistency implementation schemes used in many applications such as the Web and shared memory systems are not directly applicable to our settings. For
example, most cache implementations in various distributed software systems use lazy invalidation based update propagation to maintain consistency of cached copies. In contrast, CASCADE employs a push based update propagation scheme for consistency support. Note that in object oriented client-server middlewares an object state typically takes much more space than update data. Therefore, it is typically more efficient to push update requests to all caches than to retrieve the entire object state later when it is required. Thus, push based update propagation lends itself to such settings better than state invalidation schemes.

We have considered several distribution applications that exist in practice and discussed the ways in which they could benefit from using CASCADE. We intentionally took applications with different requirements to show that CASCADE is flexible enough to meet the requirements of each of these applications. Furthermore, we took the sources of a real world application and gave them to senior undergraduate students to port it to CASCADE. This student project took only several months to complete, which made us believe that porting a real world application to CASCADE is quite a feasible task.

Finally, we have presented various experiments we conducted with CASCADE, in order to assess the invocation request time, the network level properties of communication between the CASCADE entities, the service scalability etc. In particular, we have presented the performance measurements of running CASCADE in a wide area network. While the performance gain from using CASCADE strongly depends on the consistency and other applications requirements as well as the application workload and many other factors, we have shown that the speedup in object response time due to caching can be few dozens of times compared with a centralized service implementation. However, this speedup deteriorates as the length of update requests increases. This phenomenon is partly because of the increase in bandwidth consumption due to the push based update propagation mechanism that is used in the consistency algorithms of CASCADE. However, this issue still requires a more detailed investigation.

Thanks to the conducted experiments we have been able to draw conclusions about various flow control techniques that can improve the performance of distributed applications that employ TCP based communication. Furthermore, we have identified few TCP related performance issues such as the Nagle algorithm and conservative message loss detection. Middlewares intended for wide area applications will be able to achieve substantially better performance by coping with these issues.

8.2 CORBA Lessons Learned

Object by Value CORBA has the advantage over many commercial RPC systems in that it allows object references to be manipulated as regular data type values in a straightforward manner. In particular, an object reference can be passed as an argument of a method invocation on another object. However, other parameter passing semantics are not supported by CORBA. Specifically in CASCADE, there is a need to pass objects by value when the object is cached at some DCS. Currently, this is implemented with the aid of Java serialization. The shortcomings of this approach are that (a) we are limited in the choice of programming language, and (b) the object state is passed as a sequence of bytes. This does not stay in line with object-oriented approach.
While CORBA 2.3 introduced the object-by-value standard to tackle this problem, this standard is not mature enough and its applicability is limited in several respects. For example, the deployment environment has to be highly homogeneous so that the object layout in memory would be the same on the object sender and receiver. Furthermore, CORBA lacks the ability to forward an incoming request to another server, i.e., to pass it by value. This ability is important for CASCADE that executes the same update request at multiple locations and totally orders update requests in order to achieve consistency. Below we explain how we circumvent the lack of this ability with the aid of interceptors.

**Interceptors** Use of interceptors is vital for CASCADE implementation. Interceptors allow CASCADE to perform a set of operations transparently to the client applications: to encrypt and sign requests and replies, to maintain consistency and to pass implicit per-request policy parameters.

As we previously mentioned, the CORBA standard does not allow for forwarding an incoming request to another server. One possibility for CORBA would be to provide this ability at the level of interceptors. For example, the proprietary Visbroker 3.2 interceptors allow for capturing the whole GIOP request message and for extracting the part of the message that constitutes the body of the request. This can be used in the following non-straightforward but efficient way: When DCS A receives a request to be propagated to another DCS B, A calls an appropriate *InternalRequest* interface method on B and passes the GIOP request message body as a parameter. If B is to apply this request, it creates an instance of CASCADE class that implements *org.omg.CORBA.ServerRequest* and that is initialized with the received GIOP request body.

However, this solution is non-portable both because it is not supported by the Portable Interceptors standard and because it could lead to little endian - big endian incompatibilities in some ORBs other than Visbroker: If A and B have the same endian order, the standard CORBA input/output streams are used for marshaling/demarshaling: The request message body is written to the *org.omg.CORBA.portable.OutputStream* by calling *write_octet_array*, *org.omg.CORBA.portable.InputStream* is created out of the OutputStream, and the typed data is read from this InputStream. However, if A and B have different endian order, we had no choice but to use input and output streams implemented in CASCADE.

An alternative way of redirecting a request would be to use request-level interceptors, to demarshal request parameters into *anyS*, and to call an appropriate method on B, passing the list of these *anyS*. This way is more portable as it complies with the Portable Interceptors standard. However, in this solution the types of *anyS* would be also passed on the wire, while these types would be absolutely unnecessary.
8.3 Future Research

8.3.1 Dynamic Hierarchy Reorganization

Ideally, the hierarchy organization should be adaptive to different environmental changes, such as load increase on nodes and links as well as their possible failures. When such changes occur, it might be desirable or necessary for the hierarchy to perform self-reconstruction. To achieve this, a caching service implementation can adopt the techniques used in peer-to-peer systems such as [108] or [127] for similar purposes.

However, the real challenge lies in continuously providing semantic guarantees in face of dynamic hierarchy reorganization. This is particular difficult in CASCADE because of its push-based update propagation and complex queue management as described in Section 5.8. Even removing a single intermediate tree node requires complex synchronization between various local queues and queues at neighbor nodes. Furthermore, local queues may contain part of the neighbor nodes’ state such as update requests intended for propagation. If the reconstruction algorithm attaches the removed node to a different part of the hierarchy, the content of the queues may need to be resent to the new neighbors of the node. Thus, reorganization may turn out an extremely expensive operation, especially when communication between the nodes is extensive.

This raises the question of whether hierarchy reconstruction is desirable for transient failures such as temporary load increase or temporary link failure. Obviously, the answer depends on the duration of the change. In our experiments with wide-area environments, we observed frequent transient changes (up to few seconds) combined with rare lasting or permanent changes. Apparently, distinguishing between the two cases is essential for efficient caching service operation. However, it is deemed very difficult to perform detection at the application level without accessing network level information or interacting with network level protocols.

We are currently trying to port our caching service to the platform provided by the Scribe publish-subscribe system [31]. This system is based on Pastry that provides typical benefits of peer-to-peer application such as scalability, adaptation and dynamic self-reorganization in case of changes. Like CASCADE, Scribe uses hierarchies for data dissemination. While Scribe does not solve the above problem of maintaining semantic guarantees in face of dynamic changes, it allows the application to register and implement upcalls and invokes the corresponding upcall when some event occurs during the execution. We are currently investigating if it is possible to enforce various consistency conditions through these framework of upcalls. An alternative way would be to re-implement a new publish-subscribe system with flexible framework of semantic guarantees directly on top of a peer-to-peer platform like Pastry.

8.3.2 Support for Security

A distributed service like CASCADE, especially a service operating in wide area, must offer security support that is essential for many distributed applications (see Section 6). We grasp that CASCADE should provide the following security measures:

From the standpoint of an application developer, the use of some security measures taken by CASCADE should be determined by a per-object security policy that can be specified
by the object creator when the object is registered with the caching service. Security policy may designate the use of the following security features:

**Protection from malicious clients:** A malicious or misbehaving client can easily cause the denial of service condition by filling up its local DCS cache with object copies, or by bombarding its local DCS with invocation requests. To address this problem, CASCADE should support the following two features: a) service access control, so that only authorized clients can access the DCS, and b) a limit on the maximal total size of cache that can be occupied by the object copies requested by each authorized client (or an authorized group of clients).

**Protection from malicious objects:** While Java objects have a more limited access to system resources compared with objects written in lower level programming languages, executing a malicious bytecode can still be harmful for the system. If no restrictive steps are taken, objects can read information from a disk, communicate with the outside world, interfere with the operation of other objects etc. Thus, there is a need to isolate each object by running it in a *sandbox*.

However, creating a separate virtual machine for each of the objects is too costly and cannot be considered a practical approach. The typical Java technique for dealing with this issue is employing a *Security Manager* that intercepts all potentially dangerous operations initiated by the object. In this way, it is feasible to prevent the object from accessing a disk or opening a network connection (note that an object does not need to open connections in order to send replies to invocation requests). Unfortunately, using security managers affects the system performance and does not guarantee full protection. For example, an object can grow uncontrollably unless the expensive reflection mechanism is run after each small change to evaluate a new object size.

Another step that can be taken in this direction is accepting only crypto-signed Java classes for caching and verifying the signatures. Such trusted classes do not need a close security surveillance as other classes do. Furthermore, mechanisms for signing classes, verifying the signatures and preventing unauthorized code modifications are well known in modern cryptography. In Java, they are achieved by a cooperation between the class loader and security manager.

**Per-application security:** In addition to the measures required to protect the service, a particular application may have its own security requirements (for example, see Section 6). The use of such application required security features should be determined by a per-object *security policy* that is specified by the object creator when the object is registered with CASCADE. Specifically, the following steps may need to be provided:

**Encrypted communication:** If this feature is requested for an object, all communication between the object and its clients should be encrypted. Both public and secret key cryptosystems should be considered when determining the most suitable encryption scheme.

**Object access control:** This feature allows the object creator to specify different object access policies for different clients or groups of clients. A separate access policy
can be specified for each individual object method, for a group of methods, and for the whole object. Clients can be authenticated using the same digital signatures that are employed for protecting the service from an unauthorized access.

A natural way for implementing the encryption/decryption in CORBA and passing digital signatures is by using CORBA interceptors. In addition, it would be worthwhile to evaluate the applicability of standard security tools like CORBA Security Service and Secure Socket Layer (SSL) for implementing the security policy.

It would be also interesting to consider arbitrary (i.e., Byzantine) failures that are known to be a very hard problem. There exist a few object replication systems dealing with such failures such as Rampart [105], Phalanx [79], and the BFT project [2]. Unfortunately, the mechanisms used in these systems are quite complex and they do not always comply with the goals of CASCADE. In particular, scalability of these mechanisms is typically limited to local area networks.

### 8.3.3 Further Research on Consistency Conditions

In the course of our work on CASCADE, we have implemented several interesting combinations of the basic consistency conditions as described in Section 5.3. However, for the sake of performance, we have created a single monolithic implementation for each chosen combination, rather than having a truly modular implementation. The main obstacle in providing such a modular implementation is that some of the basic conditions can be implemented much more efficiently when it is known that other conditions are also provided. The challenge is to generate automatic optimizations for a given composition of conditions, based on a set of implementations, one for each condition. Such optimization can either be done in compile time, or ideally, on-the-fly.

Another open problem is to generalize the framework to more generic operation types, and to be able to capture other consistency conditions such as release consistency [52], entry consistency [88], and hybrid consistency [16]. Also, an interesting question is whether there exists a basic condition that is weaker than linearizability, which can be combined with sequential consistency to yield linearizability. An even grander challenge is to arrive at a complete set of basic consistency conditions. That is, be able to show that any consistency condition can be provided as a combination of a subset of these conditions, and that each of these conditions is necessary for implementing at least one of the currently known consistency conditions.

It would be also interesting to further explore the locality of consistency conditions. To this end, there is a need to generalize the definition of a consistency condition to cover such known conditions as release consistency [52], and hybrid consistency [16]. Furthermore, our locality results still leave a “gap of uncertainty” with respect to consistency conditions incomparable with PRAM. While such conditions are probably too weak to be applicable in most practical settings, it would be interesting to have a criterion that allows for an easy locality verification of such consistency conditions. Finally, it appears that there is a tight relation between locality of a condition and its property of being blocking or nonblocking. A nonblocking condition implies that a pending request invocation is never required to wait for another pending invocation to complete. In particular, [59] shows that linearizability is
both local and nonblocking, while serializability [72] is both nonlocal and blocking. Thus, it would be interesting to investigate the relation between the two properties in a formal way.

8.3.4 Resource Management

CASCADE was initially designed as a strict caching system that would be for distributed objects what Web proxies are for documents. Thus, in the beginning we limited our consideration to communication infrastructure, consistency maintenance, and cache replacement algorithms. However, the more progress we made in our work, the more far away we were getting from our initial vision. At some point we understood that caching running objects that contain both data and code is fundamentally different from caching non-structured data in many respects.

To start with, caching of code poses several challenges that do not exist in standard data caching, as described in Section 5.6. Efficient code caching requires discovering the dependencies between various pieces of an object code and between the code and the data to which this code refers. Thus, there is a need in data flow analysis and other tools typically used in programming languages.

Second, running multiple objects concurrently requires a multithreaded environment. CASCADE implements a pool of threads that are intended for execution for invocation requests (see Section 5.8). We also synchronize between thread accesses to cached objects and other data shared by the threads. However, it would be interesting to investigate various thread scheduling algorithms that would take into account specific requirements of individual objects and object methods (e.g., execution of some methods may be particularly CPU hungry while other methods may allocate more memory space).

Most important, there is much more to cache management of general objects than just cache replacement algorithms, which only consider the amount of space each object takes. Since objects are running, they consume CPU and other computational resources that should accounted for by the cache management mechanism and cache replacement decisions. Ideally, a caching service like CASCADE should have a resource management module that would constantly monitor the consumption of various resources by each cached object. This module would provide feedback both to the scheduling mechanisms that schedules requests for execution and to cache replacement protocols that would base their decision on this information. Thus, we envision that a caching service would eventually evolve in an architecture that resembles high-level distributed operating system.

8.3.5 Interrelated Objects

Frequently objects are semantically interrelated and one object refers to another. For example, an object can have a set of resources attached to it. When either referring or referred object is relocated or cached, a question arises of what happens with the second object. [60] explores this problem in a different context, but its results can be potentially applicable to the framework of the Caching Service. This work defines five types of object references:

Link: For this type of references objects do not have to be collocated. If one of the objects move, the second object remains at the same place. The reference can be remote (this type of references is used in CORBA).
Pull: Objects have to be collocated. When the referring object migrates, it "pulls" the referred object along, so the referred objects also moves to the new place.

Bi-directional Pull: Objects have to be collocated. When one of the objects moves, it pulls the other object as well.

Duplicate: Objects have to be collocated. When one of the objects is relocated, the other object is replicated and the new replica moves to the same place.

Stamp: Objects have to be collocated. When one of the objects moves, the copy of another object is created at the new place.

[60] also discusses various applications and their need in different types of references. It would be interesting to examine the applicability of using different types of object references for the Caching Service. In addition, there will be a need to introduce new types for partial object caching.

Another idea for determining object resources is to give the object a possibility to define its own companions by calling an appropriate object function upon caching or migration. This function should return a set of references to objects that the object being migrated would like to bring with it to the new location. However, this approach leads to defining a new programming model for the object implementation since the object has to implement the callback function.

Interesting questions arise when two interrelated objects are both put into cache. In particular, consistency guarantees that would lend themselves to such a situation have not been explored yet. For example, consider the case when a method is invoked on a cached object copy, and the code of this method implementation calls a method of another object which is also in a cache. The latter method, in its turn, invokes a request on yet another object in cache etc. It is challenging to define a meaningful consistency condition that would be adequate for such a chain of nested calls that may be quite long and may have cycles.

8.3.6 Porting to Wireless Ad-Hoc Networks

Hand-held computing devices are becoming increasingly popular. For example, today's high-end devices have the same (theoretical) computing power and memory capacity as high-end desktops of merely five years ago. Judging from the development of laptops, this trend is likely to continue at an even faster pace in the next few years. These powerful computing devices come equipped with commodity operating systems, such as Linux and Windows CE, which will progressively resemble their desktop OS counterparts as the devices become even more powerful. At the same time, hand-held and palm-held computing devices are being equipped with wireless and cellular communication capabilities, whose bandwidth is gradually approaching standard LAN speeds.

These developments open the way for a whole new set of mobile applications that operate in ad-hoc networks, or in other words, networks of devices that are formed in an ad-hoc manner, and utilize them for combined efficiency. Examples of such applications include, e.g., multi-party ad-hoc auctions, various kinds of collaborative applications etc. Like in any other distributed environment, these applications require middlewares and tools that would
facilitate development and deployment of services. Thus, we can envision an increasing need in replication and caching for ad-hoc platforms.

We have recently started a project whose goal is to adapt the design of CASCADE to the settings of ad-hoc environments. Most of the lower level modules in Figure 3.4 require cardinal changes because, e.g., flow control in ad-hoc network conceptually differs from that in the Internet environment. Furthermore, the hardware broadcast nature of wireless communication makes simultaneous propagation to multiple destinations more efficient but requires different application level routing mechanisms. However, many ideas used for the implementation of the upper layers, such as consistency policy implementation, are still applicable. Since the environment is even more dynamic than the Internet in terms of network connectivity, the design should devote even more considerable thought to handling link failures and supporting disconnected operation.

The service should also be more lightweight because the computing power of hand-held devices cannot match the capability of a standard server hardware yet. This provides a strong motivation for exploring the effect of thread synchronization, garbage collection and other operating system activities on the performance. However, the research in this direction has just begun. Furthermore, we anticipate that the techniques used, e.g., in wearable computing for reducing software footprints will also be important for building generic services in ad-hoc environments.
Bibliography


113


116


117
Appendix A

CASCADE Core API

The Domain Caching Server API can be divided into two parts:

A.1 Registering with CASCADE

An object is registered with CASCADE under its unique name which should be supplied by the application. This name is used by CASCADE for object identification.

This part of the API consists of the following primitives used for the interaction between a client and its local DCS:

\textit{register\_object(objName, object\_by\_value, policy\_obj)} initializes the object’s hierarchy and the info object according to the (per-object) policy object \textit{policy\_obj}, puts the object into the DCS under \textit{objName}, and makes this DCS the root of the hierarchy. Normally, this call returns the object reference to the cached object. However, it can also terminate abnormally and return “NAME\_ALREADY\_IN\_USE” or “NOT\_ENOUGH\_SPACE” CASCADE exceptions;

\textit{unregister\_object(objref)} destroys the object’s hierarchy. This method should be called only on the root DCS;

\textit{copy\_object(objName)} If there is an object registered under \textit{objName}, this call causes DCS to join the object’s hierarchy, obtain a copy of the object registered under \textit{objName} along with its per-object policy, and return the object reference to this object copy to the client. Otherwise, this call terminates abnormally and throws “NO\_SUCH\_OBJECT” CASCADE exception;

Note that there is no API support for explicit deletion of a cached copy. Instead, the decision to remove a cached object can be accepted by the DCS itself. However, this can happen only if no client has issued a request for the object during a pre-defined timeout (see precise description in Section 5.5).

A.2 Working with Cached Objects

For the purpose of the caching service, we assume that all object methods can be categorized as \textit{updates}, which change object data, or \textit{queries}, which only retrieve this data. In this framework, an
object implementor should explicitly specify for each object method whether it is an update or a query.

This part of the API consists of the following primitives:

\texttt{update\_object(objref, request\_data, policy\_obj, consistency\_data)} causes DCS to update the object according to the request\_data and per-request policy object policy\_obj. For some policies the request should be first propagated among other DCSs before being applied at the local DCS (see more detailed explanation in Section 5.3).

\texttt{query\_object(objref, request\_data, policy\_obj, consistency\_data)} causes DCS to query the cached copy of the object and send the retrieved information to the client.

\texttt{lock\_object(objref)} causes DCS to obtain a global lock for this object (see description in Section 5.4).

\texttt{unlock\_object(objref)} causes DCS to release a previously obtained global lock for this object.

In CASCADE there is an alternative way to update and query cached objects instead of using update\_object and query\_object. An application can directly invoke a method on the cached object as it would do on a usual (non-cached) CORBA object. In this case, CORBA interceptors and the service context field of the standard GIOP request message header are used to pass additional information about the request (like per-request policy) transparently for the client. This way we preserve the standard programming model of working with CORBA objects.

As explained in Section A.1, CASCADE can delete a cached object transparently to the client. Therefore, when issuing a request to a cached object, the client should be prepared to get the CORBA standard "OBJECT\_NOT\_EXIST" exception \(^1\). If this happens, the client can catch the exception and re-issue the \texttt{copy\_object} request.

\(^1\)Some old ORB implementations incorrectly raise the "INV\_OBJREF" exception in this case.
Appendix B

Caching Service IDL

// CachingService.idl

module CachingService {
    typedef sequence <octet> ObjImage; // object code and state
    enum PropagationPolicy {Push, Pull}; // propagation of update result
                                      // from root to leaves
    enum ConsPolicyType {Weak, Strong, Group}; // types of consistency
                                              // strong - using total order,
                                              // weak - without total order,
                                              // group - for a group of objects

    struct StrongPolicyParam { // strong policy parameters
        boolean dirtyCopy; // is dirty copy maintained
        boolean propagateObjects; // propagate objects or requests
        boolean supportNonblockingUpdates; // keep counter information
                                            // about other DCSs
        PropagationPolicy pPolicy;
    };

    struct GroupPolicyParam { // group policy parameters
        boolean dirtyCopy;
        boolean propagateObjects;
        PropagationPolicy pPolicy;
        string groupName;
    };

    union ConsPolicy switch (ConsPolicyType) { // consistency policy
        case Strong: StrongPolicyParam spp; // defined for the object
        case Group: GroupPolicyParam gpp;
    };

    struct UpdateId { // used for consistency
        string name; // maintenance
        long count;
    };
}
typedef sequence <UpdateId> Timestamp;

struct StrongPolicyState {                        // consistency state
    long globalCounter;                  // for strong consistency
    Timestamp ts;
};

union ConsPolicyState switch(ConsPolicyType) {     // consistency state is passed
    case Strong: StrongPolicyState sps;    // to a DCSs when it caches
    case Group: Timestamp gts;             // this object for the first
        // time
};

struct UpdatePolicy {                             // per-update policy
    boolean blocking;
};

enum QueryPolicy {Strong, Weak, Dirty};           // per-query policy

union MethodPolicy switch (boolean) {             // per-request policy
    case TRUE: UpdatePolicy up;
    case FALSE: QueryPolicy qp;
};

struct MethodInfo {
    string name;
    MethodPolicy mp;
};

typedef sequence <MethodInfo> Methods;           // list of object methods
                                                   // and their description

enum PersistencePolicyType {UpdateDriven, TimerDriven, ClientDriven};

union PersistencePolicy switch(PersistencePolicyType) {
    case UpdateDriven: long updatesNumber;
    case TimerDriven: long period;
};

/*
*   Object Policy consists of several issue-specific policies
*/

struct ObjPolicy {
    ConsPolicy cp;
    Methods methods;
    PersistencePolicy pp;
    ...
};
exception NotEnoughSpace{};
exception NoSuchObject{};

enum LockType {Read, Write, Update};

/**
 * The following interface is for registering with CASCADE
 */

interface CachingAdministration {
    struct RegisterInfo {
        string name;
        ObjPolicy op;
        ObjImage oi;
    };

typedef sequence <RegisterInfo> RegisterInfos;
typedef sequence <Object> Objects;

exception ObjAlreadyCached {
    string name;
};

Object register_obj(in string name, in ObjPolicy objpolicy, in ObjImage objimage)
    raises(ObjAlreadyCached, NotEnoughSpace);
Objects register_objs(in RegisterInfos infos)
    raises(ObjAlreadyCached, NotEnoughSpace);
void unregister_obj(in string name)
    raises(NoSuchObject);
};

/**
 * The following interface is the main interface for a client application
 * interested to work with a cached object copy
 */

interface CachingRequest {
    Object copy_obj(in string name)
        raises(NoSuchObject, NotEnoughSpace);

    void update_obj(in string name, in UpdatePolicy upolicy,
        in RequestData request, in Timestamp predecessors)
        raises(NoSuchObject);

    void query_obj(in string name, in QueryPolicy qpolicy,
        in RequestData request, in Timestamp predecessors)
        raises(NoSuchObject);
};
void lock_obj(in string name, in LockType locktype)
    raises(NoSuchObject);
void unlock_obj(in string name)
    raises(NoSuchObject);
void save_obj(in string name)
    raises(NoSuchObject);
};

/*
 * The following interface is an internal service interface
 * for DCS to DCS communication.
 */

interface InternalRequest { // details omitted
    ...
};
