MONASH UNIVERSITY
THESIS ACCEPTED IN SATISFACTION OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
ON..... 9 September 2003 .....
.....[redacted].....
Sec. Research Graduate School Committee

GLOMAR: A Component Based Framework for Maintaining Consistency of Data Objects within a Heterogeneous Distributed File System

by

Simon Cuce, BComp(Hons)

Dissertation

Submitted by Simon Cuce

for fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the School of Computer Science and Software Engineering at

Monash University

Monash University

January, 2003

Contents

List of Tables

List of Figures

GLOMAR: A Component Based Framework for Maintaining Consistency of Data Objects within a Heterogeneous Distributed File System

Simon Cuce, PhD
Monash University, 2003

Supervisor: Arkady Zaslavsky

Abstract

Maintaining one-copy equivalence of replicated data is one of the primary tasks of any distributed file system (DFS). This involves ensuring that the results of concurrent operations are made consistent between all replicated data objects. Current mechanisms used to maintain consistency between replicated data objects are usually highly focused towards a strict set of constraints, like hardware, timeliness, correctness, availability and/or reliability. However, with current DFS environments exhibiting a multitude of different constraints and scenarios, current concurrency control and consistency maintenance mechanisms are unable to adequately adapt to all possible constraint variations that can be experienced.

The proposed and developed GLOMAR framework resolves this lack of adaptability. This framework allows for the creation, co-existence and management of different concurrency control and consistency maintenance mechanisms under a single DFS implementation.

GLOMAR achieves this by abstracting the concurrency control and consistency maintenance functionality from the operating system and/or application and re-implementing it using a component-oriented architecture. This abstraction is referred to as the *Relationship Component* and is responsible for encapsulating actual concurrency control and consistency maintenance functionality, the context of a particular component and what replicas (files) are governed.

GLOMAR also provides a middleware layer for handling run-time management of Relationship Component implementations. The primary purpose of the middleware layer is to select the "most appropriate" Relationship Component to handle consistency maintenance of replicas, based on the current scenario and constraints exhibited by the DFS.

As part of determining the feasibility of GLOMAR, a full implementation was built. Included within this implementation were a number of Relationship Component implementations, including components for handling the constraints of mobility-enabled environments. The resulting system illustrated how a component based framework for maintaining consistency of data objects within a heterogeneous DFS was achieved.

GLOMAR: A Component Based Framework for Maintaining Consistency of Data Objects within a Heterogeneous Distributed File System

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Simon Cuce
September 4, 2003

Acknowledgments

The enormity of a PhD is not something that can be completed alone. The influence and sacrifice of friends and family owe much to the success of this work. For this reason, I would like to thank the people who made this possible.

Firstly, much of my gratitude goes to my supervisor Arkady Zaslavsky. Not only did he give me insight and direction into the academic process, but whose personal crusade to ensure I finished was the primary reason why this work was completed. For this I am eternally grateful.

I would also like to thank peers and staff members of the University for their support. Firstly, I would like to thank Christine Mingins for all her work during the latter stages of my thesis submission. I would also like to thank Dean Thompson whom could only be described as Mr Reliable. My thanks goes out to my office mates, past and present, including, Chee Yeen Chan, Nick Nicoloudis, Troy Milner and the irrepressible Daniel May. Not only do I thank them for their support, but also their friendship. Others whom I cannot afford to forget include, Damien Watkins, Trent Mifsud, Megan Seen, Shonali Krishnaswamy, Hugo Leroux and Peter Stanski. You made the process and pain of a PhD bearable. Finally, of the University people I want to thank, one can never forget the Administration and Technical staff, past and present. In particular, I would like to thank Michelle Ketchen, who always made the time to personally handle any issues that arose. Thanks so much.

I would like to thank Jignesh Rambhia and Bing Hu for their work on the Twin Transaction Model implementation. Also I would like to thank Dan Fay of Microsoft Research for his support of my work. My thanks goes to Carolyn and Bryan Payne, who through their generosity, allowed me to utilise their holiday home for periods of uninterrupted writing. Without this, I am sure I would not have done as much or as well.

However, the biggest thanks goes equally to three people. These people were instrumental in me undertaking this adventure and ensuring I completed. These people include my Mum and Dad, who supported me longer than they had too. Without their support I would have never gone down this road. The third

person I must thank is Sonja Payne. Her sacrifice has made this work possible. My loving thanks goes to all of you.

Simon Cuce

Monash University
January 2003

Outcomes

Journal arising from this thesis include:

Cuce, S and Zaslavsky, A. (2003) Supporting Multiple Consistency Models for a Mobility Enabled File System using a Component Based Framework. *Special Issue of MONET on Mobile and Wireless Data Management* (Accepted for Publication) Vol 8, No. 4. August 2003.

Award arising from this thesis include:

Cuce, S (2002) GLOMAR: Adaptive Consistency Control for Distributed File Systems. 3rd Place in the *The ACM International Post Graduate Competition.* Held at SIGCSE 2002. February 27th - March 3rd, 2002. Northern Kentucky - The Southern Side of Cincinnati, USA.

Publications arising from this thesis include:

Cuce, S and Zaslavsky, A. (2002) Adaptable Consistency Control Mechanism for Mobility Enabled File System. *3rd International Conference on Mobile Data Management* (MDM 2002), 8th to 10th of January 2002. Singapore.

Cuce, S. Zaslavsky, A. Hu, B. and Rambhia, J. (2002) Maintaining Consistency of Twin Transaction Model Using Mobility-Enabled Distributed File System Environment. *5th International Workshop on Mobility in Databases and Distributed Systems* in conjunction with the *13th International Conference on Database and Expert Systems Applications* (DEXA'2002). September 2nd to 6th, 2002. Aix-en-Provence, France

Cuce, S and Zaslavsky, A. (2002) Run-Time File System Consistency Support in Mobile Computing Systems. *2nd Asian International Mobile Computing Conference.* (AMOC02) 14th to 17th of May, 2002. Langkawi, Malaysia

Cuce, S (1999) Conflict Avoidance within a Disconnected Mobile Environment, *Proceedings of the 6th Australian Conference on Parallel and Real-Time Systems* (PARTS99). 29th November to 1st December, 1999. Melbourne, Australia.

Cuce, S and Zaslavsky, A.(1998) Adaptive Cache Validation for Mobile File Systems. *Advances in Database Technology*, Y.Kambayashi, K.Lee, E.P.Lim, M.Mohania, Y.Masunaga (Eds), LNCS 1552, Springer-Verlag, p.181-192, 1998.

Cuce, S and Zaslavsky, A.(1998) Partially Consistent Cache Management Model for a Mobile Environment. *1st Annual South African Telecommunications, Networks and Applications Conference* (SATNAC 98). 7th to 10th September, 1998. Cape Town, South Africa.

Presentations arising from this thesis include:

Cuce, S (2002) GLOMAR: Adaptive Consistency Control for Mobile Enabled File Systems. *The Coda Research Group*, Carnegie Mellon University, 6th March 2002. Pittsburgh, USA.

Cuce, S (2001) A Component Based Framework for Maintaining Consistency of Data Objects within a Heterogeneous Distributed File System. *Distributed System Technology Centre* (DSTC CRC). 6th December 2001. Melbourne, Australia.

Cuce, S (2000) GLOMAR: Adaptive Consistency Control for Mobile Enabled File Systems. *School of Computer Science and Software Engineering*, Monash University. 10th August 2000. Melbourne, Australia.

Permanent Address: School of Computer Science and Software Engineering
Caulfield Campus
Monash University
Australia

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Glenn Maughan and modified by Dean Thompson of Monash University.

Chapter 1

Introduction

This dissertation explores the issues of concurrency control and consistency maintenance within a distributed file system (DFS). It argues that current concurrency control and consistency maintenance mechanisms are too strict and inflexible in their design rationale to exist within current and future DFS environments.

The premise for this lore arises from the inadequacies of existing concurrency control and consistency maintenance mechanisms to fully support heterogeneity. This is due to current environments on which DFSs exist being a collection of varying hardware, software and user requirements, rather than static environments exhibiting predictable behaviour.

This dissertation asserts that multiple concurrency control and consistency maintenance mechanisms should be constructed and scoped to specific scenarios, each existing concurrently and implemented when appropriate. The contribution of this dissertation is a proposed, developed and implemented working system that supports this doctrine.

1.1 Distributed File System

The paradigm known as distributed systems has emerged as a combination of *personal computing*, *time sharing computing* and *interconnecting communication infrastructure* (Satyanarayanan 1990). Personal computing gave users autonomous access to resources. Time sharing computing gave users the ability to share resources and information easily. Interconnecting communication infrastructure allowed for the communication between nodes to take place. When

combined, the result gave users the ability to easily access many shared resources and information, regardless of physical location, however still providing a level of autonomous support. The basic architecture of a distributed system comprises a number of computing nodes connected via a communication network, each with a supportive operating system (Levy and Silberschatz 1990). Messages are then passed between nodes to facilitate the sharing of resources.

The file system is a collection of data objects that are persistent until explicitly destroyed. File systems support four fundamental issues, *naming structure*, *programming interface*, *physical mapping* and *integrity* (Satyanarayanan 1990). The naming structure within a file system allows for the indexing and navigation of file objects. The programming interface provided by the file system allows applications to access a file via a standard set of file system primitives. The physical mapping provides the functionality to ensure a continuous view of a file object is available regardless of the physical storage or media. Integrity ensures that consistency of a file object is maintained regardless of failures, whether hardware and/or software.

Originally, file systems supported a single user performing operations at a standalone node. Such systems as IBM PC-DOS (DOS 1983) and the Apple Macintosh (Apple Computer 1985) are examples of these systems and best define the personal computing paradigm. However, as the need to share resources and information increased, so did the importance of time sharing computing. Systems like Unix (Ritchie and Thompson 1978) allowed multiple users to access a single file concurrently. As a result, file systems were required to ensure that integrity of the file was preserved in the face of possible modifications by concurrent operations. Thus, the process of maintaining integrity became more than just a failure resolution mechanism (as it was with personal computing). However, classic time sharing systems still centralised the storage of file system objects and did not offer autonomous interaction (Satyanarayanan 1990).

The DFS was the merging of the two (distributed systems and file systems), with many distributed users able to share a single file object regardless of their location, but have access to their own processor and permanent storage. By distributing file objects throughout the network, the chance of being partitioned from them due to a communication failure increased. DFSs resolved this by improving the availability of files through *replication* (Berstein, Hadzilacos, and Goodman 1987).

1.2 Replication Basics

Replication is the placement and management of replicated file objects for the purpose of improving the availability, performance and usability of file objects (Helal, Heddaya, and Bhargava 1996). This is achieved by replicating file objects throughout the network so that when the primary file objects cannot be accessed, the secondary replicas can be used to service the request.

This series of events are masked from the originator of the request, resulting in the perception that the operation was successful. The replication mechanism then assumes responsibility for ensuring the operations (or result of operations) are propagated to other replicas. Thus, replication is in most cases a hidden process that is responsible for *Replica Placement* and *Consistency Maintenance* between replicas. This thesis is solely concerned with consistency maintenance of replicas within a DFS.

Consistency Maintenance

While replication improves availability and performance (via replica placement (Kuenning 1994; Lei and Duchamp 1997; Liu and Maguire 1994; Tait et al. 1995)), it also supports how consistency between replicas is maintained. Consistency between replicas is guaranteed through consistency maintenance. Consistency maintenance is the process of ensuring that events that manipulate data on one replica are visible on all others, thus making them correct.

However, availability and consistency are competing objectives, as an increase in the level of consistency will result in a reduction of availability and vice versa. Thus, for a replication mechanism to find a balance, a trade-off between the two is required. The resulting trade-off thus encapsulates the functionality used to provide consistency maintenance, as well as supporting availability.

A trade-off is determined by evaluating the consistency and availability needs of the DFS. In other words, this is done by determining the needs of the user, application and/or hardware constraints. In most cases, a compromise can be found between two extremes, *pessimistic* and *optimistic* concurrency control. With a pessimistic approach, availability is constrained in favour of consistency, as operations must be committed on all replicas prior to being completed. Such an approach does provide the highest level of consistency, but reduces the availability as one failed operation can force the rollback of the operation. This is apposed to a system that supports availability in favour of consistency, (an optimistic approach). With an optimistic approach, operations can concurrently be

performed on separate replicas, regardless of tbe inconsistency that may arise. Only after the operation is complete does the replica's consistency maintenance mechanism attempt to achieve correctness across replicas.

One could argue strongly for each approach and would be justified in their argument. Primarily, this is because the design constraints of the required consistency maintenance mechanism dictates which approach is the most suited. In some cases, availability is fundamental for the system to achieve its design goals, this is the case with mobility-enabled DFSs (Kistler and Satyanarayanan 1991; Tait and Duchamp 1991; Ratner 1998). However, there are cases where correctness is paramount, this is the case with systems that support a high level of file sharing (as shown by the case study (Gill et al. 1994)). Thus, no single approach can suit the multitude of scenarios that can exist. The numerous scenarios generated by heterogeneous systems and applications also compound this.

1.3 Support for Heterogeneity

Originally, DFSs existed on static environments that implemented standardised components, which in turn exhibited predictable behaviour. Such systems were homogeneous with certain characteristics assumed. Consistency maintenance mechanisms when built for these environments assumed the characteristics of the DFS were static, for example, a DFS would exist within a wired network, connecting numerous similar computers. As a result, the underlying consistency maintenance mechanism would be tailored to that environment.

As interconnecting protocols have been standardised (e.g. TCP/IP (Comer 1995)) and the cost of computing power and storage decreased, the degree of homogeneity within current environments has lessened. What resulted were different computing configurations, using different network infrastructures, servicing all types of users, rather than highly specific networks, servicing a particular need. For example, with portable computers, it is not uncommon to use both a modem (when at home), as well as an Ethernet connection (when at work) to access a DFS.

The ramification of increased heterogeneity has meant that the task of building consistency maintenance mechanisms is more complex. This is illustrated by the increase in wireless technology and the incorporation of different computing devices (PDA, Handheld, etc) into the DFS. Each variation increases the

constraints that must be considered when building consistency maintenance mechanisms.

The problem is that current DFSs are faced with a multitude of different scenarios (devices, users, communication infrastructure and applications), all existing within the one environment. However, existing approaches for consistency maintenance are mostly focussed on a small subset of these scenarios. Thus, existing consistency maintenance mechanisms fail to fully exploit the opportunities of the true heterogeneous DFS.

1.4 Thesis Focus

"How to manage the replicated data, providing the levels of consistency, durability and availability needed"

Barbara-Milla and Garcia-Molina (1994)
"Replicated Data Management in Mobile Environments:
Anything New Under the Sun?",
IFIP Working Conference on Applications in Parallel and Distributed Computing,
Page 237.

Research into consistency maintenance within a DFS has focussed on the trade-off between availability and consistency. Many of these systems implement a specific approach to suit a target system. Recently, a lot of work in the handling of the characteristics of mobility has been done (Zaslavsky and Tari 1998; Satyanarayanan 1996; Mukherjee and Siewiorek 1994; Liu and Jr. 1995; Badrinath et al. 1993). However, the narrow scoping of these implementations means that the trade-off between availability and consistency is mostly suited for mobility. Thus, the flexibility and portability of these approaches are limited.

This dissertation claims that rather than creating a consistency maintenance and concurrency control mechanism for a generic scenario, it is preferable to create a number of consistency maintenance and concurrency control mechanisms scoped for specific scenarios all existing concurrently. This claim is based on the following motivations

- **Limited support for fine granularity consistency maintenance.** The structure of existing systems is one that supports a generic approach to maintaining consistency of replicas. Primarily, these systems focus on

a coarse level of granularity, thus ignoring the consistency requirements of devices, environments, applications and users. Thus, this results in consistency for some applications being degraded in favour of others. What is required is an approach that allows granularity to be programmatically defined (whether coarse or fine).

- **Streamline the process of creating consistency maintenance and concurrency control mechanisms.** The process of creating and implementing concurrency control and consistency maintenance mechanisms is made complex by the tight integration with the DFS. Much of the functionality is implemented at the system level, with developers not encouraged to modify or create different mechanisms. Thus, this limitation reduces the effectiveness of a concurrency control and consistency maintenance mechanism, as modification for a specific scenario is difficult.

- **The failure to provide transparent consistency maintenance that is effective and efficient.** Much of the work to provide scenario based consistency maintenance is via an application-aware approach (Satyanarayanan, Noble, Kumar, and Price 1995). For this to work, applications are required to utilise an additional API. As a result, such approaches do not offer support for fine grain consistency maintenance for legacy systems.

Therefore:

For a DFS to effectively and efficiently support true heterogeneity (whether being hardware, software or user), requires a scenario based approach that can encapsulate multiple concurrency control and consistency maintenance mechanisms concurrently. This dissertation proposes, develops and illustrates the ability to apply a component-oriented architecture to concurrency control and consistency maintenance functionality within a DFS, such that the scope and granularity of consistency can be adjusted accordingly.

1.5 GLOMAR Concept

The GLOMAR framework[1] is the component-based approach that supports consistency of data objects within a heterogeneous DFS. GLOMAR provides

[1]The genesis of the name originated from the deep-sea mining ship the Glomar Explorer. It was a joint CIA/Howard Hughes project to build a ship capable of retrieving a Soviet Golf II-class ballistic missile submarine that sank off the Hawaii coast in 1968.

the framework required to encapsulate concurrency control and consistency maintenance functionality within a component-oriented architecture (Szyperski 1997), focusing on the maintenance correctness of file replicas.

GLOMAR makes three major contributions:

- The ability to abstract concurrency control and consistency maintenance functionality into a single component.
- A framework that handles component selection and concurrent implementation.
- A development methodology to facilitate component creation.

A complete design and implementation of GLOMAR was produced, with a live production system running on a number of computing nodes. The system was evaluated to provide a set of quantitative and qualitative results that validates this thesis.

1.6 Outline of the Dissertation

The remainder of the Dissertation is organised as follows. Chapter 2 provides background on the issues of replication, concurrency control and consistency maintenance within a DFS. It also discusses inadequacies of current systems with their support for highly heterogeneous distributed file systems. Chapter 3 discusses the design rationale for GLOMAR. Areas covered include the aims, an in-depth discussion of the motivation and the design evolution. This chapter also introduces major achievements, the concept of the Relationship Component, the middleware layer and the development methodology. Chapters 4 and 5 discuss these major contributions in more detail. Chapter 6 discusses GLOMAR's implementation, including issues and constraints. Chapters 7 and 8 evaluate GLOMAR by detailing the implementation of a number of Relationship Components and determining the impact upon performance and resources the middleware layer imposes. Chapter 9 concludes the thesis, with a summary and possible future work.

Chapter 2

Consistency Models

This chapter encompasses a detailed discussion on the purpose of replication, the need for concurrency control and the num . ous techniques used to maintain consistency (referred to as consistency model/ between replicas. Following this will be a discussion of the issues that affect the design and structure of a consistency model within the context of a DFS. These include issues like transactions, sessions, application requirements, communication schemes and devices. Finally, the implications of each of the issues in relation to consistency model construction will be addressed within the context of mobility. This will form the motivation of this dissertation, focusing on how heterogeneity influences consistency maintenance within current DFS implementations.

2.1 Replication within a Distributed System

The primary motivation for replication (Berstein et al. 1987) within a distributed system arose from the need to improve the availability of data objects. The necessity for this improvement was the inevitability of segments within the distributed system failing. These failures can be in the form of software bugs, human error, overloaded resources and/or media failure. When failures occur, they can result in fail-stops (Schlichting and Scheider 1983), where all operations are required to either wait indefinitely or be aborted as the operations destination is no longer contactable. Such events decrease the reliability of a system, especially when the resources are highly shared. As a result, by using replication, distributed systems were able to mask and manage failures gracefully.

Replication is achieved by implementing redundancy via replicating data objects throughout a distributed system. During periods of failure, when the primary replica is no longer contactable, secondary replicas are used until the failure is resolved. As a result, regardless of the failures that can exist within a distributed system, the repercussions can be masked transparently to the application, improving the availability of data objects.

In addition to improved availability, replication improves performance. This is achieved through the underlying mechanisms having the ability to detect bottlenecks, routing operations to appropriate replicas and allowing many sites to serve data simultaneously.

Within the distributed system domain, replication can be found in distributed systems such as DFS (Levy and Silberschatz 1990; Satyanarayanan 1990; Borghoff and Nast-Kolb 1989), Distributed Database Management Systems (DDBMS) (Ozsu and Valduriez 1991) and Distributed Shared Memory (DSM) (Galli 2000). However, this thesis will focus upon replication within a DFS.

Replication is responsible for both *Replica Placement* and *Consistency Maintenance*. Replica placement is the process of creating, propagating and destroying replicated data within the distributed system. Initially these processes were manual or automated (Howard 1988), explicitly for performance benefits. More recently these processes have been used to improve availability (Kuenning 1994; Satyanarayanan et al. 1990). However, replica placement issues are outside the scope of this thesis and therefore will not be addressed. What will be detailed, is how concurrency control and consistency maintenance are achieved within a DFS.

2.2 Concurrency Control and Consistency Maintenance

The distributed nature required for availability and the centralised storage of file system objects generates a conflict when combined within a DFS. This is because file system activities are built around a centralised storage device, whereas availability requires that data objects be distributed. The result of this combination, is that the data object must be perceived as centralised, but implemented distributed (figure 2.1). This concept is known as *one-copy equivalence* (Berstein et al. 1987) and is the source of most of the complexity within a DFS.

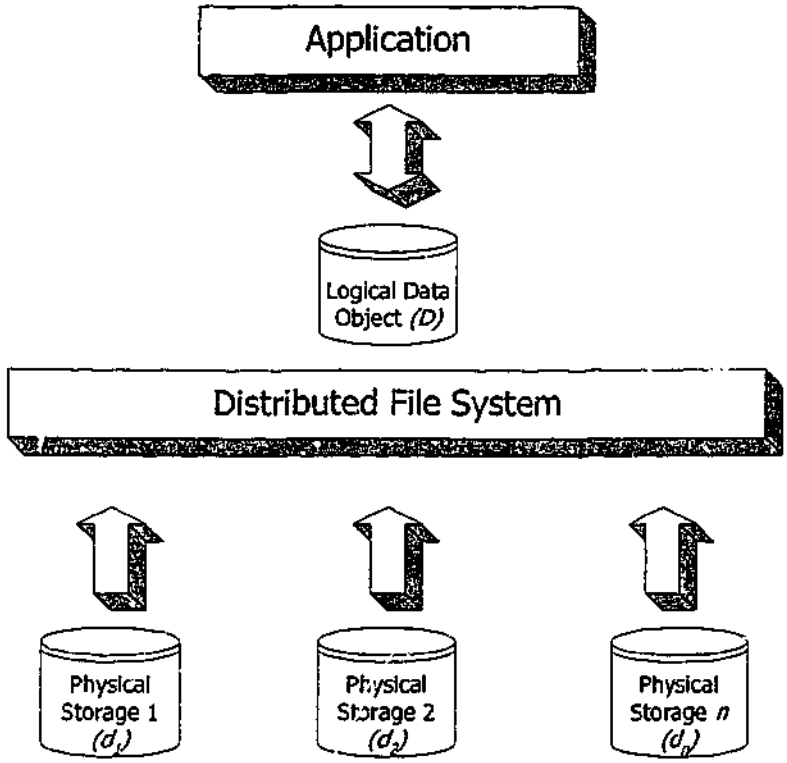

Figure 2.1: One-copy Equivalence

To illustrate one-copy equivalence, consider a data object D[1] which is logically viewed as a single data object. However, physically the data object consists of a number of distributed objects d_n (thus $D = \{d_1, d_2, \ldots, d_n\}$) whose actual implementation is hidden from the application's view. If an application performs a *write* operation, referred to as w (thus a transaction would be $T_i = \{w[D]\}$), the resulting physical transaction to preserve one-copy equivalence would be $T_i = \{w[d_1], w[d_2], \ldots, w[d_n]\}$.

Concurrent operations compound the cost of achieving one-copy equivalence. The reason is that operations perform modification, insertion or deletion operations (so called *update* operations) on physically separated replicas simultaneously. As one-copy equivalence is required so that all replicas remain correct, a means of maintaining an acceptable view of the data is necessary. *Concurrency Control and Consistency Maintenance* aim to manage these concurrent operations, such that conflicts between replicas are avoided or masked from the application, with correctness maintained.

[1]The formalisms used are based on Bernstein and Goodman (1986) description of serialisation theory

Without some form of concurrency control and consistency maintenance, the DFS is not able to handle or resolve concurrent operations that place replicas in conflict with each other. The types of conflicts that can occur fall into two categories; operations that affect the correctness of a data object (serialisation) and operations that compromise the consistency of replicas (one-copy equivalence).

2.2.1 Serialisation Conflicts

Primarily, serialisation conflicts result out of a failure to serialise operations, such that correctness of a data object is maintained. For example, two concurrent applications modify the contents of the data object, regardless of each other's own consistency needs. This form of conflict does not solely arise out of replicating data objects throughout a distributed system. Rather, it is because of concurrent operations being performed unchecked and unmanaged on a data object. In most cases, serialisation of concurrent operations is managed by the local file service, as part of maintaining the integrity of the data objects. However, this does not mean that these types of conflicts should be ignored by the DFS. Instead, the distributed nature just complicates the resolution of these conflicts when they do occur. The two basic conflicts that can effect serialisation include:

- *Dirty read*
- *Over writing of uncommitted data*

The *dirty read* conflict (Coulouris, Dollimore, and Kindberg 2001) arises out of a concurrent operation reading data that is later aborted by the originator of the data. For example, consider two transactions (T_i and T_j) that process a data object x ($T_i = \{r[x], w[y], c\}$ and $T_j = \{w[x], a\}$). A history (H) of the two concurrent transactions illustrated that the value committed to y by T_i was based on data (x) that was aborted.

$$\begin{array}{rcccccc} H & = & r_i[x] & \rightarrow & w_i[y] & \rightarrow & c_i \\ & & \uparrow & & & & \\ & & w_j[x] & \rightarrow & a_j & & \end{array} \tag{2.1}$$

The second form of conflict is over writing of uncommitted data (Coulouris, Dollimore, and Kindberg 2001). In this situation, a concurrent transaction over writes the value of a data object which has yet to be committed (and thus

achieve atomicity) by the originator of that operation. For example, consider two concurrent transactions (T_i and T_j) that consist of a number of *read* and *write* operations ($T_i = \{r[x], w[z], c\}$ and $T_j = \{w[z], r[y], c\}$). A history (H) indicated that before T_j has committed the changes to z, T_i has modified the same data object. Thus when T_j commits, the value of z will be different to the actual value.

$$\begin{array}{rcl} H = & r_i[x] \rightarrow w_i[z] \rightarrow c_i \\ & w_j[z] \rightarrow r_j[y] \quad c_j \end{array} \tag{2.2}$$

2.2.2 One-copy Equivalence Conflicts

One-copy equivalence conflicts result as the distributed system is unable to exhibit one-copy equivalence of a replicated data object. This type of conflict exists because of concurrent operations being performed on different physical replicas indiscriminately. The difficulty with one-copy equivalence conflicts is that in most cases the operations (transactions) have adhered to all the local serialisation constraints. However, the global implications have not been considered, resulting in the failure to maintain one-copy equivalence. The basic types of conflicts that can arise include:

- *Write-write conflicts*
- *Read-write conflicts*

A *write-write* conflict (Coulouris, Dollimore, and Kindberg 2001) results when a number of concurrent operations write to separate replicas, with no concern for consistency amongst all replicas. What makes this conflict different to a write of uncommitted data is that serialisation might have been achieved for each of the operations. However, since they never intersect, the implications of their actions are not known. For example, a data object D is replicated over two physical sites, d_1 and d_2 ($D = \{d_1, d_2\}$) and two transactions (T_i and T_j) modify each replica concurrently ($T_i = \{w[d_1], c\}$ and $T_j = \{w[d_2], c\}$). The history (H) of the concurrent transactions illustrated that the replicas are no longer one-copy equivalent ($d_1 \neq d_2$).

$$\begin{array}{rcl} H = & w_i[d_1] \rightarrow c_i \\ & w_j[d_2] \rightarrow c_j \end{array} \tag{2.3}$$

Read-write conflicts (Coulouris, Dollimore, and Kindberg 2001) arise where data read by one operation is invalidated by another operation on another replica. As a result, operations are unaware that out of date data is being used as a basis for their operations. For example, two transactions T_i and T_j perform concurrent operations on data object D (object D is replicated over two physical sites, $D = \{d_1, d_2\}$). Transaction T_i consists of reading from d_1, writing the results to non replicated data objects (x and y) and then committing the changes ($T_j = \{r[d_1], w[x], w[y], c\}$). Transaction T_j concurrently performs a *read* of x, then the results are written to d_1 and committed ($T_j = \{r[x], w[d_1], c\}$). As the history ($H$) shows, the values T_i committed to x and y were based on a value that were modified by T_j.

$$\begin{array}{rcl} H = & r_i[d_1] \rightarrow w_i[x] \rightarrow w_i[y] \rightarrow c_i \\ & r_j[x] \rightarrow w_j[d_2] \rightarrow c_j \end{array} \tag{2.4}$$

2.2.3 Pessimistic and Optimistic Approaches

There are many approaches to concurrency control and consistency maintenance within a distributed system, so that replica conflicts are avoided, one-copy equivalence is maintained and serialisation of data is achieved. These can be categorised into two extremes, *Pessimistic* and *Optimistic* approaches (Saito and Shapiro 2002). This taxonomy is based on the trade-off between the availability of replicas and the level of consistency.

In certain situations, there is a need for data to be correct and for integrity to be timely preserved. For this to be achieved, a pessimistic approach would be more suitable. The nature of a pessimistic approach ensures that the validation phase occurs early in the execution's life cycle. In other words, the validation of the data correctness is performed prior to any operations being persisted. Figure 2.2 illustrates the phases of execution for an operation within a pessimistic approach.

The benefit of such an approach is the reduction in the probability of conflicts arising. However, as a result this reduces the availability of the replicas. For example, if the validation process is not allowed to complete, due to a network partition (certain replicas were non-contactable), then the operation would not be allowed to complete or was blocked until the connection was restored. Such an approach is suitable for application domains that require a high-level of consistency, but not suitable for systems that require availability of replicas.

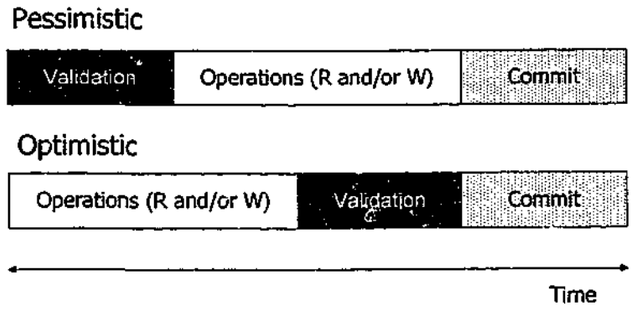

Figure 2.2: Optimistic and Pessimistic Approaches

Optimistic approaches (Davidson 1982; Kung and Robinson 1981) improve availability in favour of correctness. This is achieved by reordering the phases of execution such that validation takes place after the operation has been committed locally. For example, concurrency control and consistency maintenance mechanisms do not need to contact all other replicas to commit an operation. Rather, the validation process occurs after the operation has been persisted. Figure 2.2 illustrates the phases of execution for an operation within an optimistic approach.

The benefit of improved availability of replicas means that operations can be executed concurrently on other replicas. This is especially useful for systems that exhibit frequent partitioning. However, the negative aspect of postponing the validation process until after execution has completed, is the cost of maintaining one-copy equivalence. In most cases the conflict resolution mechanism is complex (Kumar 1994) because conflicts are compounded. The longer the replicas exist partitioned, the increase in the complexity of the actual conflict.

2.3 Consistency Model Taxonomy

The implementation of how concurrency control and consistency maintenance is achieved within a DFS (or any distributed system) is described as a *consistency model* (Galli 2000). The consistency model details how concurrency control and consistency maintenance handle certain aspects of the DFS, such that a suitable level of consistency and availability is attained.

The following sections present a taxonomy of consistency models, based on how operations are managed (*read* and *write* operations) and the degree of trade-off

that is achieved between availability and consistency. The categories include *Polling*, *Token*, *Voting* and *Available Copies* methods (Honeyman and Huston 1995; Helal et al. 1996). However, not all variations are covered here, for example, approaches not detailed include; Differential File (Severance and Lohman 1976), PCCM2 (Cuce and Zaslavsky 1998b; Cuce and Zaslavsky 1998a; Cuce 1999), Tree Quorums (Agrawal and Abbadi 1990), Virtual Partitions (Abbadi et al. 1985), Virtual Primary Copy (Faiz 1995), etc.). For more details see Ceri et al. (1991) and Helal et al. (1996), which give an excellent overview of the major consistency models.

2.3.1 Polling based Consistency Model

Read Once Write All (ROWA) (Bernstein and Goodman 1984) is a simple approach that attempts to provide one-copy equivalence of the results of operations. With this approach, an application can read from any replica. However, an update operation is required to be propagated to all replicas before it is allowed to be committed. In other words, a *logical read* can read any physical data object, whereas a *logical write* must perform writes on all replicas. If a *write* operation fails to be committed on any replica, the operation is aborted or blocked indefinitely until the replica becomes available (depending on the variation of the implementation). By enforcing this constraint, the distributed system can ensure that all the replicas have the most up-to-date view of the data.

The benefit of *ROWA* is that the cost to read a data value requires only contacting a single replica, rather than the entire collection. Much of this is based on the concept that *read* operations out number *write* operations (Burrows 1988; Roselli et al. 2000; Kistler 1993). However, the benefit of maintaining consistency comes at a cost of replica availability, as a single site failure can affect the distributed system's ability to support update operations on all replicas.

The *Primary Copy* (Alsberg and Day 1976; Stonebraker and Neuholf 1979; Oki and Liskov 1988) approach differs to that of ROWA by forcing operations to use a centralised replica (*primary copy*), with the remainder of the replicas used as *backups*. For example let D represent a data object replicated on a number of sites ($D = \{d_1, d_2, \ldots, d_n\}$). One of these sites will be defined as the primary copy p ($p \in D$) and let the remainder be backups b, ensuring that:

$$b \subseteq D \tag{2.5}$$

$$|\, b \,| = |\, D \,| - 1 \tag{2.6}$$

$$p \neq b \tag{2.7}$$

To perform a *read*, the operation must be performed on the replica designated as the primary copy ($r[D] \rightarrow r[p]$). Where as a *write* operation has to write to the primary copy and all backups ($w[D] \rightarrow w[p]$ and $w[b]$) before the operation is complete.

The motivation for the *Primary Copy* approach is to improve the handling of network partitioning, compared to that of *ROWA*. As a result, replica availability is improved by allowing a backup to be upgraded to a primary copy. For example, consider the scenario of a network made up of five nodes, with one of those nodes deemed as the primary copy. Due to a communication failure, two of the five nodes are disconnected from the primary copy. Rather than block or abort an operation, one of the backups in the partitioned segment is upgraded to a primary copy, by either a line of succession or a voting approach. Thus, there exist two primary copies, in two different segments of the network. As a result, update operations are allowed to continue even when partitioning within the system has occurred.

An example DFS implementation of a *Primary Copy* approach is the Harp (Liskov et al. 1991) system. Harp implements a *write back* strategy for update propagation, utilising a modified primary copy method. For a client to process an operation, the request must go via the replica designated as the primary copy. However, what makes Harps implementation of primary copy different is that the primary copy (rather than the client) manages the modification operations and the subsequent calls to backups. For example, a *write* operation generated by a client is performed on the primary copy (operations are stored in a log in volatile memory until committed). Prior to the *commit* request being returned to the client, each backup will acknowledge the request (but not actually committing changes to persistent storage). Requests are then sent back to the primary copy indicating that the backup has received the operation. The primary copy then returns a *commit* message to the client adjusting the primary copy's value to match. A background process is then started to inform each backup that they are now able to commit their logs.

A different approach to how consistency is maintained is *Quasi-Caching* (Alonso et al. 1988; Alonso et al. 1990). With existing systems, validation events take place when either a *read* or *write* operation occurs, regardless of the data being used and its consistency requirements. Within the Quasi-Caching approach, a coherency condition determines when the validation between the central site and remote sites occurs. This coherency condition is a user specific criterion (usually based on time, version or value) that defines an allowable deviation between physical copies (quasi copies). For example, a data object can be modified on a remote site without being propagated to the central site, as long as it does not invalidate the coherency condition. Such an approach takes advantage of application semantics, reducing the cost of an update operation. However, this is offset by the additional overheads required to determine when validation is necessary.

2.3.2 Token based Consistency Model

With the *True Copy Token* (Minoura and Wiederhold 1982) approach, consistency is maintained by assigning a matrix of relationships (lock types) that can coexist (figure 2.3). For any data item that exists, an exclusive lock or many shared locks can be held. An exclusive lock ensures that whoever holds that lock has exclusive control of the data item. Thus, no other locks are allowed. Shared locks indicate that the holder of the lock has shared control of the data item. Therefore, more than one shared lock may exist on a single data item. With the *True Copy Token* approach only when an exclusive lock is held can a *write* operation be performed on a data item. Whereas a *read* operation is allowed to take place if a shared or exclusive lock is held.

	Read Lock (shared)	Write Lock (exclusive)
Read Lock (shared)	Yes	No
Write Lock (exclusive)	No	No

Figure 2.3: Matrix of Relationships

The benefit of a *True Copy Token* approach is its ability to maintain a high-level of consistency in a volatile environment. This is achieved by controlling

the number and type of locks that can exist. For example, by restricting the number of exclusive locks that exist, partitioning will not result in concurrent update operations. If a segment does not host an exclusive token, then no *write* operations are allowed to proceed. However, there are situations where locks need to be generated for the sake of replica availability. In this case, token regeneration protocols are used. For example, creating an exclusive lock on a data item requires that all shared locks be invalidated. However, the process and rules regarding this regeneration need to be specific to the consistency and availability requirements of the distributed systems.

The Echo (Mann et al. 1994) implementation of a DFS is an example of the use of *True Copy Tokens* to ensure replica coherence. As one of the design goals was to achieve one-copy equivalence, a strict (pessimistic) consistency maintenance protocol was enforced. The Echo topology consists of stateful servers, that control and manage replica coherence (via a delayed *write back* approach) and clients (clerks), caching local data in memory. For a clerk, performing an operation on behalf of an application requires the acquisition of a specific token. Primarily, two basic tokens exist; a *read token* and a *write token*. These follow the same rules defined within the *True Copy Token* approach. For example, when a *read token* is allocated, the clerk is then able to only read the contents of the cache. When a *write token* is held, this gives permission for the clerk to modify the cached version of the data. However, for the actual UNIX implementation, this simplistic approach was extended for security reasons. Rather, than a single token used for a *read* operation, it is divided into sub operations (eg *OpenToken*, *SearchToken* and *ReadToken*). Thus an extension to the relationship matrix was defined (refer to Mann et al. (1994) for more details).

One of the interesting aspects of Echo is the process of allocating tokens. Primarily, Echo is a stateful system, with the server monitoring the number of tokens that exist and the owners of the tokens. Thus, for a clerk to gain a token, a remote procedure call to the server is required to fulfil the request. However, there maybe concurrent operations that place the system into deadlock. For example, two clerks hold *write* locks on two separate data files. For them to proceed, they require additional tokens on each other's data. Thus, a deadlock arises. To avoid this, Echo structures the allocation of locks, based on the operation's *operands*. Another implementation detail is that tokens cover whole files, rather than blocks. Leases (Gray and Cheriton 1989) (an agreed time out period) are then used to determine when the server should reclaim a token from a clerk.

Another example of a DFS implementation of *True Copy Token* is Tait's Mobile File System (Tait 1993; Tait and Duchamp 1992; Tait and Duchamp 1991). What makes Tait's approach different to other DFS implementations is that *read* operations are divided into *strict-reads* and *loose-reads* (referred to as the Dual-Read-Call Interface). *Strict-reads* enforce a high-level of correctness by contacting all replicas and determining the most up-to-date data value. On the other hand *loose-reads* return the most convenient value.

Within Taits system, tokens are used to illustrate to a client that the data they are reading is the most up-to-date. This is done by defining the source of frequent updates and *strict-reads*. Clients are defined as *Potential Consistent Writers* (PCW) and receive a *Currency Token* (CT) illustrating that no other client is performing *strict-reads* on the same data. Only when a PCW has a CT can *strict-reads* be performed. Otherwise, a series of complex tasks are initiated to gain the CT for the PCW. Unlike other file systems that implement *write* tokens, Tait focuses on using tokens to define the level of correctness of *read* operations.

There are many other examples of DFS that use a *True Copy Token* approach. These include AFS (Howard 1988) (referred to as *callbacks*), Decent (Marzullo and Schmuck 1988), Sprite (Ousterhout et al. 1988; Nelson et al. 1988) and MFS (Burrows 1988). However, much of the differences are not related to the token implementation, but rather to implementation of other aspects of the DFS.

2.3.3 Voting based Consistency Model

Voting approaches for concurrency control and consistency maintenance are achieved by performing operations only if permission is granted from a set of replicas (the quorum). Thus *read* or *write* operations can continue only when a consensus is achieved within the quorum. Voting approaches improve the support for *write* operations and handle failures gracefully. This is different to polling approaches that favour *read* operations and manage communication failures poorly.

The basic architecture of a voting approach revolves around replicas being members of *quorum sets*. A quorum set is a collection of replicas that must all be satisfied prior to allowing an operation to commit. In most cases two sets exist, the *read quorum* (RQ) and the *write quorum* (WQ). As the name suggests, for each specific operation to be successful, these quorums (specific replicas) must be accessible.

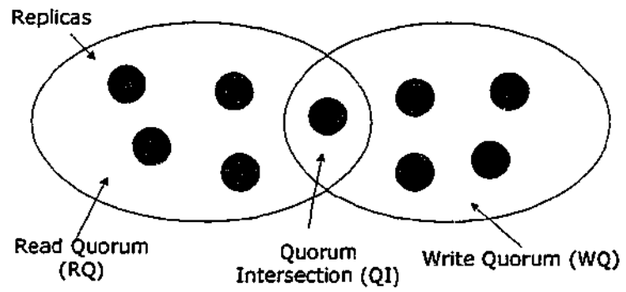

Figure 2.4: Quorum Intersection

Consistency is maintained by ensuring that the quorums (*read* and *write*) intersect, referred to as a *quorum intersection* (*QI*) (figure 2.4, or $RQ \cap WQ = QI$) and that a *write quorum* must have a common member, so that *write* operations are not executed concurrently.

One of the simplest implementations of voting is the *Uniform Majority Quorum Consensus* (Thomas 1979; Kumar and Segev 1988) approach. In this approach, a majority of the sites must be contacted for an operation to be committed. Thus only when half or greater approve of the operation, can the operation be completed. However, not all members need to modify their local copy. They are just required to vote. For example, only one copy needs to be contacted to read data, whereas half or greater of the replicas need to be accessed to perform a *write* operation. The benefit of such an approach is simplicity. However, the cost associated with contacting the quorum for every operation becomes high. For this reason a weighted majority approach was developed.

Weighted Majority Quorum Consensus (Gifford 1979) allows each replica to have more than a single vote. A non-negative integer is assigned to each replica (d_v where $v = 1, 2, \ldots, n$) to define its weight. Then a threshold is defined for both *read* rq and *write* wq. Only when the sum of the total weight ($t = \sum_{v=1}^{n} d_v$) is greater than the threshold, is quorum achieved and the operation is allowed to complete. The validation process is satisfied only when the following constraints are met:

$$rq + wq > t \tag{2.8}$$

$$wq > t/2 \tag{2.9}$$

The benefit of a *Weighted Majority Quorum Consensus* approach is the fact that weight and threshold can be adjusted in such away that consistency or replica availability are favoured. For example, by adjusting the weight of the *write quorum* to equal the same of all weights ($wq = t$) and the read quorum only requiring one replica ($rq < 1$), then a model similar to *ROWA* (section 2.3.1) is achieved. If however the *write quorum* is adjusted to only require a single copy to be accessed to achieve quorum consensus ($wq < 1$), then an approach that is highly available becomes feasible. However, as the process of allocating weights is difficult, there are approaches that automate this task. One such method is *Annealing* (Kumar 1991), in which a process adjusts the weight, so that an optimal threshold is found.

An example of a system that dynamically adjusts weight and threshold is the *Missing Write* approach (also referred to as the Hybrid ROWA/QC) (Eager and Sevcik 1983). Primarily, the system exists as either a *ROWA* or a *Uniform Majority Quorum Consensus* approach, depending upon the status of the system. When no failures are detected, the system exists in *normal mode*. Within this mode, a *ROWA* approach is implemented. When it has been detected that an update operation is non-serialisable (unable to write-to-all), then the system enters a *failure mode*. Within this mode a *Uniform Majority Quorum Consensus* approach is used, as this ensures that members of the quorum will at least see the update operation.

2.3.4 Available Copies based Consistency Model

ROWA-Available (Goodman et al. 1983; Bernstein and Goodman 1984; Berstein et al. 1987) is an altered version of the *ROWA* that improves the availability of update operations such that they are not blocked if some of the replicas are not contactable. *Read* operations are managed in the same way as *ROWA*, with any available replica able to handle the request. The difference with *ROWA-A* is that *write* operations only have to be committed at all available copies, rather than every replica. In other words, update operations can still be performed with N-1 site failures.

Reducing the need for all replicas to see the committed changes, improves the availability of *write* operations to be performed in the face of site failures. Rather than being a full pessimistic approach, *ROWA-A* offers a relaxed pessimistic approach, which reduces the serialisability of operations. For example, some replicas might contain stale data as they are partitioned from the *write* operation. Inconsistencies are avoided by allowing stale data replicas to be

deemed available only after they have synchronised with an up-to-date replica. Only then a replica is deemed as available and able to accept *read* and *write* operations. Unlike other non-blocking approaches, this approach does not allow operations to be performed on stale data.

As Soon As Possible (ASAP) (Berstein et al. 1987; Bernstein and Goodman 1984) is a variation of the *Primary Copy* approach (section 2.3.1)), that relaxes the constraints placed against *write* operations. In the *Primary Copy* approach, *write* operations must be committed on all backups before committed on the primary copy. However, this approach does reduce the replica availability for *write* operations when faced with network partitioning. For this reason *ASAP* was created to allow *write* operations to be committed to the primary copy without all backups having to be available. Update operations are sent to the primary copy, where they are processed and committed. Another process informs all backups that the primary copy has changed. Accompanying this message is the actual results of the update operation.

The benefit of such an approach is that the replica availability for *write* operations is improved as only the primary copy needs to be available for an operation to be committed. However, this lack of consistency between the primary copy and the backups may result in the usage of stale data. If for example a communication failure occurs prior to an update operation being propagated to a backup, when the backup within the partitioned segment upgrades to a primary copy, the data it uses will be stale. However, the motivation for this approach is based on the idea that there are low *write-write* conflicts when data sharing, and the need for replica availability outweighs the possibility of inconsistencies arising.

The above mentioned approaches (*ROWA-Available* and *ASAP*) might improve the availability of replicas for certain operations. However, the order in which validation occurs defines them as pessimistic approaches. There are a series of approaches that are optimistic, as they perform the validation phase after the operation has been persisted. These are referred to as *Available Copies Algorithms* (Bernstein and Goodman 1984; Saito and Shapiro 2002). The primary difference between these and other approaches is the burden placed upon the validation to ensure correctness. For example, operations (whether *read* or *write*) are performed on any available replica, regardless of the availability or consistency requirements. After the operations have been committed to persistent storage, an additional process is used to synchronise the replicas when appropriate.

The Little Work (Honeyman and Huston 1995; Huston and Honeyman 1993) project was an attempt to improve the support for nomadic computing within an operating system and demonstrate an *Available Copies Algorithms* approach. The final implementation was an extension to AFS (Morris et al. 1986) that relaxed the consistency to a more optimistic approach when partially connected or disconnected. By modifying the V-node interface (Rosenthal 1990) of an AFS client and implementing whole file caching (in this case extending the block size to 1 Mbyte), they were able to achieve support for disconnected operations within an AFS network. This was achieved by allowing a client to exist in four different states depending on the state of the connection. These are:

- *Connected*
- *Disconnected*
- *Fetch-Only*
- *Partially Connected*

When connected, the client acts as a normal AFS client, using *callbacks* to invalidate stale data. When a connection with a server is lost, the client changes to a disconnected mode, where operations can still be performed, writing to a cached version of the data. Each operation would be recorded within a replay log to be used upon reintegration. To assist in the reintegration of data with the server (bidirectional propagation), the client enters a fetch-only mode. This mode allows for synchronisation of data and resolution of conflicts (only a simplified approach is implemented). Finally, if a connection with the server exists, however, the limited bandwidth does not allow for normal AFS callbacks, the client enters a partially connected mode. In this mode, operations are allowed to be performed. However, to achieve some level of cache coherence, priority based queuing of operations is used.

Coda (Satyanarayanan et al. 1990; Kistler and Satyanarayanan 1991; Satyanarayanan 1989; Kistler 1993) is a classic example of a DFS implementation of an *Available Copy Algorithms* approach. The aim of this DFS is to support availability of replicas efficiently and thus allow for disconnected operations (Kistler and Satyanarayanan 1991). This is achieved by exploiting persistent storage, non-blocking operations and a complex conflict resolution implementation (Kumar and Satyanarayanan 1995), such that failures and elective disconnections are masked effectively. By doing this, Coda can perform file operations in two states, *connected* and *disconnected*.

The Coda topology consists of a number of replicas mapped to a Coda namespace, called *volumes*. These volumes are perceived as one-copy equivalent, but are physically distributed. A set of replica sites that house a volume is referred to as a *volume storage group* (VSG). A subset of the VSG that is available is referred to as an *Accessible VSG* (AVSG). Consistency is maintained between replicas via the *server replication* mechanism (Satyanarayanan et al. 1990). This mechanism implements a *callback* approach, such that when an *open* operation is requested by an application, the servers notify all other replicas that their version of the replica is no longer valid. Only after the operation has completed does an additional process propagate changes to all members of the AVSG.

When a client is disconnected from the network and unable to rely upon the AVSG, the contents of a local cache are used. Unlike the members of the VSG that implement first class replication, the cached version is a second-class replica, only containing the data specific to the client. The process used to govern cached data and manage client file operations is called the *Venus*. The Venus has three states:

- *Hoarding*
- *Emulation*
- *Reintegration*

When connected, the Venus process is in a hoarding state. Within this state, the contents of the cache are fetched from the server. When disconnected, the Venus moves into the emulation state. This state emulates the existence of a network connection (pseudo-server) using the cache as the data source for operations to perform upon. By doing this, data becomes available as operations are allowed to continue regardless of the connection (this assumes that the data required is present locally). Update operations are catered for in this state by recording the events in a replay log. This log is used during the reintegration state so that operations can be executed on the AVSG upon reconnection.

Coda handles conflicts by initiating an additional mechanism that detects and resolves them. Application-specific resolvers (Kumar and Satyanarayanan 1995; Kumar 1994) are used by Coda to resolve specific conflicts defined by users. For example, conflicts within a Concurrent Versioning System (CVS) are resolved by recompiling source code such that object files are valid. Once conflicts have been resolved (automatically and/or manually), then the Venus returns to the hoarding state allowing normal operations to continue.

Another major DFS implementation of an *Available Copies Algorithm* approach is Ficus (Page et al. 1998; Guy et al. 1990) (influenced by Locus (Popek and Walker 1985)). Unlike other approaches, Ficus supports replica availability within a peer-to-peer model. As with all available copies approaches, serialisation is not guaranteed. Ficus rather ensures that modifications are not lost due to concurrent operations. When a replica is modified, it is not immediately propagated to other replicas. Rather, replicas periodically consult other available replicas to determine if propagation is necessary. When a difference between replicas is detected, Ficus enters a *Reconciliation* phase. This phase determines what is required to bring all replicas to a correct state. This is done by consulting the version vector (Parker et al. 1983; Ratner et al. 1997) of each replica. Using this information, the reconciliation process can determine if one replica "dominates" another replica. If so, then the dominating replica is used as the primary data source for propagation. If however Ficus is unable to determine the dominate replica then manual intervention is required (there are also automated tools to resolve certain conflicts (Reiher et al. 1994)).

Rumor (Guy et al. 1998) is a user-level implementation of Ficus. Like Ficus, it shares the same design philosophy (one-copy equivalence and no lost update guarantees). However it has two additional aspects: *selective replication* and *gossip* (improved replica related metadata sharing). Selective replication (Ratner 1995) is the ability to adjust the consistency granularity within the replicated volume. In other words, Rumor has the ability to selectively adjust what is replicated to each physical site. Gossiping is the ability to share update information regardless of who actually owns a replica. For example, two rarely connected machines can learn replica status via a mutually accessible third machine. An extension to Rumor is Roam (Ratner 1998; Ratner et al. 1996; Ratner et al. 1996). Roam improves the scalability of Rumor, by improving the gossip and selective replication mechanisms.

2.4 Design Issues

Whether designing, modifying or selecting an appropriate consistency model within a DFS, many design issues (beyond that briefly discussed in section 2.3) need to be addressed. This section looks at these, including:

- **Implications of a DFS.** The effect of transactions, sessions and the basic unit as implemented within a DFS.

- **Application, User and Environment requirements.** The effect of file access patterns, scalability and conflict resolution within a DFS.
- **Models.** The different models that exist within DFSs.
- **Environment Hardware.** The effect of communication infrastructure and devices on consistency model development.

All of these issues are pivotal when trying to support consistency maintenance and concurrency control.

2.4.1 Implications of a DFS

The DFS domain imposes specific constraints that are unique to its environment. These constraints can restrict aspects of a consistency model. One such constraint is how transactions and sessions are understood and handled within a file system. Another constraint is the basic unit (blocks, files, directories and volumes) a consistency model uses.

These are of particular importance within the DFS environment as many of the fundamental design features and approaches found within existing consistency models are derived from Distributed Database Management System (DDBMS) environments. Handling these constraints, within a DFS, becomes a major issue as a DDBMS environment perceives them differently.

Transactions

Traditionally, the concept of a transaction is a collection of basic operations denoted by a beginning and an end operation. Each transaction is considered a single unit that can succeed or fail (Ozsu and Valduriez 1991). By defining a structure for basic operations to exist within, properties can also be defined that indicate details about the transaction. For example, the mnemonic "ACID" (Haerder and Reuter 1983), indicates that a transaction must be atomic, preserve data consistency, isolated and durable. However, the implementation of transactions and these properties in a file system domain is more difficult.

Within the DDBMS domain, the concept of a transaction is intertwined with its functionality (for example, queries). However, within existing (common) file system implementations, there is no explicit definition of a transaction. Rather, file systems are more general than a database, thus are not constrained to a detailed abstraction. To achieve a pseudo-transactional model, either file

system primitives have to be substituted to indicate the beginning and the end of a transaction or an extension to a file system is required.

The simplest approach for achieving a pseudo-transactional model is to substitute the *open* and *close* file primitives as the start and end of a transaction. The benefit of such an approach is that applications can transparently create transactions without actually being aware they are doing so. However, the negative aspect of the tight coupling between the boundaries of a transaction and file system operations is that transactions are restricted to a single data object. Also there is no control over whether the transaction succeeds or fails. The main reason for using this approach is to guarantee the correctness of a transaction. If an *open* function is called on a file, then a locking approach could be used (section 2.3.2) ensuring that other transactions do not attempt to intercede prior to the original transaction committing.

The second approach for achieving a pseudo-transactional model is to extend the set of file system primitives so that applications are aware of the file systems transactional capabilities. For example, there are implementations of file systems that exploit a transactional approach (XDFS (Mitchell and Dion 1982)). The benefit of this approach is that fine grained transactional control can be achieved. However, applications have to be explicitly aware of this interface to take advantage of transaction semantics.

Coda's implementation of Isolation Only Transaction mechanism (Lu et al. 1997) is an extension to file system primitives to improve the transactional capabilities of a file system. Rather than operations being ungrouped, an additional mechanism within Coda allows applications to encapsulate operations between a *begin_iot* and *end_iot* operation. As a result, *read-write* conflicts can be detected and flexible conflict resolution strategies can be employed.

Sessions

Sessions are an abstraction of multiple *read* and *write* operations. However, unlike a transaction, atomicity and serialisability are not the primary intention. Rather, prior to a session being invoked, a guarantee from the system is obtained by the application. This guarantee ensures that the view of the data object remains consistent for the operations performed by a single application. An implementation of session guarantees is illustrated in Bayou (Terry et al. 1994; Demers et al. 1994; Edwards et al. 1997). Bayou allows an application to use four different guarantees, including *Read Your Writes*, *Monotonic Reads*,

Writes Follow Reads and *Monotonic Writes*. Each approach ensures different levels of consistency and availability.

The implication of implementing sessions is similar to transactions, as most DFSs have no implicit support for session guarantees. As a result, the solution is similar to how file systems support transactions, that being either transparently implementing session boundaries based on overloading existing operations or non-transparently extending the file system. Overloading existing operations to define the boundaries of a session, restricts the scope of a guarantee and fails to offer the ability to select different session guarantees. Where as, extending file system operations fails to offer transparent support, only supporting applications that are explicitly aware of the file system extension. As a result, defining session guarantees within a DFS is complex, with the resulting implementation solely based on a balance of the benefits against the constraints.

Basic Unit

The primary purpose of the file system is to store file objects in an ordered manner to a persistent storage for future accessing. Since it is only a service, it has no need to be aware of the contents of the file. In other words, to service the needs of the file system, a generic service, based on a coarse grain unit is used. This is different to Database Management System, as the basic stored unit is fine grain and explicitly defined (Ozsu and Valduriez 1991).

The implication is that DDBMS has the ability to maintain consistency of a much finer grain data object, compared to a DFS, which has a file (or encapsulating types, like directories and volumes) available to it. Thus, a consistency model is constricted by a DFS, as it uses a course grain unit as the basic unit for all consistency maintenance events. The ramification of using a course grain unit is that the file must be propagated in its entirety, even when a minor portion of it has been modified. This is because the file system does not persist the specifics of an operation, only the effects of the operation. The result of whole file propagation is inefficient utilisation of resources, in particular bandwidth. This is a major implication when building consistency models within a DFS.

A solution to inefficient utilisation of resources is to impose an additional unit on top of the file system. Traditionally, this has taken the form of file blocks (Dwyer 1998b). In other words, files can be broken into sub elements, which can be treated as units in their own right. For example, blocks can take the form of static and dynamic partitions (Dwyer 1998b), or computed deltas (Burns and Long 1997). The benefit of this approach is that utilisation of resources can be

better allocated. However, the negative aspect is that the consistency model is required to manage, monitor, define and implement the file blocks, independent of the file system. Thus, when considering file blocks as the basic unit, the benefit of efficient utilisation of resources might be offset by the additional cost of implementing the necessary infrastructure.

2.4.2 Application, User and Environmental Requirements

One of the primary requirements that must be considered in the selection or design of a consistency model is the minimal level of consistency an application, user and environment requires. In other words, whether an application, user and environment are dependant upon achieving one-copy equivalence and serialisation, or whether a relaxed form of consistency maintenance is acceptable. This concept can be described as *uniformity* (Saito and Shapiro 2002), meaning the demand that replica content must eventually converge. How they converge and the timeliness of the convergence is partly dictated by the following areas:

- File Access Patterns
- Conflict Resolution
- Scalability

File Access Patterns

When designing or selecting a consistency model, the file access patterns of an application, user and/or environment can be used to configure the implementation to more effectively service consistency and availability requirements. For example, if implementing an approach tailored to a CVS, then the ratio of *write* to *read* operations would prompt the implementation of an approach that favours *write* operations over *read* operations. For this reason, the file access pattern of a DFS, user and/or application are one of the major factors determining the type of consistency model to implement.

Traditionally, defining a level of consistency and availability (within a consistency model) has been based on well-known and general accepted assumptions about file access patterns. For example, numerous and extensive studies into file access patterns (Kistler 1993; Burrows 1988; Roselli et al. 2000; Ousterhout et al. 1985) have consistently confirmed that *read* operations out number *write* operations 5 (or 10) to 1 and that concurrent modifications are rare, equating

to only 1% of all operations. Many current consistency models are built upon these assumptions, as they are assumed adequate in real world systems.

However, there are situations where this generalisation does not adequately exploit the needs of the application, user or environment. For example, the discussion found in Gill, Zhou, and Sandhu (1994) illustrates that files within an academic environments are smaller in size and exhibit less sharing. This is compared to commercial environments where files are larger and exhibit a higher degree of sharing. Thus, the implementation of a consistency model has many of its goals based on the file access patterns of the application, user or environment in which it is attempting to service.

Much of the difficulty associated with using file access patterns, as a major criteria for the final design or selection of a consistency model, stems not from the ratio of *read* to *write* operations, but rather the complexity associated with the frequency of concurrent *write* operations (write-sharing). As the BSD study showed (Ousterhout et al. 1985) (more recently (Baker et al. 1991)), the likelihood of write-sharing operations occurring is minimal. However, this might not be the case in all situations. Rather, the frequency needs to be determined so the criteria of performance, availability and consistency can be satisfied.

Conflicts Resolution

It is inevitable that concurrent operations on replicated data objects will result in conflicts that break one copy-equivalence. With a pessimistic approach, conflicts are avoided up front, as the validation phase is prior to all read or write operations. However, the cost associated with the validation phase can place unnecessary burden on a system, if the nature of the conflicts can be resolved easily.

Within an optimistic approach, as the validation phase is after *read* and/or *write* operations, the likelihood of two replicas no longer being valid is high. However, the benefit is that the cost per operation is reduced, as validation is no longer directly part of the transaction's life cycle.

Determining which approach to choose can be based on the complexity to resolve conflicting replicas. For example, some conflicts can be resolved manually, where as others can be resolved automatically. Manually conflicts can be time consuming. Whereas, automatically resolved conflicts might require less

intervention of the user, however are limited by the implementation of the automatic resolvers. For example, the implementation of the *Application Specific Resolvers* in Ficus (Guy et al. 1990) and Coda (Kistler and Satyanarayanan 1991) demonstrate how simplistic conflicts are managed well. However, these systems become less appealing as the complexity of the conflict increases.

Thus, when determining which consistency model to implement, the possibility of conflicts needs to be evaluated. The extent of the conflicts needs to be measured and the cost associated with resolving the conflict (whether automated or manual) needs to be reviewed. Each of these aspects can have implications on the consistency model chosen.

Scalability

The DFS (or any distributed system) can attribute the beneficial qualities of performance and availability improvements directly to its distribution of data and functionality throughout the network. However, the scale of this distribution can effect consistency model development. As the number of replicas increase, so does the likelihood of concurrent accesses, the increase in cost overheads associated with propagating updates and the increase in conflicts occurring (Saito and Shapiro 2002). For example, implementing a pessimistic approach within a large scale DFS would avoid conflicts, but would create additional overheads and greatly reduce availability. Whereas, implementing an optimistic approach would reduce the overheads associated with propagating updates (as they are moved to a background process) and increase availability, but allow for conflicts between replicas to arise. Thus, choosing a consistency model when scalability is an issue, is highly dependent on what issues are deemed important for the consistency model to service.

2.4.3 Models

An important aspect of any DFS implementation is the model defining the structure and relationship of file replicas. As has been illustrated with some of the existing implementations of DFSs, the relationship between replicas has far reaching implications for consistency models. Currently, the two most dominate models are the client/server model and the peer-to-peer model.

2.4.4 Client/Server

The client/server model is one form of defining relationship between replicas. The basic design consists of a server element, managing resources and a client element, interacting with the server. Commonly the client/server model is referred to a second class replication. The reason is that the clients replica is always second class compared to the replica held by the host designated as the server.

The motivation for this model within the DFS domain stems from two major aspects, the ability for these systems to scale easily (compared to a Peer-to-Peer systems) and to provide an elegant model to manage files in a centralised manner.

Scalability within a client/server model is highly dependant on the specific approach implemented. Within client/server model implementations, there can exist two different approaches; stateful and stateless. A stateful approach requires the server to maintain some form of state relating to the relationship between a client and a server. For example, when a client replica is created, the server keeps a record, using callbacks (Satyanarayanan 1989) to inform the client when the server version of the replica has changed. Such approaches are appropriate when the consistency maintenance techniques are server centric. However, this statefulness of the server does limit the scalability of a file system, as only a finite number of clients can be serviced by a server at any particular time. Rather, with stateless implementations, the server is not concerned for the clients that call upon its services. The server merely provides a service to the clients, regardless of who they are. The benefit is that such systems can scale easily as the server is not required to keep track of the number of client replicas. This lack of tracking means that informing clients of change to a servers replica can not be disseminated, unless some form of broadcast messaging approach is used. Thus, clients must assume responsibility of consistency maintenance of replicas. However, since the server is known to the client, the client has a known location to determine the consistency of a replica.

The other beneficial aspect of the client/server model is that the server offers a centralised point of control for all clients. The obvious benefit of this structure is that many of the management techniques employed to ensure consistency between replicas can be centralised at the server. Thus concurrency issues can be managed in an ordered and serialised manner, as all requests must go via a server. Primarily, for this reason, many existing implementations of DFSs employ a client/server model.

However, the main negative aspect of the client/server model is that a server is the central point of failure. Therefore, if a server is not contactable, then the functionality of the client is restricted. Of course, the level of restriction is dependent on the specifics of the implementation, for example implementing optimistic consistency maintenance on a client. However, this trade-off does come at a cost, as detailed in section 2.2. Solutions have been employed to improve this negative aspect through the implementation of fault tolerance mechanisms. In many implementations, like CODA (Satyanarayanan 1989), multiple servers are deployed, thus reducing the disruption of the client as a result of the primary server failing. However, this does result in a cost, as the mechanism to update servers becomes more complex.

2.4.5 Peer-to-Peer

With the advent of Peer-to-Peer (P2P) (Flenner 2002) file sharing systems (e.g. Gnutella (GNUTELLA. 2002)), Peer-to-Peer has emerged as a viable method of defining replica relationships within DFSs. The basic elements of a peer-to-peer model include three aspects; firstly, all nodes within a network are equal members. Secondly, nodes within the system are autonomous, providing both the facilities to consume resources and expose them (thus simultaneously being a client and a server). Finally, much of the communication is unidirectional.

The implications of a Peer-to-Peer model in a DFS primarily focused on the equality of nodes. As there is no implied structure between replicas; each replica has the same weight. As a result, such systems refer to replica with a peer-to-peer implementation as first class replication. The benefit of not having an implied structure is that replicas can synchronise between peers, rather than with a server. This results in adhoc networks forming, with members able to synchronise replicas regardless of whether they are attached to a server or not. In some respects, this improves the dissemination of replica information within a network, however, this improvement does come at a cost.

This cost is that there is no centralised arbitrator to resolve replica consistency issues. For example, two peers attempt to synchronise their replicas and a conflict results. Since both carry the same weight, as a result, the system has trouble determining which replica is the most up-to-date. Within a client/server implementation, either this conflict would have been avoided or when resolved, the entire network would be made aware of the changes, not just two peers. Such conflict issues lead to the assignment of weights to replicas to define the relationship between peers (e.g. version vectors (Parker, Popek,

Rudisin, Allen Stoughton, Walton, Chow, Edwards, Kiser, and Kline 1983)). However, even with the assignment of weights, ensuring that all changes made by all users on shared data are made visible requires a complex conflict resolution system.

Thus, when designing a DFSs and associated concurrency control and consistency maintenance, a choice between the client/server and peer-to-peer models has a lot to do with which model best describes the interaction between replicas, the issues of scalability, the need for fault tolerance and the complexity associated with resolving conflicts that may arise.

2.4.6 Environmental Hardware

The mechanisms of the consistency model are highly dependent on the environmental hardware of a DFS. In other words, the choice of consistency model must take into consideration the environment in which it exists. Primarily, this means that the constraints of the communication infrastructure and devices that make up a DFS need to be evaluated as part of consistency model development.

Communication Infrastructure

Constraints of the communication infrastructure effect many aspects of consistency model development. These unique constraints are important as they affect specific design characteristics of the consistency model. For example, issues that are affected by the type of communication infrastructure include timeliness of update propagation, update strategies and the actual implementation of the mechanism used to service the consistency needs of the application and/or user.

Communication infrastructure can be categorised into a simple taxonomy; *High Throughput Wired*, *Low Throughput Wired* and *Wireless* connection. This is by no means a definitive classification, but is acceptable for consistency model development.

High Throughput Wired Connection

High throughput wired connection exhibit the characteristics of stable connectivity and low bit error rates. As a result, consistency model implementations built upon these systems can place less emphasis on availability issues, focusing rather upon improving performance time to access a replica and/or maintaining

consistency of the replicas. This is because stability of the wired connection implies lesser need for replication.

In addition these connections provide high throughput (Ethernet at 10/100 Mbps and ATM at 155 Mbps). With access to ample resources (in this case bandwidth), a consistency model has more flexibility in how consistency between replicas is maintained. For example, a bandwidth intensive approach to maintaining consistency can be employed without greatly affecting the available resources. The benefit is that such resource intensive approaches can reduce the time for replica synchronisation, as updates are propagated to all replicas as they occur. This is in contrast to resource-constrained approaches that prioritise propagating updates, such that bandwidth utilisation is improved.

Low Throughput Wired Connection

Low throughput wired connections exhibit some of the behaviour of high throughput wired connections, but with some additional characteristics that are unique to only this connection type. As the name suggests, the main characteristic difference is throughput. As a result, mechanisms implemented within a consistency model and their usage of bandwidth to maintain consistency becomes a critical issue within the DFS. A classic example of a low throughput wired connection would be a connection established using a traditional telephone based modem.

One of the major differences is the characteristic of disconnections. With high throughput wired connections, disconnection events are mainly the result of failures. However, with some low throughput wired connections, disconnection events could also be elective. This means that nodes of the DFS can choose when to connect and disconnect from the network. Not all low throughput wired connections have elective disconnection events as a characteristic. However, within the context of consistency model development, it is assumed that most low throughput wired connections exhibit voluntary disconnection as a characteristic.

The implication for consistency model development when faced with the issue of disconnection events, is whether the event was elective or a result of a failure. As a result, availability becomes an important issue, as support for disconnected operations (the ability to interact with the DFS while disconnected from the network) (Kistler and Satyanarayanan 1991) might be required to ensure the availability requirements of an application and/or user are met.

Wireless Connection

Wireless communication creates an additional set of constraints for a consistency model. One of these is the nature of the connection. Rather, than having a connection that has two states (connected or disconnected), the actual throughput can fluctuate due to environmental factors. This fluctuation is not a rare occurrence, but rather a common characteristic of wireless communication. The implication is that the underlying bandwidth can never be assumed. This results in consistency maintenance approaches that must prioritise events (for example, update notification), as no level of service or future service can be assumed.

Nodes of DFS using this type of connection might disconnect and connect randomly. Rather than being a critical failure or an elective event, disconnection events within a wireless connection infrastructure can also be temporary occurrences. For example, loss of connection can be as a result of being out of range or moving into a "black spot" within a radio network. A consistency model needs to cater for this unpredictable behaviour. This might take the form of a more robust consistency maintenance mechanism that makes no assumptions about the longevity of a connection or the available bandwidth.

Devices

The capabilities and constraints of devices that are members of a DFS can affect the implementation of certain aspects of a consistency model. In the context of a DFS, these constraints primarily focus on the processing and storage capabilities of a device, more than the type of input/output devices and other physical characteristics. A simple taxonomy is defined for devices, including *Desktop and Server*, *Portable* and *Handheld*. Like the communication infrastructure taxonomy (section 2.4.6), this is not a definitive classification. Rather, it is an adequate classification within the context of a consistency model development for a DFS.

Desktop and Server

With the decrease in the cost of processing and storage capabilities, the difference between servers and desktop devices within the context of consistency model development becomes minor. Both devices have access to ample resources and are stable members of the network. The major difference is not in the actual architecture of the device, rather the task performed. For example, it is common for the average desktop device to be equal with small to medium enterprise servers.

When developing consistency models intended for the use on these devices, implementations can exploit ample resources in order to improve performance. For example, access to abundant storage allows these devices to house all elements of a replica (e.g. whole volumes), rather than storing a partial copy (blocks). The ability to house more of the replica locally reduces the need to search the network for a particular file object, in turn improving performance.

The stability of servers and desktop devices reduces the need for software support for availability. This is because the responsibility for reliability rests with the stability of the actual device and not any underlying replication subsystem. Thus, approaches that favour consistency over availability can be employed. This is illustrated in early DFSs (NFS (Sandberg et al. 1985) and AFS (Howard 1988)), as they existed on networks containing abundant resources (this is relative) and implemented pessimistic approaches for concurrency control and consistency maintenance.

Portable

Unlike desktop and server devices, portable devices are built to be mobile (for example laptop computers). The actual architecture of these devices is similar to desktop devices. However, their portability means that their processing and storage capabilities are reduced. From a consistency model point of view, awareness of the available resources is important. For example, replicating whole volumes might not be physically feasible on a portable device. Rather replicating only the files a user requires would be sufficient.

The nature of portable devices creates a number of issues relating to availability. As portable devices are designed to be mobile, the issues of connection and disconnection become important. Section 2.4.6 has already briefly discussed that the ability to connect and elective disconnect creates issues for consistency model implementations.

Handhelds

Handheld devices (Palm (Rhodes and Mckeehan 1999), PocketPC (Boling 1998), Pison (Psion 2002)) might exhibit some of the same characteristics of portable devices (namely mobility). However, they are architecturally different. Primarily, handheld devices are extensions of the desktop, storing user specific data in an easy to access format. What makes these devices different to portable devices is the varying nature of their underlying hardware and their design purpose. Unlike portable devices which replicate desktop functionality, these units

have more specific aims. Their generality is reduced in favour of horizontal application support, with each variation of handheld device varying greatly from one another.

This architectural difference has many implications for consistency models, beyond just the issue of mobility. These implications include; the need for availability, the limited resources exhibited by these devices, the storage mechanism and the integration with the DFS.

- **Availability.** Much of the life of these units is spent disconnected from the desktop (network). As a result, availability is an important aspect, as the unit needs to operate in a disconnected mode for extended periods.
- **Limited Resources.** Unlike other devices, the resources of these units are very limited. For instance, storage, processing capabilities and power supply are reduced. As a result, addressing these concerns is essential. In turn, performance of the consistency model implementation is paramount with these devices, as inefficiency can adversely affect all aspects of the device.
- **Storage Mechanism.** The concept of storage usually conforms to a hierarchical file system model (of course, other implementations do exist). However, due to the varying design goals and underlying hardware, the concept of storage is different within handheld devices. For example, Palm uses a database model to store data object (Rhodes and Mckeehan 1999), whereas Pocket PC has a hierarchical file system, coupled with a universal Database Management System for its storage needs (Boling 1998).
- **Integration.** The connectivity of handheld devices is varied, with either batch (referred to as synchronisation) or on-line processing. However, much of this is based on the supported communication infrastructure. For example, using an IrDA (IRDA 2002) or serial connection facilitates synchronisation, whereas on-line support is achieved with cellular modems and more traditional approaches (Wireless or Ethernet).

Current DFSs may contain various combinations of devices (server, desktop, portable and handheld) and communication infrastructure (wired, wireless, etc). They also service different users and different applications. This landscape creates new issues for consistency model development, as not only the effects of individual elements help determine design choices, but when fully integrated simultaneously (the heterogeneous system) into the single DFS, these issues are compounded.

2.5 Aspects of Heterogeneity

The issues that arise from mobility [2] enabled systems are an excellent real life illustration of heterogeneity within the context of a DFS. The rise of mobility occurred with the expansion of portable devices and wireless communication infrastructures. However, the introduction of mobility enabling devices did not mean that existing infrastructure (wired connection and desktop-server devices) were no longer part of the DFS. Rather, it resulted in a mixture of static and dynamic infrastructure using different communication mechanisms, all sharing the same resources. Thus, mobility and all the associated devices and constraints lessen the homogeneity of a DFS.

Much research has been done into a DFS's ability to effectively handle mobility (Satyanarayanan 1989; Tait 1993; Dwyer and Bharghavan 1997; Guy et al. 1998). However, solutions developed are merely scoped to handle mobility, and not support for heterogeneity. Mobility thus illustrates the need and possibilities of heterogeneity, rather than solves it.

True heterogeneity, with respect to concurrency control and consistency maintenance within a DFS, is creating an approach suitable for all devices, communication infrastructure, users and applications (beyond support for just mobility). Thus, how can true heterogeneity be supported in a DFS?

2.6 Research Area targeted by this Dissertation

When determining a consistency model to implement within a DFS (or any distributed system), the decision process consists of evaluating the application requirements (numerous applications and user needs) and distributed topology (devices and communication infrastructure). Resulting from this evaluation is a number of scenarios, each with a specific need. Scenarios thus define a problem space for which a consistency model will attempt to solve. For example, scenarios can be the need to support a wireless infrastructure or the level of consistency required for a specific application.

Traditionally, serving all these scenarios has resulted in a balance being found between consistency and availability (Saito and Shapiro 2002). The resulting solution is a single consistency model that "as best as possible" handles the constraints of all the scenarios. However, providing a generic approach has

[2] Mobility in this context refers to the effect of mobility enabling hardware, rather than actual movement of a device.

major implications, as certain applications, users and hardware might not be served effectively and efficiently. As a result, the cost of generality is a lack of fine grain support for certain scenarios by a consistency model.

Many current implementations of DFSs illustrate specific scoping of consistency models to support specific scenarios. For example, early DFSs were built upon stable, high throughput wired network infrastructure. As a result, many assumptions were made about the underlying consistency model implementation. These systems supported users and applications well when these assumptions held true, but failed when the assumptions changed. A classic example of this failure was Coda's inability to adapt to a limited bandwidth infrastructure (Mummert et al. 1995). As a result, Coda had to be re-engineered to cater for the new communication infrastructure.

Thus, as the heterogeneity and complexity of a DFS increases, the effectiveness and efficiency of a single consistency model approach is reduced. As a result, a new approach is required to give the DFS the ability to support heterogeneity in relation to concurrency control and consistency maintenance. This research is targeted by this dissertation.

2.7 Summary

This chapter has detailed the fundamental issues surrounding concurrency control and consistency maintenance within the context of a DFS (the consistency model). This chapter has also detailed how consistency models address issues like transactions, sessions, application requirements and hardware elements. Based on the research, the inability of current consistency models to cater for the unique and variable nature of heterogeneous DFS was identified.

The next chapters outline GLOMAR, an approach to provide a workable solution so that an additional level of adaptability can be added to a DFS's implementation for concurrency control and consistency maintenance.

Chapter 3

GLOMAR Design Rationale

GLOMAR provides a component framework for the maintenance of data objects within a heterogeneous DFS. This chapter describes the motivation, aims and high-level design of GLOMAR. The issues that are discussed in this chapter include;

- Why support for heterogeneity is important.
- Why streamlining processes for consistency model development is required.
- And why flexibility of DFSs needs to be improved.

These issues are then addressed in the context of GLOMAR's aims and design. This chapter is the precursor to detailed discussions regarding the system implementation presented in further chapters.

3.1 GLOMAR Motivation

The motivation for GLOMAR (Cuce and Zaslavsky 2002a; Cuce and Zaslavsky 2002b) evolved out of three major issues within distributed system development. The first is the difficulty for existing consistency models to support heterogeneous environments. This is because current DFS implementations make assumptions about the target platform, specifically scoping functionality to that platform only. For example, some implementations assume abundant resources (wired connections), whereas others assume resources are tightly constrained (wireless connections). Thus, by assuming the target platform, the

DFS's ability to adapt to the elements that make-up a heterogeneous environment is reduced.

The second issue relates to the failure of some distributed system developments to adhere and benefit from commonplace software engineering practices. Existing approaches incorporate all facets of DFS functionality into a single system. Consequently, these systems are difficult to configure, complex to extend and static in their ability to adapt. For example, when a DFS implementation packages all the concurrency control and consistency maintenance mechanisms in an ad-hoc manner, extending the existing code base is subsequently more difficult.

The third issue is the lack of flexibility associated with development of consistency models, configurability of these consistency models and the ability to define specific heuristics custom-built for different target platforms. Current DFS implementations do not offer broad flexibility in relation to the tasks they perform. For example, installing a new heuristic (in this case packet loss information) into an existing DFS cannot be easily accommodated. Rather, the existing heuristics found within the DFS are the only ones available to be used. In some cases, the existing heuristics might not service a particular purpose as well has a custom-built heuristic. This lack of flexibility is common in many existing DFS implementations.

3.1.1 Support for Heterogeneous Environments

It is common for heterogeneous environments to consist of servers, desktops, laptops and handheld devices, all running different applications and servicing different users. These might be interconnected via different network infrastructures, including broadband, wireless and infrared (figure 3.1). Applications running within this environment might require a high level of consistency, whereas others might not. The type of user that can exist may vary from transient (connecting and disconnecting at random intervals) to more stationary. All these variations and many others can coexist within the one heterogeneous environment. Thus, the primary motivation for GLOMAR lies with the narrow scoping of existing DFS implementations to cater to the variety of different hardware elements, application domains and user requirements, which exist concurrently within a heterogeneous environment.

The term scenario is used to describe a single variation of hardware, software and user that can exist. Within a heterogeneous environment, many scenarios can coexist concurrently. The problem with existing DFS's is that if a number of scenarios are experienced, only a single scenario will be effectively and

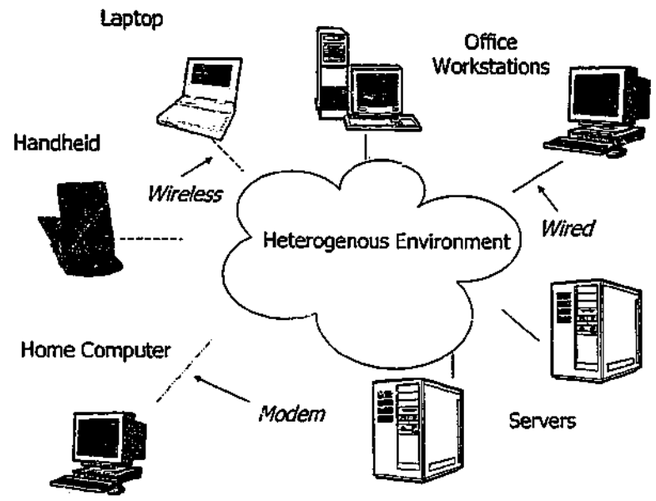

Figure 3.1: A Heterogeneous Environment

efficiently supported. Consider a DFS built upon a network where the connectivity is unstable. Such a system would implement appropriate mechanisms for that scenario only, for example, optimistic concurrency control with periodic updates. These chosen mechanisms are specifically suited to this scenario only.

However, when implementing a mechanism on a system where the underlying assumptions are different to which it was designed, the issue of effective and efficient support becomes apparent. For example, consider the mechanisms tailored for an unstable network implemented on a stable network connection (previous example). The result is not a total failure of the DFS, but rather a failure of the DFS to take advantage of the stable network, as an optimistic concurrency control approach is not the most effective method of maintaining consistency in this scenario. As a result, the features that were a benefit in an unstable network become a burden in a stable network.

Thus, making assumptions about the underlying constraints that a DFS exists within hinders its ability to provide an effective and efficient level of service. This in turn results in a failure to support the needs of a truly heterogeneous system.

3.1.2 Consistency Model Development

If one wishes to create a consistency model to be used in a DFS, he/she will find a number of issues that would make the process difficult. The first and most difficult issue is how the consistency model code is embedded within an application or operating system. In most cases, much of the concurrency control and consistency maintenance functionality is tightly coupled to the file system module, which is entrenched in the operating system. To integrate new consistency models into an existing DFS requires some system level programming, which can be an extremely difficult and time consuming task (in particular when the operating system is proprietary).

In the case of a Distributed Database Management System (DDBMS), the code responsible for the concurrency control and consistency maintenance is in most cases tightly coupled to the DDBMS core. To gain access to this functionality for the purpose of implementing a new approach, requires access to the original source code. Such is the nature of concurrency control and consistency maintenance that even if the source code is available, defining where the consistency model portion of the DDBMS core begins and ends is difficult. In other words, rarely is the process of concurrency control and consistency maintenance abstracted into a single module for easy modification within the DDBMS or DFS.

Why consistency model development fails to adhere to a unified and compartmentalised approach is because one does not exist. Many of the existing DFS's have just focused on "getting it working", rather than defining good specifications and exploiting good software engineering practices. In turn, the code associated with consistency models is hard to create, difficult to port, convolute to modify and unable to be extended easily.

3.1.3 DFS Flexibility

Current DFS's provide limited flexibility to configure, extend and adjust many elements of their implementation easily. For example, what events constitute a disconnection operation cannot be easily changed within an existing DFS.

The benefit of limited flexibility is that the complexity of certain tasks is hidden from the user. However, this restricts a DFS's ability to adapt. An example of this is the difficulty associated with modifying a DFS's consistency model for a new scenario. Rather than promoting a structure that allows any variation of consistency model to exist, many DFSs, by their inflexible nature restrict the

implementation possibilities. In other words, the DFS's inability to adapt is due in part to the inflexibility of its implementation.

3.2 GLOMAR Aims

The aims of GLOMAR are to:

- **Create a framework that supports existing and future heterogeneous environments.** This means for GLOMAR to support a truly heterogeneous environment, it must provide an open specification to allow for new and different scenarios to be expressed, built with few underlying assumptions about the target platform and allow many different consistency maintenance and concurrency control approaches to be supported.
- **Provide a formal specification for the creation of consistency models.** The process of creating consistency models has been ad-hoc, with no formal approach or specification available. GLOMAR attempts to rectify this by defining a development process, where interfaces and well defined formal specifications determine the structure of the consistency model. This will improve the software engineering practices associated with consistency model creation and offers greater reusability across platforms and scenarios.
- **Focus developments on the differing aspects of a distributed file system.** When studying the different DFSs, much of the functionality and practices are shared across implementations. Thus, when developing a DFS, a more effective approach is to reuse existing systems and change only the parts where one wishes different functionality. GLOMAR attempts to provide this by exposing the basic functionality of a DFS via a number of services. As a result, GLOMAR only requires new functionality to be implemented in relation to consistency model creation. Thus, GLOMAR aims to reduce the development burden, since much of the functionality will be already available. This is similar in concept to how the CORBA ORB (Object Management Group 1999) manages communication independent of CORBA objects.
- **Allow highly configurable consistency model implementations.** In keeping with the aim of supporting heterogeneous environments, GLOMAR will incorporate an advanced dynamic configuration mechanism for

consistency models. This provides the flexibility to define when a consistency model is activated and what data objects it governs. Such a mechanism is easily adaptable and flexible enough to add new constraints when they become available.

- **Provide enough flexibility to ensure that any alternative approaches can be supported within GLOMAR.** As Chapter 2 has illustrated, there are many approaches to maintaining consistency and controlling concurrency within a DFS. No one approach is better than others in all aspects. Rather, they each have unique properties that service a particular need. For GLOMAR to exploit alternative approaches, the mechanism that houses them will need to be flexible enough not to restrict their objectives. Thus, GLOMAR will provide a balanced approach, one that promotes formal specification in relation to consistency model creation, but be open and flexible enough to accommodate easily legacy and differing approaches.

3.3 Proposed Architecture of GLOMAR

3.3.1 How Heterogeneity is Supported

To support the multitude of scenarios that can exist within a truly heterogeneous environment, the DFS is required to be adaptive. This adaptation is achieved either as a result of a single mechanism that adapts based on a particular constraint (*the single consistency model approach*) or one that has a multitude of solutions available to it, for which it can select and use the most appropriate (*the multiple consistency model approach*).

The purpose of the single consistency model approach (figure 3.2) is to be adaptive by defining a semantic space that encompasses all the foreseen consistency and concurrency requirements. Such a mechanism would define a semantic space, with each end corresponding to an extremity, in this case, pessimistic and optimistic concurrency control. Depending on a number of triggers (e.g. number of *read* versus *write* operations) the consistency level provided at a particular point of time is defined within this semantic space. Such an approach provides an elegant mechanism for consistency maintenance and concurrency control. However, the type of service provided can only be somewhere within this semantic space. Essentially, a single consistency model approach imposes a

level of generalisation for the sake of elegance and simplicity. This does not provide the fine grain consistency maintenance, concurrency control and flexibility that a truly heterogeneous system requires.

The other approach is the multiple consistency model approach (figure 3.2). Unlike the single consistency model approach, the multiple consistency model approach entrusts responsibility for concurrency control and consistency maintenance to other modules. Such an approach is similar to plug-in architectures (Netscape 1998; UPNP 2002), where tasks are delegated to external modules containing the majority of the functionality. The benefit of this approach is that a number of consistency model modules are potentially available, allowing very specific implementations for unique events. This results in a level of flexibility that is not possible with the single consistency model approach.

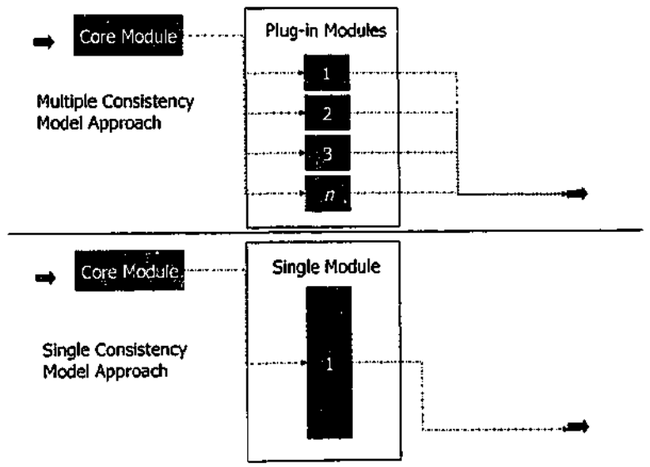

Figure 3.2: Single Consistency Model and Multiple Consistency Model Approaches

One negative aspect of the multiple consistency model approach is that the effectiveness of consistency maintenance and concurrency control is only as good as the available modules. For example, consider a scenario for which no available consistency model module was present. Within the multiple consistency model approach, the event would fail, as the event could not be serviced. Of course a generic module could be implemented to reduce the repercussions of the failure to find a suitable model. But it would experience the same negative aspects of a single consistency model approach as a result. However, within a single consistency model approach, an attempt would be made to manage this event.

Thus, the success of the multiple consistency model approach is solely based on the modules at its disposal.

Another major concern with the multiple consistency model approach is the performance cost associated with implementing this approach. It is inevitable that an additional management layer is required to manage multiple consistency models. As a result, every event is required to pass through a middleware layer before being processed. This adds a cost factor, in addition to the file system.

The approach chosen for GLOMAR is the multiple consistency model approach, as it best fits with the aims of GLOMAR. Some of the motivation behind selecting the multiple consistency model approach for GLOMAR is that it offers the flexibility required to service the needs of a truly heterogeneous environment. For example, with the single consistency model approach, the ability to adapt is based on definable constraints. However, with the multiple consistency model approach, there is the ability for future scenarios to be addressed through the implementation of multiple consistency models.

The negative aspects of the multiple consistency model approach were also considered in this selection. As these negative aspects are inevitable within the multiple consistency model approach, GLOMAR will firstly aim to streamline the creation process of consistency models. Thus, this will improve the likelihood of many consistency model implementations being available. Secondly, is foreseeable that the multiple consistency model approach will impose an additional cost. However, these are estimated as minor, compared to the flexibility that this approach will provide.

3.3.2 Streamline Creation of Consistency Models

GLOMAR intends to streamline the development process associated with the creation of consistency models, by applying a specification and using a component-oriented architecture (Szyperski 1997). This allows GLOMAR to provide the benefits of software reuse, easy deployment and an improved ability to extend existing consistency models. The solution chosen is to implement a component approach, with the consistency model functionality encapsulated behind a custom-built interface.

A component-oriented architecture was chosen as a means of streamlining the encapsulation of consistency model functionality. This choice stems from the ability to provide a unit of abstraction that embodies all elements of consistency maintenance and concurrency control. Primarily, this allows for the flexibility

required to house different and varying consistency models and the use of meta-data to describe the scenario a consistency model is best suited.

A component-oriented architecture also promotes good software engineering practices, including the improved deployment and management of resulting components. This is essential for consistency models to fit within the multiple consistency model approach.

Finally, the component approach offers the framework to define a specification and development methodology for the creation of consistency models. The result is an encapsulation of consistency functionality and context such that they are easily created, easily deployed and easily managed.

3.3.3 Abstracting DFS Complexity

GLOMAR takes the point of view that time and effort should be spent on the creation of the differing aspects of a DFS, rather than the creation of existing techniques and methods. In GLOMAR's case, this is how concurrency control and consistency maintenance are handled. As a result, much of the complexity of the DFS is abstracted into purposely built services. This means a DFS developer need not be concerned about the minor aspects of a DFS, but rather can concentrate on the consistency model portion of the DFS solely, using these services when necessary.

However, beyond just providing services to be consumed by consistency models, the issue of their management needs to be addressed. For example, with a multitude of consistency models, there requires a process to determine which model is more appropriate for a particular scenario. Thus, GLOMAR also needs to provide a service that is responsible for the run-time management of consistency models. The GLOMAR middleware layer does this and is similar to other middleware implementations (CORBA (Object Management Group 1999) and DCOM (Brown and Kindel 1998)) where a set of standard services and management tools are provided. In the case of GLOMAR, the services that will be provided include naming, resource evaluation, operating system and/or application integration and consistency model management. This results in a centralised system that coordinates and manages all concurrency control and consistency maintenance activities, appropriately.

3.3.4 Flexibility

One of GLOMAR's design objectives is to support many variations of consistency models to service a truly heterogeneous environment. Such a task is a difficult one and is the crux of this dissertation. The solution proposed attempts to provide the flexibility necessary to handle as many consistency model variations as possible. This is achieved by focusing on three areas:

- The creation of an open non-restrictive architecture, which is the basis for consistency model implementations.
- Improve the extendibility of middleware layer services, such that new and unique services can be added easily.
- Provide a level of configurability, such that many aspects of the GLOMAR system can be adjusted and modified.

As stated earlier, the use of a component-oriented architecture promotes improved software engineering practices (Szyperski 1997). In addition, the nature of a component is such that any functionality can be bound to it, but still offering a single unit of representation. Thus, the component itself does not restrict the flexibility of the concurrency control and consistency maintenance functionality.

For unique consistency models to be supported within GLOMAR, might require that additional support services be available. For example, a consistency model might require packet loss information to determine the scenario for which it is valid. Many existing DFSs do not provide this packet loss information, let alone a mechanism to insert a purposely built module that derives this information. GLOMAR enhances this aspect of a DFS by providing a mechanism to improve its extendibility, by allowing new services to be added to the middleware layer easily. Such services might include specific modules targeted at a particular consistency model, or more general services that provide a service to all consistency models. This is achieved by exploiting a plug-in architecture, where external functionality can be inserted into GLOMAR. These services can be accessed easily by any or a particular consistency model. The specific implementation of this approach is detailed in Chapter 5.

GLOMAR also promotes a high level of configurability of all its components. For example, GLOMAR has the ability to set specifically what data objects, users and applications a consistency model can govern. Specific details on how

this and other configuration features are achieved are outlined in later chapters (4 and 5).

3.4 GLOMAR Overview

GLOMAR supports multiple concurrency control and consistency maintenance mechanisms under a single system. These mechanisms are defined as a consistency model and are coupled with metadata within a housing component, referred to as a *Relationship Component*. Depending upon the current scenario, GLOMAR determines via a middleware layer, which Relationship Component is best suited to effectively and efficiently service the consistency needs of the application, user, data and the DFS.

The contribution of GLOMAR consists of three elements, the component based methodology in relation to the creation, usage and implementation of concurrency control and consistency maintenance mechanisms, the design and structure of the components that houses the concurrency control and consistency maintenance functionality and the middleware layer used to manage run-time activities.

3.4.1 Relationship Component

The concept of the *Relationship Component* stems from the need to provide a framework that can encapsulate different concurrency control and consistency maintenance mechanisms. However, the structure of the Relationship Component is more than just a component-based wrapper around a consistency model. Additional functionality is required to help describe the context for which the consistency model is suited. Thus, the make up of the Relationship Component is one that contains the functionality to control concurrency and define its context.

These elements are implemented by delegating each specific area (functionality and context) into sub-components within the Relationship Component. These sub components include the *Consistency Model*, which contains a single concurrency control and consistency maintenance mechanism; the *Relationship Scope*, which defines the context for which it is valid; and the *Clone List*, which defines the data objects the Relationship Component will govern (figure 3.3).

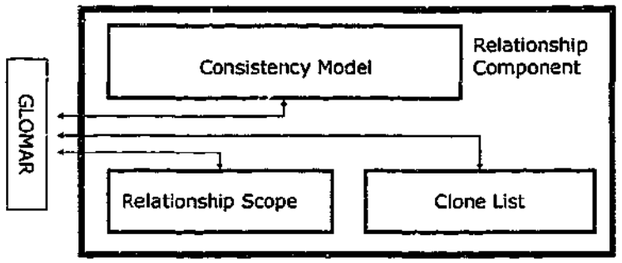

Figure 3.3: Relationship Component

Consistency Model

The crux of managing concurrency control and consistency maintenance is encapsulated in the Consistency Model. The Consistency Model is the packaging of functionality associated with the maintenance of consistency between replicas within a DFS. Much of the complexity and development time in creating the Relationship Component is associated with the Consistency Model. Specific details of the nature, interface and structure of the Consistency Model are provided in Chapter 4.

Relationship Scope

With the multiple consistency model approach, a variable number of consistency models can co-exist. This in turn requires that a process be used to determine which model is the most appropriate at a particular point in time. As a result, determining the most appropriate consistency model relies on detailed information used to describe the current scenario. GLOMAR achieves this by defining the Relationship Scope. The Relationship Scope is a set of rules that define when a Relationship Component is invoked. The legitimacy of the Relationship Component is determined by evaluating its Relationship Scope against the current scenario. For example, if a Relationship Component is to be invoked only when a network partition occurs, then the current status of the network will be the primary factor when determining whether this Relationship Component is invoked or not. Essentially, the Relationship Scope facilitates this by providing a simple interface that can be implemented as needed.

Clone List

The purpose of the Clone List is to define what physical replicated data objects (Clones) this Relationship Component will govern. The Clone List is a configurable mechanism used to define the relationship between one or many data objects to one or many consistency models. As a result, the Clone List is used as part of the decision making process to determine which Relationship Component to invoke and when.

3.4.2 GLOMAR Middleware Layer

Managing and servicing of Relationship Components is handled by the GLOMAR's middleware layer. Within the middleware layer, there are a number of services required to operate GLOMAR smoothly. These include *Local Operation Interface*, *Remote Operation Interface*, *Clone Distribution Manager*, *Service Manager*, *System Grader* and *Relationship Component Repository*. Each of these are used or governed by the *Executive* (figure 3.4).

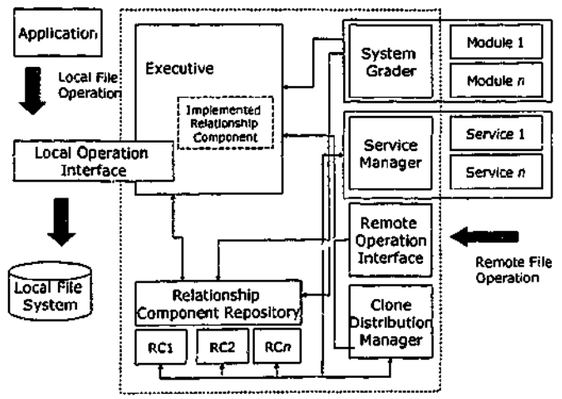

Figure 3.4: GLOMAR Middleware Layer

Local Operation Interface

The purpose of the Local Operation Interface is to capture all local file operations and forward them to GLOMAR for further processing. The Local

Operation Interface has two interfaces, a transparent (for operating system integration) and a non-transparent (for direct application invocation) interface. The Local Operation Interface works by intercepting an operation and redirecting it to GLOMAR. Only when GLOMAR deems the operation a success (the Relationship Component has successfully performed the operation remotely), can the file system allow the changes to be stored on the local stable storage.

Remote Operation Interface

Not all file operations, handled by the Executive, are a result of local file operations. Some operations originate from remote sources. To ensure that these operations are handled appropriately, the Remote Operation Interface is used to manage these operations.

Clone Distribution Manager

The purpose of the Clone Distribution Manager is to provide the Executive with information about a Clone. For example, this includes resolving the ID of a file to its absolute address. The need for such a mechanism is necessary as name resolution is made more difficult with replicas distributed in different parts of the network and in different parts of the file system's hierarchy.

Service Manager

Beyond the default services provided by GLOMAR (in particular the Clone Distribution Manager), additional services may be required to deal with the specific needs of some Relationship Components. For example, a Relationship Component may require a transaction management service. For this reason, GLOMAR provides a Service Manager that is responsible for the implementation of user-specific services.

System Grader

Within the GLOMAR middleware layer, the System Grader is used to derive the information to describe the current scenario. This is used for selecting a Relationship Component or re-evaluating its legitimacy. The unique aspect of the System Grader is its ability to describe any type of scenario. Traditionally similar systems have provided only a standard set of heuristics (e.g. storage size, cpu speed, bandwidth) to describe the current scenario (Noble 2000; Yu and

Vahdat 2000a). However, there are situations where purposely built *Context Providers* are required to derive information that is not intrinsically supplied by GLOMAR. In keeping with the component-oriented architecture, the System Grader acts as a container for third party Context Providers to be plugged in. This provides the flexibility to add new mechanisms to describe a scenario for which a Relationship Component is valid.

Relationship Component Repository

The purpose of the Relationship Component Repository is to manage the runtime selection of Relationship Components. For example, the Executive requires a Relationship Component be selected due to a change in the current scenario. The Relationship Component Repository ultimately compares the new information collected by the System Grader, with the Relationship Scope of each installed Relationship Component to determine which Relationship Component is valid. However, the process is more complex than this (Chapter 5), but essentially its purpose is to provide the tools to manage multiple Relationship Components.

Executive

The purpose of the Executive is to manage all services and events within GLOMAR. For instance, the Executive is responsible for calling the services within the middleware layer to derive such things as the current scenario (via the System Grader) and the valid Relationship Component (via the Relationship Component Repository). It is also responsible for handling file operations that are passed to it by the Local Operation Interface and Remote Operation Interface. Thus, the Executive binds all services and events, such that the aims of GLOMAR are met.

3.4.3 Implementation Issues

Much of the design of GLOMAR has focused on providing a system that is independent of a particular platform or distributed system implementation. Primarily, it was designed to be able to service the needs of multiple distributed system platforms, with much of the ideas independent of a specific implementation. However, as the design advanced, there was this need to define a particular distributed system implementation. The choice made was to implement

GLOMAR as part of a DFS. As a result, some of the system design has been customised to service the needs of a file system.

The choice of implementing GLOMAR as a DFS stemmed not from any specific reason. Rather, a DFS offered more challenges in relation to support of flexibility, concurrency control and consistency maintenance. This is not to say that this approach could not be ported to other distributed systems. However, the vast collection of constraints and possible scenarios make a DFS an appropriate choice to demonstrate the feasibility of GLOMAR's approach.

3.5 Related Work

Much of the motivation of GLOMAR stems from the inadequacies of existing DFS designs and implementations. However, some aspects of GLOMAR's final design are derived from existing work. This section details the genesis of GLOMAR in relation to the existing work currently tasked with the management of consistency maintenance and concurrency control within a DFS.

TACT (Yu and Vahdat 2000a; Yu and Vahdat 2000b) is an implementation of a distributed system that adapts it consistency model at run-time. TACT shares much of the same aims as GLOMAR, with both systems focusing on the concept that different applications can have different consistency needs. TACT implements different levels of consistency by using metric information to determine where within the consistency-availability semantic space a replica should be positioned. TACT differs in that it chooses consistency based on events (single consistency model approach), whereas GLOMAR chooses consistency based on rules (multiple consistency model approach). The TACT design was one of the catalysts for the final approach undertaken within GLOMAR.

The Teapot (Chandra and Larus 1999) approach demonstrates a similar approach to GLOMAR with respect to how consistency models are created. Teapot implements a domain specific language used in the creation of consistency models for parallel and distributed systems. However, the resulting cache coherence protocols produced by Teapot are implemented as a single general-purpose approach. This is different to GLOMAR, which promotes many Relationship Components within a single DFS. In addition, the constraints of the language restrict the development of consistency models that are unique. For this reason, the design choice made by GLOMAR was to support domain specific implementations, rather than a domain specific language.

The ability to select and use multiple consistency models to govern a file has been attempted before. Tait (Tait 1993) implements a *two-read* approach. In this approach the application has the ability to determine which type of *read* would best serve its purpose. This concept has been enhanced again with MFAS (Dwyer 1998a), which extended the basic file primitives to include information regarding the consistency requirements of the operation. Both Tait's system and MFAS require applications to be made aware of the additional APIs to exploit this functionality. Also the types of consistency models available were restricted, as only a number of static implementations were available. For this reason, GLOMAR chooses to implement a transparent approach, allowing for multiple pluggable consistency models.

The concept of a framework that manages consistency models and implements a design methodology can be found in how Pocket PC (Boling 1998) and the Palm operating system (Rhodes and Mckeehan 1999) implement Conduits (Rhodes and Mckeehan 1999; Microsoft 1998). Consistency models are implemented behind a simplified interface, with a manager deciding when to implement the models. However, where they differ to GLOMAR is that Conduits are highly focused to offer one-to-one interaction between desktop and handheld devices. Thus, there is no mechanism to determine the current scenario. In contrast, GLOMAR aims are more general, with fewer assumptions made about the underlying platform. Thus, this means GLOMAR attempts to be more flexible in the target platforms it supports.

3.6 Summary

This chapter describes different issues that affect the efficiency and effectiveness of consistency maintenance (via the control of concurrency) within a distributed system. Of particular interest is how current implementations of consistency maintenance mechanisms are inherently complex to create and inflexible when faced with heterogeneous environments.

Emerging from these constraints is GLOMAR's approach to managing concurrency control and consistency maintenance. The crux of GLOMAR is the combination of multiple consistency models (coupled with metadata) and middleware layer working collectively. This attempts to exploit the current scenario to efficiently and effectively service the consistency needs of the resources, users, data and applications.

The next chapters (4 and 5) focus specifically on the major contributions, including the Relationship Component approach to packaging concurrency control and consistency maintenance functionality and the middleware layer used in the effective and efficient management of these components.

Chapter 4

The Relationship Component

One of the contributions of GLOMAR is the ability to encapsulate alternative consistency models within a general-purpose component called the *Relationship Component*. This chapter discusses the Relationship Component, its purpose, motivation, aims and outlining in detail why certain design choices were made. This chapter also looks at the three sub-components within the Relationship Component, outlining the purpose and structure of each. Finally, this chapter concludes with a discussion on the issues that arise from a component-oriented architecture.

4.1 Relationship Component Design Origin

Existing implementations of concurrency control and consistency maintenance mechanisms are said to be either fine grain (application level (Horton and Adams 1995; Stonebraker and Neuholf 1979)) or coarse grain (operating system level (Kistler and Satyanarayanan 1991; Guy et al. 1998)). Fine grain concurrency control and consistency maintenance mechanisms attempt to provide one-copy equivalence by tightly coupling specific functionality to a single application. As a result, this allows application specific data to be exploited to achieve a very precise mechanism to control concurrency and maintain consistency. However, due to the specific scoping of these systems, they are complex to create, difficult to maintain (and improve) and are not easily ported across application domains.

Coarse grain concurrency control and consistency maintenance mechanisms on the other hand, aim to provide one-copy equivalence to all elements within the scope of a distributed system, rather than a specific application. In other words,

they provide a cross domain generic approach, which chooses not to exploit specific data and structures, preferring to use a coarse grain unit (for example, file objects or operations) on which any concurrency control and consistency maintenance event is based. The negative aspect of coarse grain mechanisms is that concurrency control and consistency maintenance is static across different applications and users. In addition, coarse grain concurrency control and consistency maintenance mechanisms are difficult to extend and inflexible when ported to different platforms and environments.

What is required is an approach where the benefits of both fine and coarse grain mechanisms can be utilised concurrently. In other words, a balance between the two approaches is required, such that the resulting mechanism can be extended easily, be more adaptive to platform and environmental change, but not reduce its functionality capabilities. This balance can be achieved (and is the crux of this dissertation) by decoupling the concurrency control and consistency maintenance mechanism from the operating system and/or application, re-implementing it within a reusable component.

The primary motivation for using a component-oriented architecture to encapsulate concurrency control and consistency maintenance functionality is that it provides:

- **A single unit.** Rather than concurrency control and consistency maintenance functionality embedded within the operating system and/or application, all the functionality is compartmentalised and centralised within a single unit (a component). One of the benefits of using a component is that deployment is simplified, as the costly process of migrating functionality is reduced. Another benefit of using a component is that concurrency control and consistency maintenance functionality can exist side-by-side, concurrently and independent of each other.

- **Improved maintenance and extendibility.** The process of decoupling consistency maintenance and concurrency control functionality from an operating system and/or application decreases the complexity associated with maintaining and extending it. This is because only the component needs to be modified (or extended), not any external modules.

- **Implementation independent of interface.** By using a component to encapsulate concurrency control and consistency maintenance functionality, an interface for that component can be defined. As a result, regardless of the development type, language of choice or motivation, as long as the interface is met, then additional components can be created simply and easily. Why this is important is the component-orientated architecture supports flexibility, as the specifics of the concurrency control and consistency maintenance functionality are independent of the interface that defines the component.

Even though the component-orientated architecture is flexible enough to support different and unique implementations, there still needs to be guidelines on the structure of the components to ensure that the aims of GLOMAR are met. In this case, all components need to service and support the concurrency control and consistency maintenance needs of the DFS. To achieve this, minor restrictions have been placed on the nature, structure and scope of the Relationship Component and sub-components. An example of one such restriction is that all interaction is based on file system operations (e.g. *read*, *write*, etc). This has ramifications, because operations are stateless. Why this restricts the Relationship Component is that traditional transactional models (section 2.4.1) do not lend themselves well to a file operation based systems. More of these minor restrictions will be made apparent during this and latter chapters. However, it should be noted, that effort has been made to achieve a balance with the objectives of GLOMAR.

With all the benefits that a component-orientated architecture brings, there are some negative aspects why this approach has not been attempted earlier. Two aspects in particular are the costs associated with a component-oriented architecture and that this level of flexibility and expandability was not deemed necessary within the DFS domain.

The cost associated with existing component-oriented architectures are well documented (Szyperski 1997). This essentially indicates that the cost of metadata processing and late binding make component based programming more expensive than inline functional programming. This issue is of particular importance for DFSs and file systems, as performance is critical for providing an acceptable level of service. Thus, the merging of a performance dependent system and a potentially expensive programming approach would seem unacceptable.

However, the benefits of a component-orientated architecture in this situation (the need to support heterogeneity within a DFS) far outweighs the cost overheads. The motivation for this statement stems from observing performance dependent systems that implement a component-oriented architecture. One such system is the Windows NT operating system (Solomon and Russinovich

2000), which employs a component-oriented like architecture[1]. This system illustrates an approach that can be more extendable and flexible, without greatly compromising efficiency. An example of this is how developing system level modules (device drivers) for Windows NT (Nagar 1997) is far simpler and more powerful than for Windows 95/98/ME (Oney 1999), which does not follow a component-oriented design.

This argument applies to GLOMAR also, as the need for flexibility outweighs the need for performance (however, this does not mean performance issues are ignored). By utilising appropriate concurrency approaches to manage consistency, GLOMAR ensures that resources and users are serviced adequately. Thus, the potential degradation of performance experienced by the implementation of GLOMAR would be offset by the efficient and effective utilisation of other more critical resources.

4.2 Relationship Component Design

The actual design of the Relationship Component stems from a number of issues that are intrinsic to DFS implementations and the requirements that a multiple consistency model approach imposes. The first issue is how best to abstract the concurrency control and consistency maintenance functionality into a single module. The second issue is how to define the context of a consistency model.

The actual design of the Relationship Component abstracts these two issues into an element that contains the specific concurrency control and consistency maintenance functionality and an element that defines the context for which it is valid. In other words, the Relationship Component partitions code and context.

The need to partition the Relationship Component into two elements stems from the necessity to simplify development. In other words, packaging code and context in an ad-hoc manner does not allow for element sharing and independent development. For example, the implementation of a context element can be easily shared across different Relationship Component implementations.

[1]NT implements an object oriented model, written in C

4.3 Relationship Component Structure

The structure of the Relationship Component is more complex than simply partitioning code and context. As a result, the Relationship Component consists of three coarse grain sub-components;

- **Consistency Model.** A sub-component responsible for the encapsulation of concurrency control and consistency maintenance functionality.
- **Relationship Scope.** A sub-component responsible for defining context, that being the scenario a component is valid.
- **Clone List.** Another sub-component responsible for defining context, that being data objects a component governs.

The choice of this structure was deemed the best balance between functionality, interface, flexibility and complexity.

4.3.1 Consistency Model

The functionality of the Consistency Model[2] sub-component is to manage an appropriate level of concurrency and maintain an acceptable level of consistency. The interface of this sub-component is based on file system operations (figure 4.1). It reflects the basic operations available to a file system. The reason for using file system operations is due to a number of benefits. These include compatibility with existing file system implementations (heterogeneity), the ability to support fine grain consistency models and provide flexibility when implementing solutions.

The motivation for using file system operations as the basis for the Consistency Model interface is that this approach becomes highly portable to other platforms and DFS implementations. For example, systems such as the Vnode interface in Unix (Rosenthal 1990) and the Filter Drivers in Windows NT (Nagar 1997) allow for intermediate layers to be inserted into the file system call stack. Since their interfaces are based on file system operations, GLOMAR's approach fits elegantly into these systems.

The Consistency Model interface is low level intentionally. It deals with specific operations, rather than an abstracted high-level concept (like a transaction).

[2]The uncapitalised consistency model term refers to a description of concurrency control and consistency maintenance functionality. The capitalised Consistency Model term is used to describe GLOMAR's component implementation of a consistency model.

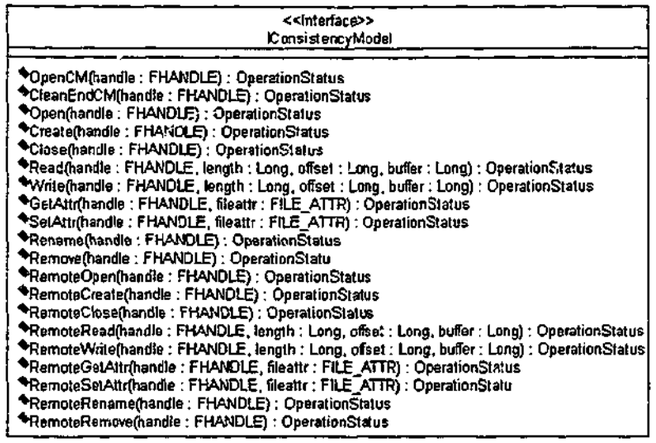

Figure 4.1: UML Diagram of the Consistency Model

The benefit of such a low-level interface is the fine grain control available for a file operation. For example, consistency maintenance can be tailored for an individual *write* operation, rather than a high-level *save event* (containing many *write* operations). This fine grain level of control is a feature of using file system operations as the basis of the interface.

The Consistency Model interface by its nature is very generic, consisting of basic operations, with basic known types used as parameters. In some respects this same interface design is a quasi-standard for interacting with a file system (Nagar 1997; Huston and Honeyman 1993). Such an interface design results in improved flexibility, as the interface uses basic types and has file system events clearly defined. However, it has the greatest impact for housing existing approaches, as the interface is similar to others used in existing systems.

The negative aspect of using file system operations is the loss of application-specific information pinned to an operation. Most file systems are not concerned with specific information other than the details of the request for it to service. As a result, additional information that could be used to improve the concurrency control and consistency maintenance process is not available. For this reason, the signature for each file system primitive is extended to include a *tag* string parameter [3]. The *tag* string parameter can contain information that an application might deem useful to a Consistency Model. However, there are some restrictions on its usage. If the Consistency Model is implemented transparently, then there is no way for this information to pass. This is because

[3]The *tag* string parameter is a member of the FHANDLE type

the file system has no explicit knowledge of the *tag* string. The reason for this is the underlying file system implementations have no facilities to handle or pass parameters, which are not directly related to the defined interface or interaction of the file system. If GLOMAR is called explicitly by an application, the *tag* parameter can be handled, as GLOMAR has the facilities to pass the *tag* parameter in addition to file operation information. Thus, a trade off between transparency and the ability to pass application-specific information is achieved.

One of the unique aspects in the design of the Consistency Model interface is the partitioning of file system operations into two types of operations, *Local* and *Remote*. This is done in order to identify the origin of an operation. The distinction between the origins of an operation is important for Consistency Models, as it reflects the nature of the operation. For example, local operations originate as a result of an application generating an operation locally. These types of operations are usually the catalyst for some form of consistency maintenance event. On the other hand, a remote operation is usually as a result of a consistency maintenance event (figure 4.2).

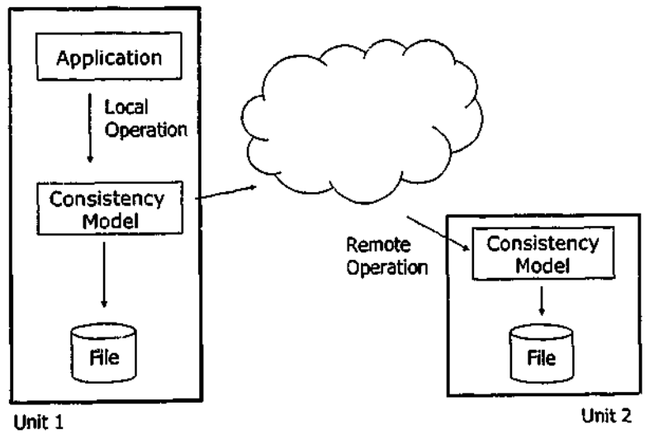

Figure 4.2: *Local* and *Remote* Operations

To ensure that local operations are handled differently to remote operations, the Consistency Model rigorously enforces an interface for both local operations and remote operations. A by-product of this distinction is that specific functionality based on the type of operation can be defined, thus giving another level of flexibility in relation to consistency management.

4.3.2 Relationship Scope

The Relationship Scope sub-component is one way (the other way is via the Clone List) of defining the context of the Relationship Component. Thus, the purpose of the Relationship Scope is to inform GLOMAR if a Relationship Component is best suited to a particular scenario or not.

The concept of defining a scenario for a Consistency Model is new in relation to DFSs and is one way of improving the configurability of Consistency Model implementations. The reason scenarios has not been formally defined is that most systems implement a single consistency model approach. However, with implementations that use multiple consistency models (Tait 1993; Dwyer 1998a), the criteria for determining the scenario is usually set within the DFS itself. These criteria are mostly concerned with static properties that are not easily adapted or changed. For example, a system might define available bandwidth based on packet loss, regardless of whether a more appropriate approach is available.

The interface of the Relationship Scope (figure 4.3) consists of four methods. The methods are simplified, with only a Boolean result indicating success or failure. This aids in the quick and efficient determination of context. The motivation for using a Boolean result was based on two specific reasons. Firstly, a Relationship Component is only valid or invalid for a scenario. It is unlikely that a Relationship Component would be partially valid for a scenario. Secondly, the methodology of the Relationship Scope implies that all the processing to determine the validity of a Relationship Component should be done within the context of the Relationship Scope.

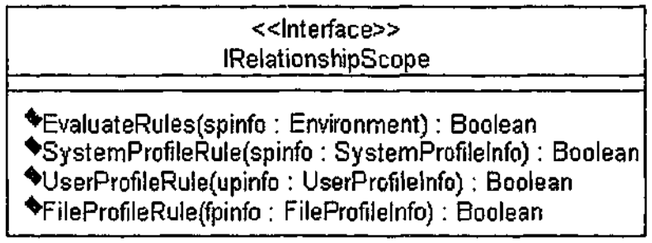

Figure 4.3: UML Diagram of the Relationship Scope

The most important method of the Relationship Scope interface is the *EvaluateRules* method. This is the entry point used by GLOMAR into the Relationship Scope. The results from this method determine if the Relationship Component is valid or not. Implementing the Relationship Scope is done by either implementing the functionality for determining the scenario within the

EvaluateRules method or using system generated information, delegating the processing to other methods. GLOMAR favours using system generated information, as the resulting cost of processing the Relationship Scope is less expensive than implementing the functionality independently.

Determining the scenario is done by evaluating the system generated information returned from the System Grader (section 5.2.5), which is passed into the Relationship Scope. However, the information being passed in might be too detailed for the needs of most Relationship Components. Thus, this information can be passed to other specific methods (that also make up the Relationship Scope interface) to be broken into a more consumable format.

These methods consist of *UserProfileRule*, *FileProfileRule* and *SystemProfileRule*. They are meant to decentralise the processing of determining the scenario, based on a simplified taxonomy (section 5.2.5 for details). To consume the information generated by the System Grader requires that these methods be implemented and be explicitly invoked by the *EvaluateRules* method (figure 4.4). If they are not, then they are never called. Only the *EvaluateRules* method is implicitly called by GLOMAR.

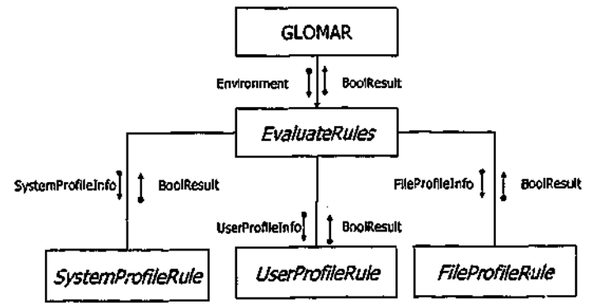

Figure 4.4: Relationship Scope Invocation

The Relationship Scope can now determine its validity. This interface thus provides the structure required to define the conditions for which this Relationship Component is valid.

4.3.3 Clone List

The Clone List provides a mechanism to map a Relationship Component to a data object (or set of data objects). In other words, the Clone List defines which Relationship Components govern which data objects. As a result, the Clone

List provides another mechanism to define the context of which a Relationship Component is valid.

The reason for using the term Clone List is that within GLOMAR, data objects (files) and encapsulating types (directories, volumes) are referred to as Clones. These Clones are not physical objects, but are rather references to files, directories and volumes within a file system. The motivation for using the term Clones is that GLOMAR views distributed data objects as clones of each other.

Prior to mapping Clones to Relationship Components within the Clone List, it is essential that Clones are firstly defined. Within GLOMAR, a Clone can be defined as one of four possible types; files, directories, volumes and systems (figure 4.5). As their name suggests, each Clone types maps directly to a corresponding type within a file system. The only major difference is that a file Clone type references a physical file object, where as a directory, volume and system Clone type encapsulate other Clone types. This is analogous to how directories contain files and volumes contain directories within existing file systems. These encapsulating types allow a hierarchical relationship between Clones to be expressed within GLOMAR.

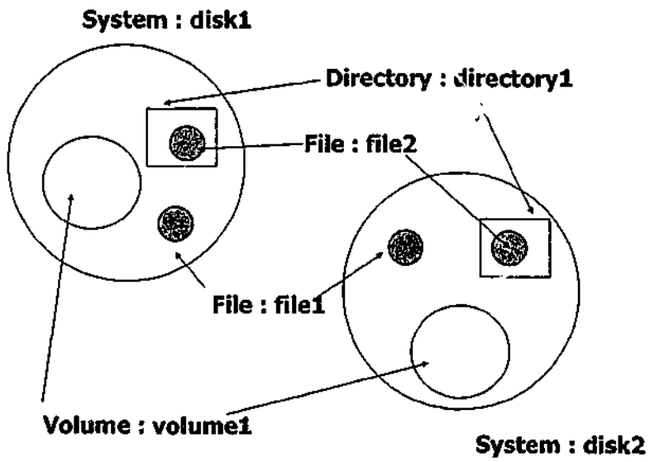

Figure 4.5: Clone Types

File Types

The file type is the base Clone type. Unlike other systems where the base type can be a block of a file, it was felt that a whole file approach for GLOMAR was more appropriate. Thus a file type maps directly to a whole file object.

This decision arose from experience with disconnected systems (Huston and Honeyman 1993). A block approach during network partitioning is more difficult to maintain and adversely effects resource processing. Experience also shows that whole files on average are small in size and that current bandwidth is adequate for most needs (Ousterhout et al. 1985; Baker et al. 1991). For example, a Consistency Model is created for a disconnection environment, for which clients are disconnected from the network for extended periods. Using a block approach reduces the availability of the replica, as not all elements of the file might be present locally when disconnected from the server. Thus, a situation might arise where an operation is to be performed on part of a file that is not available. This situation would not arise if a whole file was available and thus is the motivation for using whole files rather than blocks.

Defining the file type within the Clone List only requires creating an entry, assigning a globally unique identifier to it and referencing the physical file object from the local file system.

Encapsulating Types

As stated, the directory, volume and system types are used to encapsulate other Clone types. The main purpose for providing encapsulation Clone types is to simplify the representation of a large set of files. For example, it is far simpler to allocate a single directory to a Relationship Component than individually assigning every file object within a specific directory.

However, the three encapsulating types cannot be used indiscriminately to represent any clones. Each type has specific purposes and rules. For example, the directory types purpose is to reference a number of files within a single directory. The volume types purpose is to encapsulate any number of files and directories. Finally, the system type is the all encapsulating type that can contain volumes, directories and files.

Defining the encapsulating types within the Clone List only requires creating an entry, assigning a globally unique identifier to it and referencing all the appropriate Clones types within it.

Clone Assignment

Once Clones have been defined, the process of assigning them to Relationship Components is straight forward. For each Relationship Component entry within the Clone List, an unlimited number of Clone types can be added. For file Clone types, the relationship is straightforward, a Relationship Component governs a specific file. When assigning an encapsulating Clone type (either a directory, volume or system) to a Relationship Component, the Relationship Component governs all Clones defined within that encapsulating Clone type.

The benefit of being able to represent large file sets using encapsulating Clone types within GLOMAR results in improved flexibility in relation to configurability. For example, a Relationship Component can be implemented as a coarse grain approach (governing all files within a system) by using the system type (as it represents the whole system). On the other hand, a Relationship Component can be implemented as a fine grain approach (governing only a subset of the file system), using the file type, as it represents a particular Clone. The Clone List mechanism allows for both of these cases to be supported concurrently within the same system. Thus, allowing GLOMAR to support both fine grain consistency control and corse grain consistency control simultaneously.

However, the benefit of being able to utilising encapsulating Clone types creates problems when determining the appropriate Relationship Component to invoke when an operation occurs on a Clone governed by two Relationship Components. For example, a coarse grain Relationship Component is set to govern a directory. Within that directory is a file for which a fine grain Relationship Component is already assigned. To GLOMAR, both Relationship Components have an equal level of ownership for that particular Clone. Thus, how does GLOMAR resolve which of the Relationship Components has more ownership of that Clone, when an operation for that Clone occurs?

It is assumed in GLOMAR that Relationship Components that directly reference an individual Clone have a higher priority than ones that indirectly reference a Clone. In other words, a Relationship Component that references a file Clone type has more ownership than a Relationship Component that references a file via a directory Clone type. The justification for this is that in reality, the fine grain Relationship Component is more suited to the Clone than the coarse grain Relationship Component. The reason is that its implementation is better suited and more specific to a particular file than a generic coarse grain Relationship Component.

To define this ownership, a classification is assigned detailing the ownership hierarchy based on how a Relationship Component references a Clone within the Clone List. Thus, each Clone within the Clone List is tagged as either being *explicitly* or *implicitly* referenced by a Relationship Component. Clones that are referenced directly are said to be explicit, whereas Clones that are referenced indirectly are said to be implicit. GLOMAR uses this information in determining which Relationship Component to use for a particular Clone.

Clone List Implementation

The reason Clone assignment (as defined within the Clone List) is not packaged with the Relationship Scope is due to its nature, its purpose and how it is used. For example, the Clone List and Relationship Scope are both used to determine when to implement a Relationship Component. However, the process used to discover context within the Relationship Scope is different to that used within the Clone List. The basic difference is that information generated from the Relationship Scope is computed at run-time, based on rules. Whereas the information generated from the Clone List, is static and defined upfront. For this reason, system generated information is partitioned in the Relationship Scope and file system specific information is defined within the Clone List.

Unlike the Consistency Model and Relationship Scope, the Clone List is conceptually only a sub-component. In reality, since the modification of the Clone List is tailored to individual systems, the data store holding the relationships is external to the actual component itself. This loose coupling improves a Relationship Component's configurability, as system specific elements (like physical addresses of files) can be adjusted and modified on a per system basis. Thus, changes to the Relationship Component itself are unnecessary. The specific details of the Clone List are in Section 6.2.3.

4.4 Relationship Component Issues

The Relationship Component is more than the three sub-components packaged together. It also includes additional metadata to define how GLOMAR implementation issues are handled. The issues that are addressed with the Relationship Components metadata include the instantiation model, threading model and component life cycle. The reason these issues are addressed at this level is for simplicity. Also the cost of dealing with such issues at the sub-component

level was unfeasible and unnecessary for the needs of concurrency control and consistency maintenance functionality.

4.4.1 Instantiation

One of the limitations of the majority of file systems is that file operations are stateless. However, some concurrency control and consistency maintenance mechanisms need statefulness, for example a transactional model. GLOMAR is able to instantiate Relationship Components that are either stateful or stateless. GLOMAR determines the appropriate instantiation model to implement by inspecting the Relationship Component's metadata. This describes the model to implement.

In GLOMAR there are two models of instantiation, the *singleton* or *new instance* (figure 4.6). The singleton model (Gamma et al. 1995) is the stateless model. Operations are directed to a specific Relationship Component, which is selected from an already instantiated component. Since there is no direct linkage between the component and operation, no state information can safely exist, as potentially countless numbers of other operations for different files will use this same Relationship Component instance. In GLOMAR, there is a need for such an approach, as not all concurrency control and consistency maintenance implementations require state information. For example, some implementations might treat each operation as a single entity and have no need for state information to be stored. In addition, the singleton model is more efficient as the appropriate Relationship Component is already instantiated, with no need to incur the additional overheads associated with creating a new instance.

Nevertheless, there may be situations where the need for statefulness outweighs the cost overheads of the new instance approach. With this approach, a new instance of the specific component is created for that particular operation. However, instantiating a new instance for each operation does not provide state information across multiple operations. For this reason, when a Relationship Component specifies that the instantiation type is a new instance, this indicates that a new instance of the component will be available for the duration of its scenario. In other words, this instance will be used for all operations until it is deemed invalid. Once the scenario of a Relationship Component is over, the instance is destroyed. For example, consider a series of *open*, *reads*, *writes*, and *close* operations on a file. The *open* operation triggers GLOMAR to create a

new instance of the specific Relationship Component. As the operations continue, they are directed to use this instance. Within this instance, some form of state information is being stored. From operation to operation, this state information is available. However, it is destroyed when the scenario changes.

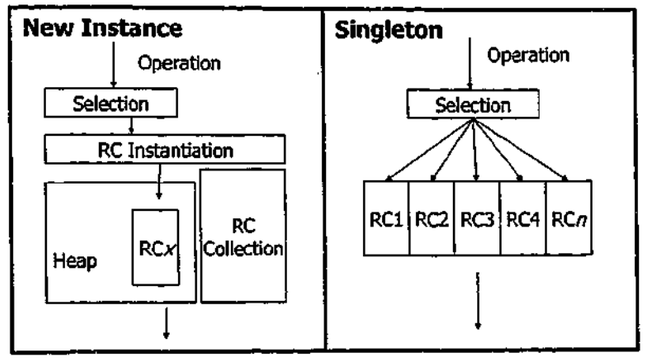

Figure 4.6: Relationship Component Instantiation Model

4.4.2 Threading Model

With the ability to create instances of a Relationship Component, the possibility of concurrent operations performing in a non-serialised manner arises (this is assumed normal in a *singleton* Relationship Component). For example, consider two instances of the same Relationship Component performing operations on the same Clone. Without knowledge of each other and assuming exclusive control of the Clone, both Relationship Component instances are affecting each other's effort to maintain consistency. Figure 4.7 illustrates some of the issues that can arise from lack of concurrency control at the Relationship Component level. As a result, there was a need to control the conditions for the successful instantiation and management of Relationship Components. For this reason, the Relationship Component has the ability to define the threading model best suited to its needs. However, this threading model is far simpler than other component threading models (e.g COM (Box 1998)), as it was designed for the purpose of Relationship Component management and the unique nature of GLOMAR.

The three threading models defined within GLOMAR are:

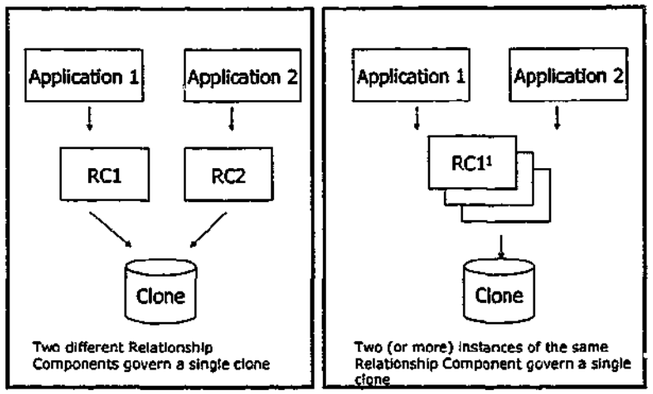

Figure 4.7: Relationship Component Threading Issues

- **Single Instance, Exclusive Lock** - A component defined as a Single Instance identifies that it is to be the only instance running within GLOMAR. This ensures that other instances do not interfere with the task of this Relationship Component. The Exclusive Lock specifies no other Relationship Component holds a reference to the particular Clone. This component will fail to create, if any of these conditions are not met.

- **Single Instance, Non Exclusive Lock** - This instance of the Relationship Component must be the only instance running within GLOMAR. However, other Relationship Components can have a reference to the Clone. If however an Exclusive lock is found for one of the Clones, then this operation fails. Only Non Exclusive locks can be shared.

- **Multiple Instance, Non Exclusive Lock** - Any number of instances can exist for this instance to be created. Other Relationship Components can have a reference to a Clone. If however an exclusive lock is found for one of the Clones, then this operation fails. Only Non Exclusive Locks can be shared.

4.4.3 Life-Cycle

As a consequence of the multiple consistency model approach, Relationship Component instances will be starting and stopping based on different scenarios during the life time of GLOMAR. Predicting when these events will occur is impossible as they are based on the scenario of the Relationship Component.

Thus, there needs to be an elegant solution to handle these events when they occur. The most appropriate approach is to mirror the constructor/destructor mechanism within object-oriented programming. For example, when a Relationship Component is instantiated, a constructor like method is called, which starts all the necessary functionality. When a Relationship Component is invalidated, prior to it being destroyed, a destructor like method is called, invoking the functionality associated with finalising the component.

Since such events are tightly coupled to the functionality of the Consistency Model, the Consistency Model interface has been extended to include two additional methods; the *OpenCM* method and the *CleanEndCM* method. The *OpenCM* method acts much like a constructor, as it is always the first method called. It is called only once and only upon the invocation of a new Relationship Component. The benefit of this method is that code placed inside can be invoked before any operation has been performed. For example, the *OpenCM* could be used to replay a log of operations recorded while in a disconnected mode.

The *CleanEndCM* method has a more important task, as it is invoked because of the Relationship Component being invalidated. Thus, code within this method attempts to generate a consistent state, ensuring that operations are not lost and that they can be recovered easily. However, the nature of this method is blocking, thus the system will wait until the *CleanEndCM* method is finished before proceeding. As performance is an issue, a Relationship Component should avoid creating procedures that block indefinitely. For this reason, each Relationship Component is given a timeout period. When that period expires, regardless of the *CleanEndCM* method state, the Relationship Component instance is destroyed.

By including the ability to define a threading model, instantiation model and life-cycle in GLOMAR improves the manageability of Relationship Components. In the attempt to cater for the differing needs of Relationship Components, a balance between functionality and cost has been achieved. However, a deep understanding of the ramifications of these properties is necessary, as implementations might fail to perform as intended. Thus, ensuring Relationship Component issues are addressed is just as important as the functionality itself.

4.5 Summary

This chapter details the approach used to encapsulate the concurrency control and consistency maintenance mechanism into a single unit, the Relationship Component. The motivation, the structure and the design decisions have also been discussed.

The Relationship Component consists of three sub-components:

- **Consistency Model.** This sub-component houses the concurrency control and consistency maintenance functionality.
- **Relationship Scope.** This sub-component describes the scenario used to define when to implement a component.
- **Clone List.** This sub-component defines the files, directories and volumes it governs.

The actual Relationship Component was discussed, focussing on issues that arise from being implemented within the multiple consistency model approach.

The next chapter (Chapter 5) discusses the GLOMAR middleware layer, detailing how it manages and supports the Relationship Components.

Chapter 5

GLOMAR Middleware Layer

This chapter discusses the mechanism (the GLOMAR middleware layer) responsible for the handling of file system operations, as well as managing and servicing Relationship Components. The proposed middleware layer is highly detailed and is one of the major contributions of this dissertation. This chapter details the aims of the GLOMAR middleware layer, the motivation for using a middleware layer approach, the justification for the final design and the specific details of each service of the GLOMAR middleware layer.

5.1 Aims

The primary aim of the GLOMAR middleware layer is to manage the interaction between the operating system and Relationship Components on one hand and between multiple applications and Relationship Components on another. This is achieved by creating a brokering system between file operations and a variety of different concurrency control and consistency maintenance mechanisms.

In addition to managing and servicing file operations, the GLOMAR middleware layer is flexible enough to provide a constantly adapting level of service, for existing and new Relationship Components. This means new and unique Relationship Components are not restricted by the GLOMAR middleware layer. This is of particular importance because future scenarios can be catered for easily.

An important aim of the GLOMAR middleware layer is efficiency. This is because file operations occur frequently and whose latency is critical. For this

reason, the GLOMAR middleware layer implements a balanced approach, considering the imposed constraints. The specifics of these constraints are detailed within this chapter.

The final aim of the GLOMAR middleware layer is to improve the process of concurrency control and consistency maintenance mechanism development. This is done by streamlining Relationship Component creation via the GLOMAR middleware layer providing services that are commonly found within commonplace DFS implementations. As a result, rather than creating common services for specific implementations, they are provided as part of the proposed framework. In addition, the GLOMAR middleware layer allows for the creation, modification and extension of these services easily.

5.2 GLOMAR Middleware Layer Design

Achieving the aims of the GLOMAR middleware layer requires a design that is flexible, extendible, manageable and efficient. The choice undertaken by GLOMAR was to follow the bridge software pattern (Gamma, Helm, Johnson, and Vlissides 1995). The justification for this design stems from similar systems, including plug-in architectures like Netscape plug-ins (Netscape 1998) and Universal Plug and Play (UPNP 2002).

Within the GLOMAR middleware layer, areas of responsibility are assigned particular services. The first set of services is responsible for providing the interface into the GLOMAR middleware layer. These services are:

- The *Local Operation Interface*
- The *Remote Operation Interface*

The second set of services is responsible for providing additional GLOMAR specific functionality for use by any Relationship Component. These are:

- The *Clone Distribution Manager* (for name resolution of data objects)
- The *Service Manager* (houses additional Relationship Component user-specific services)

The third set of services is responsible for the assignment of Relationship Components at run-time. Due to the complexity of this task, the functionality is divided into two services. These are:

Parameter	Description and Purpose
File Name	The file name is used to not only point GLOMAR to the correct data object, but also used for determining Relationship Components during the selection phase.
Operation Type	Indicates the type of operation being performed by the operating system. GLOMAR supports the following types: READ, WRITE, OPEN, CLOSE, GETATTR, SETATTR, CREATE, RENAME, REMOVE.
Offset	A position within a file an operation is referring to. This is only valid for READ and WRITE operations.
Buffer	The data structure used to carry modifications as a result of an operation. For example; READ operations would pass in an empty buffer, expecting it to be filled, whereas WRITE operations would contain a value to write at a particular offset.
Length	The length of the actual value within the Buffer.
Application ID	This is the identification of the application actually invoking the operation. This data is used as the basis for indexing.

Table 5.1: Specific information collected by the Local Operation Interface

- The *System Grader* (which determines the current scenario of the system)
- The *Relationship Component Repository* (handles Relationship Component management and selection)

All these services, in turn, are managed by the *Executive* (figure 3.4).

5.2.1 Local Operation Interface

The Local Operation Interface is solely responsible for capturing local file operations and operation specific information (in this case, file details). These operations and associated information are then forwarded to the Executive for further processing. See table 5.1 for details on the file information the Local Operation Interface collects.

Since application transparency is one of GLOMAR's aims, a user should not be explicitly aware of the activities of GLOMAR. For this to be achieved, part of the Local Operation Interface is designed as an additional layer that exists within a file system's Input/Output stack. Its purpose is to intercept file operations prior to completion by the file system. However, the Local Operation Interface does not replace the file system. Rather, it resides within the file system, forwarding operations to GLOMAR for further processing. When GLOMAR finishes with an intercepted operation, that operation is then passed back to the Local Operation Interface, which returns it into the file system. The operation is then allowed to continue its execution to lower level drivers within the file system.

The Local Operation Interface consists of two elements: the operating system specific functionali y and the bridge that connects it with GLOMAR (figure 5.1). For example, the implementation of the operating system specific functionality can be a Filter Driver in NT (Nagar 1997) or a Vnode module within UNIX (Rosenthal 1990). The similarity between these approaches and GLOMAR's interface design means that implementing the operating system specific functionality is made easier, as they both employ a file operation based interface.

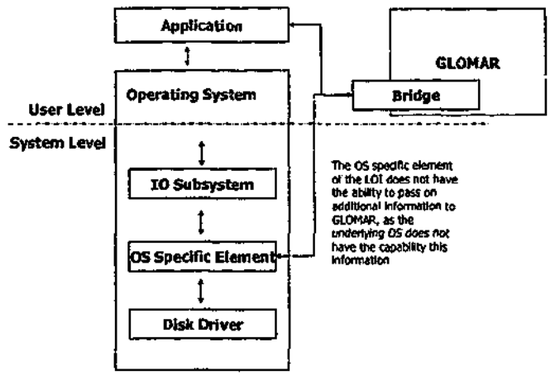

Figure 5.1: Local Operation Interface

However, not all situations require that a transparent approach be used. There are situations where a non-transparent approach would be suited, as illustrated by Odyssey (Noble 2000). Odyssey showed the benefits of a concurrency control

and consistency maintenance mechanism exploiting application-specific information, using a non-transparent interface. For GLOMAR to provide a similar service, the bridge element of the Local Operation Interface is visible to applications at the user level (rather than solely at the system level). However, no benefit would be gained if the interface remained the same between the user and system levels. Thus, the bridge interface is extended for applications that directly communicate with GLOMAR (figure 5.1). As part of the design, an application can pass any meaningful information to a Relationship Component, via the *tag* parameter (As was discussed in section 4.3.1).

Therefore, the Local Operation Interface provides not only a transparent approach for processing file operations, but also a non-transparent approach that allows applications to explicitly pass meaningful information. In addition, both approaches are implemented simultaneously.

5.2.2 Remote Operation Interface

The nature of a DFS (or any distributed system) is that operations (or results of operations) will be propagated to other members of the network. For this reason, GLOMAR has a specific interface that handles requests that originate from remote sources. The Remote Operation Interface provides this service. Its purpose is to handle remote file operations and ensure that they are forwarded to the appropriate Relationship Component. Within GLOMAR, the Remote Operation Interface ensures that the operations are directed to the remote interface within the Consistency Model sub-component (figure 5.2). Operation partitioning based on the origin is detailed further in section 4.3.1.

The Local Operation Interface and Remote Operation Interface both handle inbound file operations. As a result, they have some similarities, including stateless event based interaction, data types used and that results from the operation are returned to the caller. However, there are some differences with the Remote Operation Interface, due to the type of file operations managed. Primarily, the Remote Operation Interface is to be called only by remote Relationship Components and is not meant for direct use by applications. In addition, the information that is passed to the Remote Operation Interface is more detailed than the information passed to the Local Operation Interface. In the Remote Operation Interface, two additional parameters are passed, the *Relationship Component ID* and *Clone ID*.

These two additional parameters are passed into the Remote Operation Interface to avoid any ambiguity between the interaction of a local operation and a

remote operation. The Relationship Component ID is passed with an operation from the remote source. The Relationship Component ID informs the Remote Operation Interface that this operation was generated by a particular Relationship Component. The Remote Operation Interface then ensures this operation is forwarded to the same Relationship Component locally. As a result, any ambiguity is removed in relation to handling remote operations.

A Clone ID is also passed to determine which Clone (file) this operation is intended. The reason for using an ID rather than a name and absolute address (which is used in the Local Operation Interface) is that some files might have different absolute addresses. Thus using IDs (similar to a File Handle) ensures that operations reach their destination Clone, without having to enforce a rigorous addressing and naming scheme.

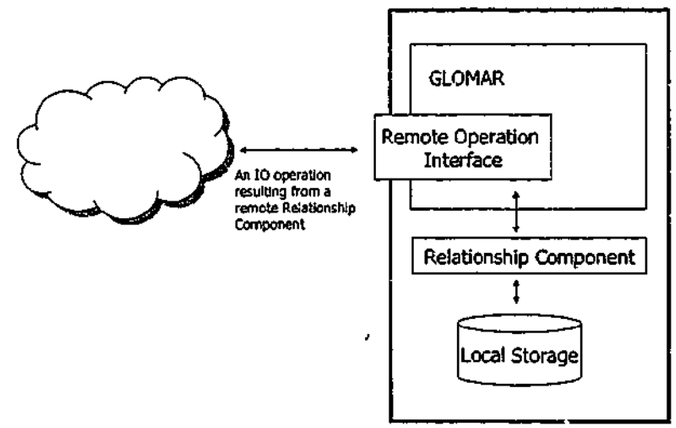

Figure 5.2: Remote Operation Interface

5.2.3 Clone Distribution Manager

In keeping with the aims of GLOMAR, much of the common functionality is provided by the GLOMAR middleware layer. One such example is the Clone Distribution Manager, which is responsible for the management of files. The purpose of the Clone Distribution Manager is to hide some of the complexity associated with DFS interaction. In a DFS, naming of replicated data objects becomes more complex when implemented within a distributed environment. For example, replicated files might be implemented within different branches of a local file system hierarchy. The Clone Distribution Manager provides a service to extract this information.

The Clone Distribution Manager is only tasked with providing simplistic functionality. More complex approaches can be created using the Service Manager (section 5.2.4). The Clone Distribution Manager is not implemented as a service within the Service Manager as much of the functionality is intrinsic to the operations of GLOMAR, in particular, to the decision making portion of the GLOMAR middleware layer.

5.2.4 Service Manager

In some situations, the Relationship Component methodology does not provide the requirements needed to service the concurrency and consistency needs of an application, user or device. For example, an external server process might be required to complement a Relationship Component. Implementing such a server process could be achieved by developing another mechanism that runs in conjunction with the Relationship Component. Thus, the GLOMAR provides for user-specific services to be added to the middleware layer. The Service Manager is the service that handles the instantiation and management of these user-specific services (figure 5.3).

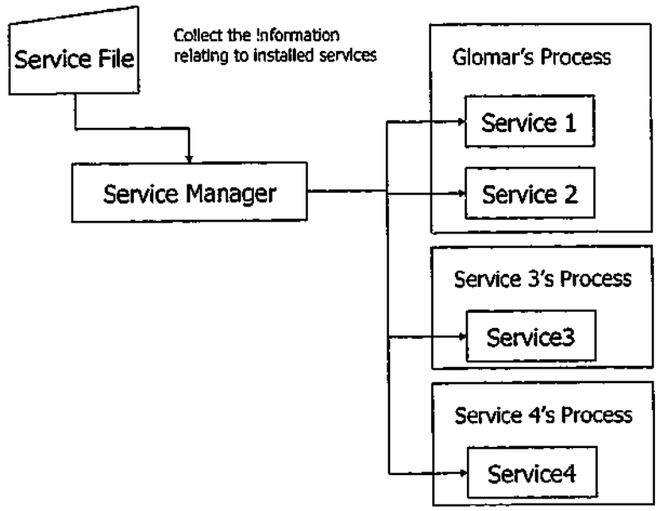

Figure 5.3: Service Manager

The Service Manager handles three types of services;

- **External.** These services run within their own process for the duration of the system.
- **Instantiated.** These services run for the duration of GLOMAR within GLOMAR's process. They start and stop depending on the lifetime of the GLOMAR instance.
- **Remoted.** These services expose their functionality via the GLOMAR communication channel and run for the duration of GLOMAR within GLOMAR's process. This is how intrinsic services (like the Clone Distribution Manager) are exposed.

Inclusion of a service into the Service Manager is done by editing a Service File. Within this Service File, the physical location of the service is indicated, as well as the entry point and the service type. Upon the start-up of GLOMAR, this file is read and the appropriate services started.

Unlike the Relationship Component, which must adhere to a detailed interface, a simple interface is defined for user-specific services. This interface consists only of a *start* and *stop* method. Within these methods, the functionality of the service is placed.

5.2.5 System Grader

Before selecting a Relationship Component or re-evaluating the legitimacy of a currently implemented Relationship Component, the Executive invokes the System Grader to collect information about the current scenario. Since this information is used to direct file operations, GLOMAR refrains from implementing a definitive mechanism for determining the legitimacy of a Relationship Component. Rather, GLOMAR emphasises that the evaluation process is "as accurate as possible" in the face of strict time constraints. This is achieved by providing a passive process for deriving information. Rather than every file operation resulting in a call to a set of system metric routines, every file operation performs a "pickup" of a shared data structure. This shared data structure contains a collection of scenario types and actual values, which are then passed into the Relationship Scope of each Relationship Component (figure 5.4).

GLOMAR maintains this information via an additional process (existing as a separate thread), which invokes specific functionality based on a defined set period. As a result, the shared data structure is constantly being updated, independent of the events occurring within the Executive.

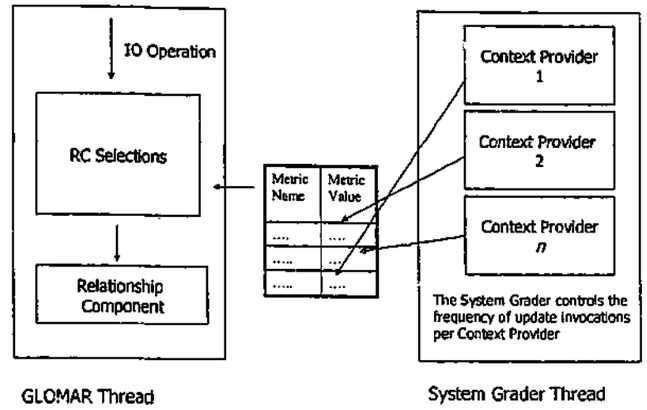

Figure 5.4: System Grader's *pickup* Approach

Traditionally, only a standard set of information (e.g. storage size, cpu speed, bandwidth) was available to describe a scenario. However, there are situations where specific information that is not part of the current set of information is required to define the scenario. One unique aspect of the System Grader is its ability to derive such specific information, by allowing additional scenario deriving mechanisms (*Context Providers*) to be added. These Context Providers are installed into the System Grader by editing the System Grader File. This file is used to inform the System Grader what values are to be inserted into the shared data structure. These values are only available after a restart of GLOMAR has occurred.

The process of updating the shared data structure is performed in two ways, by statically setting a value or by providing a pointer to a Context Provider. From this point, the System Grader takes responsibility for ensuring that the shared data structure is maintained in accordance with the System Grader File.

For classification purposes, scenario information is divided into a taxonomy based on the *User Profile*, the *File Profile* and the *System Profile*. The *User Profile* details possible interaction patterns of a user on a file. The *File Profile* details the type of file being processed. The *System Profile* is an abstraction of the current status of the hardware, software and location. This information is stored in a hierarchy, with the taxonomy defined as the root elements (table 5.2 and figure 5.5).

Both the *System Profile* and the *File Profile* allow for complex structures that mirror files systems and context providers to be represented. For example, a

Profile	Sub Branch	Profile
System	Hardware	Contains all Context Providers that evaluate the hardware elements of a system. For example, CPU, Networking, Storage, Power.
	Software	Contains all Context Providers that relate to the software elements of a system. For example, Operating System, Applications installed, etc.
	Location	Contains all Context Providers that relate to physical location. This is applicable to mobility-enabled applications.
File	Class	Contains all Context Providers that are specific for a file within the Clone list. In most cases illustrating the Operating Type (READ, WRITE or READWRITE) and Operation Action (STREAM, RANDOM, UNIFORMED, NOTHING).
	Type	Contains all Context Providers that relate to the type of file. This might include the nature of the file (text, database, multimedia, etc).
User	User	Contains all Context Providers that relate to the user of the system and how they might behave.

Table 5.2: Context Provider Taxonomy

tree structure that mirrors the files and directories on a file system. However, the *User Profile* does not have this facilitate. Rather simple, yet suitable entries within the *User Profile* adequately represent user details, without the need to resort to complex structures.

5.2.6 Relationship Component Repository

The Relationship Component Repository is responsible for the management of Relationship Components. This responsibility is divided into two areas;

- The creation and verification of a Relationship Component and
- The run-time selection of a Relationship Component based on the current scenario

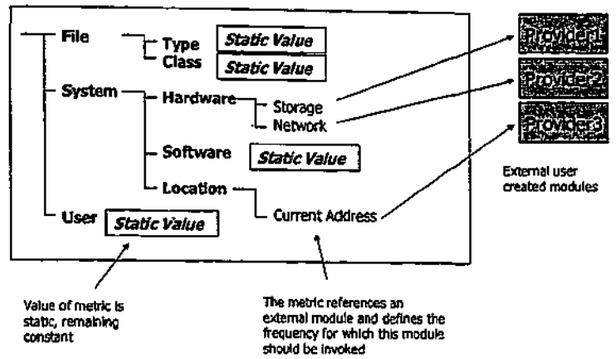

Figure 5.5: System Grader Taxonomy Structure

Relationship Component Instantiation

The ability to add different Relationship Components into GLOMAR is based on the decoupling of the Relationship Component functionality and middleware layer. However, as a result, this decoupling means the GLOMAR middleware layer has to actively seek out Relationship Components and assume responsibility for their creation and run-time storage.

The Relationship Component Repository creates and stores instances of all Relationship Components when it is started, rather than "just-in-time". The reason for doing this stems from the aim of the GLOMAR middleware layer to be as efficient as possible, as the selection phase has only to traverse the data store containing the instances of Relationship Components, rather than creating them each time.

However, this eagerness to instantiate Relationship Components does reduce the middleware layers ability to dynamically install new components at run-time. This design choice stems from the need to selectively determine the life-cycle operations of Relationship Component instances, by eliminating the ability of the middleware layer to add and remove Relationship Component instances. This improves the consistency of available Relationship Components to be used and fits within the install philosophy of Relationship Components administration as the possibility of Relationship Components not being available on remote nodes is reduced.

The events that make up the instantiation phase of the Relationship Component Repository include:

- **Finding all Relationship Component files.** All files found within a specific directory are loaded.
- **Each class within the files is checked for the correct interfaces and the correct parent object.** A Relationship Component is deemed valid only if the supplied base class *RelationshipComponent* is found within the inheritance tree, regardless of its depth. This primarily ensures the safety of the component being instantiated. Secondary to this, existing Relationship Components can be extended upon and still be valid Relationship Components (figure 5.6). For example, by inheriting functionality from an existing Relationship Component and adding additional functionality, results in a new component that has exploited code reuse and is valid.

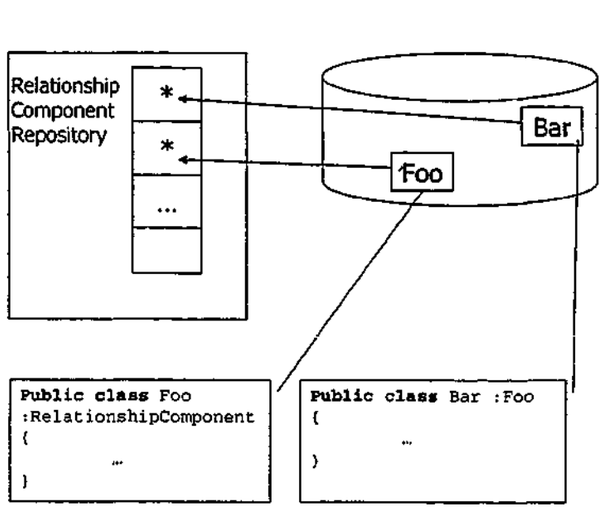

Figure 5.6: Extending Relationship Components

- **Create and store Relationship Component instances.** Once the validity of a class has been determined, an instance is created. This instance is stored within the Relationship Component Repository, using the class name as the index.

The result from these events is that all valid Relationship Components within a specific directory are instantiated, stored and available to be used.

However, having an already instantiated instance would seem to force all Relationship Components to appear as singleton objects (Gamma et al. 1995). As section 4.4.1 details, not all objects exhibit this instantiation model. The reason for implementing all instances as singleton (at this stage) is solely for accessing the Relationship Scope portion of the Relationship Component faster.

Since state information is not necessary for a Relationship Scope implementation (internal processing is static), it was determined that for efficiency reasons, a singleton design pattern would be used for accessing Relationship Scope functionality. However, the implementation of the Consistency Model portion of a Relationship Component is handled differently (section 6.3.6 for details).

Relationship Component Processing

When required, the Relationship Component Repository is asked to return the "most valid" Relationship Component depending on the current scenario. This involves evaluating all aspects of each Relationship Component, including each Relationship Scope and Clone List.

Since this evaluation event is a result of a file operation, an exhaustive process would adversely affect the system's performance. As a result, an efficient and yet rigorous evaluation process is required.

The solution is an evaluation process that exploits the frequency of certain events, so that events that are more frequent are tested early. This results in the evaluation process being divided into three sub-processes (*Existing*, *Parade* and *Investigation*), with each process imposing another level of complexity and another level of cost (figure 5.7). Such an approach ensures that efficiency is compromised only when deemed necessary.

The Existing Process

The first process of *Relationship Component Processing* consists of ensuring that the currently invoked Relationship Component is still valid (figure 5.7). Prior to an operation being performed on a Clone, the status of the governing Relationship Component is evaluated. Since the only criterion for changing a Relationship Component is a change in the scenario, the Relationship Scope is checked. If the current scenario is valid for the invoked Relationship Component, then it is used.

However, if the scenario has changed, therefore rendering the Relationship Component invalid, the second process (the *Parade* process) is invoked. Also, if a Clone does not have an invoked Relationship Component governing it, this process is by-passed and the evaluation process is moved directly to the *Parade* process.

The Parade Process

The *Parade* process promptly creates a list of suitable Relationship Components based on the set of Clones that they govern (figure 5.7). In other words, if an operation is about to be performed on file x, then only the Relationship Components that define file x in their Clone List are deemed suitable. The resulting list can either be:

- Only one Relationship Component matches the Clone
- Many Relationship Components match the Clone

If only one Relationship Component is selected, then it is assigned. There is however, a problem with this process. For example, if one Relationship Component is returned, but its Relationship Scope indicates that it is invalid, it still can be used. This is because the process that evaluates the Relationship Scope happens after this process. This is a constraint of the system and will be addressed in future work (section 9.2).

The *Parade* process becomes more complex when a number of Relationship Components are returned. In this case, an additional process is required to evaluate each of the Relationship Components in more detail to determine the "most valid". This is handled by the *Investigation* process.

The reason for evaluating the Clone List of a Relationship Component first, is that much of the processing to determine the appropriate Relationship Component to invoke would be done at this stage. In addition, the likelihood of multiple Relationship Components per Clone is low. For example, a Clone in most cases will have only one Relationship Component. Thus, evaluating the Clone List prior to evaluating the Relationship Scope means the majority of processing would be resolved early within *Relationship Component Processing*.

The Investigation Process

The first step of the *Investigation* process is to evaluate in detail each of the Relationship Components considered valid from the previous process (figure 5.7). This involves invoking the Relationship Scope of each Relationship Component and determining their validity. The time to complete such a task is highly dependent on the implementation of each Relationship Scope. The results from this process can be one of the following:

- No Relationship Components were valid
- One Relationship Component was valid
- Many Relationship Components were valid

If no Relationship Components were valid, the operation is aborted. If one Relationship Component is the result of the investigation, it is instantly returned and invoked. However, if more than one Relationship Component is returned, the *Investigation* process further evaluates the list of Relationship Components to select the most suited one.

The second and final step involves the use of the Clone List. Each Clone within the Clone List of a Relationship Component is referenced either implicitly or explicitly (section 4.3.3). The Clone's reference is based on whether the Relationship Component directly references the current Clone (*explicit*) or it is inferred as a result of a directory Clone implying ownership (*implicit*). Using this reference, the Relationship Component Repository lists explicitly referenced Clones over the implicitly referenced ones. The Relationship Component that is listed first is returned and invoked.

5.2.7 Executive

The Executive's purpose is to manage all GLOMAR middleware layer services and handle every file operation. The Executive acts as a file operation dispatcher, ensuring that file operations are directed to appropriate Relationship Components. The life-cycle of the Executive (figure 5.8) consists of:

- The Local Operation Interface passes in a file operation
- Determine if the Clone is being governed by GLOMAR
- Determine if the existing Relationship Component is still valid

- Ask the Relationship Component Repository for new Relationship Component
- Implement new Relationship Component

The initial step is to determine if the Clone being accessed is governed by GLOMAR. If not, the file operation is returned and allowed to continue. If so, then the Executive must determine if an existing Relationship Component is currently invoked or a new one has to be instantiated. This is done by searching a stored collection of Relationship Component instances, indexed on the *Process ID* (GLOMAR allows only one Relationship Component instance per process). This stored collection is not to be confused with the collection of Relationship Component instances owned by the Relationship Component Repository.

The Executive then passes the results from this search to the Relationship Component Repository, which determines the legitimacy of the existing Relationship Component and/or initiates *Relationship Component Processing* (section 5.2.6). The results from this process can be;

- A failed operation
- The invoked Relationship Component is still valid
- There is no invoked Relationship Component, thus a new Relationship Component is returned
- A new Relationship Component is returned to replace an invoked component.

If the operation fails, the Executive returns this failed operation to the caller application. If however, the invoked Relationship Component is still valid, the Executive finds the Relationship Component instance within its own stored collection and invokes the method corresponding to the operation. From this point, control of the operation is handled by the appropriate Relationship Component.

However, when a new Relationship Component is returned, regardless of whether it's a new component or a replacement, the Executive implements a number of additional steps, referred to as *Relationship Component Implementation*.

Relationship Component Implementation

The process of *Relationship Component Implementation* involves either assigning the Relationship Component Repository selection to a non-governed Clone,

or dynamically changing the currently invoked Relationship Component to a more suitable one. The process of assigning a Relationship Component to a Clone that has no governing Relationship Component is straightforward. All operations are redirected to this Relationship Component from this point onwards. However, the assigning of a Relationship Component to a Clone that has an invoked Relationship Component is more complex. The Executive shuts down the currently invoked Relationship Component and assign its replacement.

The process of changing to a new Relationship Component is done by calling the method *CleanEndCM* (member of the Consistency Model interface, see section 4.3.1), which is a blocking call that attempts to maintain consistency in preparation for the Relationship Component being shutdown. If the *CleanEndCM* method has failed to return within a set time (set time defined within the metadata of the Relationship Component), the method is aborted and the Relationship Component is shutdown.

The Executive then asks the newly selected Relationship Component whether it can coexist with other Relationship Components. In other words, the threading model for the Relationship Component is inspected. Depending on the Relationship Components already invoked, the Executive can choose whether to invoke the newly selected Relationship Component or not. If approval is given, the Relationship Component is asked what type of instantiation model it prefers.

Finally, the Relationship Component is either instantiated as a *singleton* or *new instance* (section 4.4.1). If the Executive determines that a Relationship Component prefers a singleton approach, a reference from a list of already instantiated components is returned (which reside in the Relationship Component Repository). If however, the Executive determines that a Relationship Component prefers a new instance of the object to be created, a new instance of the Relationship Component is created dynamically. This new instance is then stored in-memory within the Execute for future access. This then concludes the process of implementing a new Relationship Component.

Handling Failures

The Executive's primary role is to dispatch file operations, with no concern for the results returned from a Relationship Component. However, there are situations were the Executive (and GLOMAR) would benefit from knowing the status of an operation. For example, consider a disconnection occurring mid operation. As the remote host cannot be contacted, the operation would fail

as the appropriate Relationship Component (built for connectivity) would time out. However, why should the operation fail when a Relationship Component built for disconnection is available? The solution is for the Executive to monitor the results returned from a Relationship Component and replay the operation if a failure occurs. Prior to the operation being replayed, the Executive again calls the Relationship Component Repository and retrieves the most valid Relationship Component. In the case of loss of connectivity (the previous example), the second call to the Relationship Component Repository would result in the appropriate Relationship Component (disconnection) being returned and used. By allowing the process to feedback on itself, the operation that would have been lost due to a disconnection would be processed effectively.

5.3 Summary

This chapter has detailed the approach used to manage Relationship Components, including the aims, the design and the decision making process of the GLOMAR middleware layer. This chapter has focussed on the aims of the GLOMAR middleware layer, in particular managing interaction, flexibility, efficiency and streamlining Relationship Component development.

The resulting GLOMAR middleware layer design was divided into three areas of responsibility: the interfaces for the operating systems and applications (Local Operation Interface and Remote Operation Interface); the support services to improve the task of creating Relationship Components (Clone Distribution Manager and Service Manager); and the components that provide the management of Relationship Components at run-time (Relationship Component Repository, System Grader and Executive).

The next chapter discusses the implementation of GLOMAR, including the implementation of the Relationship Component and the GLOMAR middleware layer.


```
string RCRSelection (CloneID, CurrentScenario)
{

  // the Parade process returns all Relationship
  // Components that can govern this Clone.


  RCList = GetSuitableRC(CloneID);

  if (RCList.Count == 0)
    return "failed"; // Clone cannot be governed.
  elseif (RCList.Count == 1)
    return RCList(0);  // Relationship Component found and returned.
  else
  {
    // the Investigation process

    SuitableRCList = null;


    while (RCList.Count > counter)
    {

      // the Relationship Scope of each
      // Relationship Component is checked

      if (CheckRS(RCList(counter), CurrentScenario) == true)
        SuitableRCList.Add(RCList(counter));

      counter = counter + 1;
    }

    if (SuitableRCList.Count < 1)
      return "failed";
    elseif (SuitableRCList == 1)
      return SuitableRCList(0);
    else
    {

      // the Advanced Investigation process


      while (SuitableRCList.Count > counter2)
      {
        // return the Relationship Component that
        // is defined as EXPLICIT


        if (CheckCloneType(SuitableRCList(counter2)) == EXPLICIT)
          return SuitableRCList(counter2);

        counter2 = counter2 + 1;
      }
      return SuitableRCList(0);  // Return the first instance
    }
  }

}
```

Figure 5.7: *Relationship Component Processing*

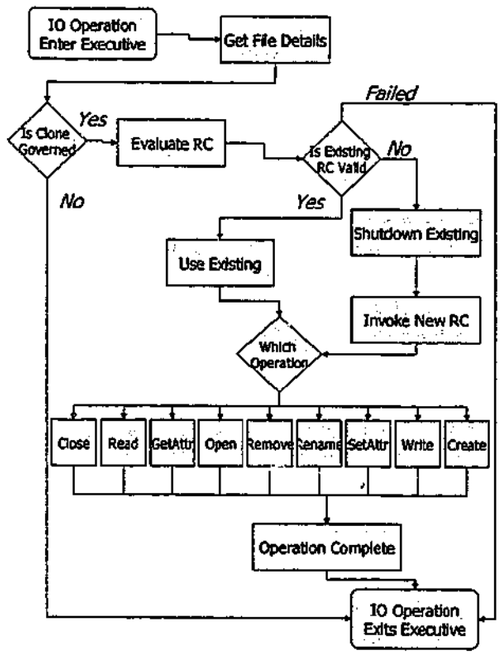

Figure 5.8: File Operations within the Executive

Chapter 6

GLOMAR Implementation

This chapter details the full-scale implementation of the GLOMAR framework within a DFS. The GLOMAR framework implementation is comprised of approximately 6000 lines[1] of .NET (Watkins, Hammond, and Abrams 2003)[2] code, consisting of a mixture of C#, Managed C++ and Visual Basic.NET. This chapter also examines the target development platform, the implementation of the Relationship Component and the internal particulars of the GLOMAR middleware layer.

6.1 Development Platform

By understanding the constraints of many file systems and using common elements of languages (and component technologies), the final design of GLOMAR is as platform independent as possible. Nevertheless, the development platform chosen for the full-scale implementation of the GLOMAR framework is .NET. This decision was made due to three critical factors, namely:

- Cross Language Support
- Advanced Component Architecture
- Multiple Platform Support

Cross language support is achieved within the .NET framework, as all code is compiled and emitted into an Intermediate Language (IL) (ECMA 2001). This

[1] This figure excludes lines of code that make up individual Relationship Components and Context Providers.

[2] The description the .NET implementation makes use of some .NET specific terminology. For details on their meaning, refer to the Glossary.

is similar to the byte code approach used in Java (Gosling et al. 2000). However, unlike Java, which implements one language for many platforms (though other languages have been ported to emit Java byte code (Gough and Corney 2000)), .NET has a number of compilers that emit IL. This is critical for the GLOMAR framework, as it allows legacy code from existing concurrency control and consistency maintenance implementations to be integrated into GLOMAR easily. For example, a concurrency control and consistency maintenance mechanism written in C++ could be incorporated (with only some minor alterations necessary) into the Relationship Component, as .NET provides a C++ compiler that emits IL code.

Central to the GLOMAR philosophy is the concept that components are the foundation of its implementation, whether being the implementation of a Relationship Component or the services that make up the GLOMAR middleware layer. Current component technologies implement approaches that are difficult to manage (COM (Box 1998)), fail to offer the native support for common component functionality (in the case of Dynamically Linked Libraries) or are too tightly coupled with a platform or language. The .NET component approach offers the GLOMAR framework the best balance between complexity and flexibility. For example, the interfaces and metadata that make up a Relationship Component can be rigorously defined and controlled. Also the advanced *Reflection* (Watkins, Hammond, and Abrams 2003) library supplied by the .NET framework offer a simplified method of handling dynamic invocations required for late binding of a Relationship Component.

As stated in Chapter 3, GLOMAR's design is platform independent. To illustrate this requires a system that targets a number of platforms. Unfortunately, the number of supported operating systems and hardware platforms available for .NET is limited. For example, currently .NET provides production quality support for Windows (98, ME, NT4, 2000, XP) only. However, there are other platforms were support is available. These include Linux (The Mono-project (Ximian 2002)), FreeBSD, Mac OS X 10.2 (Rotor (Whittington 2002)) and the Pocket PC/WinCE (The Compact Framework (Microsoft 2002b)). The current implementation of GLOMAR is built on the Windows platform only. However, porting to other non-Windows platforms is trivial, as much of the implementation of GLOMAR is platform independent and because all the non-Windows implementation of .NET comply with the European Computer Manufacturer's Association (ECMA) Common Language Infrastructure (CLI) standard (ECMA 2001). A Java implementation was considered early in

the project based on the multiple platform support. However, it was not chosen because of the lack of multiple language support.

6.2 Relationship Component Implementation

This section details the proposed and developed interfaces, as well as the base class that make up the implementation of the Relationship Component. Also discussed is how a Relationship Component is built, including how the Consistency Model and Relationship Scope are developed and how the Clone List is set and utilised. The unique interfaces and base class as defined within GLOMAR are exclusively for the implementation of concurrency control and consistency maintenance within the multiple consistency model approach.

6.2.1 *IConsistencyModel* interface

The implementation of the *IConsistencyModel* interface (figure 6.1) facilitates the creation of a Consistency Model. This interface consists of 20 methods, each relating to an operation (both local and remote), as well as a start-up and a shutdown method.

Implemented in each method is the code responsible for handling specific operations and/or events associated with concurrency control and consistency maintenance. For example, the *Read* method encapsulates all the concurrency control and consistency maintenance functionality built for handling a *read* operation.

For each method defined within the *IConsistencyModel* interface, a *FHANDLE* structure parameter is passed. The purpose of the *FHANDLE* structure is to package information about:

- **CloneName.** An object representing the file being accessed.
- **RCinfo.** The details of the Relationship Component housing this Consistency Model.
- **Source.** Whether the operation is local or remote.
- **AppID.** The process ID of the application generating the operation.
- **Tag.** A string containing any additional information passed in by an application. All members (excluding the *tag* member) are filled by GLOMAR.

```
Public Interface IConsistencyModel

    Function OpenCM(ByVal handle As FHANDLE) As OperationStatus
    Function CleanEndCM(ByVal handle As FHANDLE) As OperationStatus

    Function Create(ByVal handle As FHANDLE) As OperationStatus
    Function Open(ByVal handle As FHANDLE) As OperationStatus
    Function Close(ByVal handle As FHANDLE) As OperationStatus
    Function Read(ByVal handle As FHANDLE, ByVal length As Long,
                  ByVal offset As Long, ByRef buffer As Long)
                  As OperationStatus
    Function Write(ByVal handle As FHANDLE, ByVal length As Long,
                   ByVal offset As Long, ByVal buffer As Long)
                   As OperationStatus
    Function GetAttr(ByVal handle As FHANDLE, ByRef fileattr As FILE_ATTR)
                   As OperationStatus
    Function SetAttr(ByVal handle As FHANDLE, ByVal fileattr As FILE_ATTR)
                   As OperationStatus

    Function Rename(ByVal handle As FHANDLE) As OperationStatus
    Function Remove(ByVal handle As FHANDLE) As OperationStatus
    Function RemoteOpen(ByVal handle As FHANDLE) As OperationStatus
    Function RemoteCreate(ByVal handle As FHANDLE) As OperationStatus
    Function RemoteClose(ByVal handle As FHANDLE) As OperationStatus
    Function RemoteRead(ByVal handle As FHANDLE, ByVal length As Long,
                        ByVal offset As Long, ByRef buffer As Long)

                        As OperationStatus
    Function RemoteWrite(ByVal handle As FHANDLE, ByVal length As Long,
                         ByVal offset As Long, ByVal buffer As Long)
                         As OperationStatus
    Function RemoteGetAttr(ByVal handle As FHANDLE, ByRef fileattr As FILE_ATTR)
                         As OperationStatus
    Function RemoteSetAttr(ByVal handle As FHANDLE, ByVal fileattr As FILE_ATTR)
                         As OperationStatus
    Function RemoteRename(ByVal handle As FHANDLE) As OperationStatus
    Function RemoteRemove(ByVal handle As FHANDLE) As OperationStatus

End Interface
```

Figure 6.1: *IConsistencyModel* Interface

The return value from all the methods defined within the *IConsistencyModel* interface is an enumerated value called *OperationStatus*. This value is used to indicate the success or failure of an operation. The state of the *OperationStatus* value can be:

- **FAILED.** The operation has failed.
- **SUCCEEDED.** The operation has succeeded.
- **COMPLETED.** The operation has completed remotely, but not required to complete locally.
- **OPERATIONFAILED.** The operation failed, but can be replayed within the Executive.

For the methods *Rcad*, *Write*, *RemoteRead* and *RemoteWrite*, an additional three parameters are passed. These parameters include *length*, *offset* and *buffer*. This complies with the standard approach used to read and write to buffers (passing by reference for reads and by value for writes) and is commonly used within file system implementations (Huston and Honeyman 1993). Each of these additional parameters is a 64-bit integer (.NET long).

For methods responsible for handling file attribute requests (*SetAttr*, *GetAttr*, *RemoteSetAttr* and *RemoteGetAttr*), a *FILE_ATTR* structure parameter is passed. This structure contains:

- **LENGTH.** The length of a file.
- **LASTWRITE.** The date/time of the last write.
- **LASTACCESS.** The date/time of the last access.
- **CREATIONTIME.** The creation date/time of the file.

For consistency, each member of the structure is a 64-bit Integer. As a result, to pass an attribute requires the specific information be marshalled into a 64-bit value. For example, if creation time was being passed, it must be converted to a 64-bit representation of the time to fit within the *FILE_ATTR* structure.

6.2.2 *IRelationshipScope* interface

The scenario for which a Relationship Component is valid is defined by implementing the *IRelationshipScope* interface (figure 6.2). This interface has four methods *EvaluateRules*, *SystemProfileRule*, *UserProfileRule*, *FileProfileRule*. The top level method (*EvaluateRules*) is responsible for determining the validity of a Relationship Component. The other three methods (*SystemProfileRule*, *UserProfileRule*, *FileProfileRule*) are responsible for evaluating different aspects of the current scenario for the purpose of supporting the top level method. These three supporting methods are based on the taxonomy defined in section 5.2.5.

Creating a Relationship Scope requires implementing scenario defining rules within the three supporting methods and implementing a mechanism within the top level method, to derive an overall decision on the validity of the Relationship Component.

```
Public Interface IRelationshipScope

    Function EvaluateRules (ByVal spinfo As Environment)
                            As Boolean
    Function SystemProfileRule (ByVal spinfo As Environment.SystemProfileInfo)
                             As Boolean
    Function UserProfileRule(ByVal upinfo As Environment.UserProfileInfo)
                            As Boolean
    Function FileProfileRule(ByVal fpinfo As Environment.FileProfileInfo)
                            As Boolean
End Interface
```

Figure 6.2: *IRelationshipScope* Interface

The unique aspect of the Relationship Scope is that each method is passed a specialised shared data structure (section 5.2.5) containing the status information of the system, for the explicit purpose of helping the scenario defining process. This shared data structure (referred to as the *Environment* object) contains three hashtables. These hashtables include:

- **UserProfileInfo.** This hashtable contains all the scenario information referring to the current user.
- **FileProfileInfo.** This hashtable contains all the scenario information referring to the current file being accessed.
- **SystemProfileInfo.** This hashtable contains all the scenario information referring to the hardware, software and location of the current system.

Accessing specific information within the shared data structure requires an appropriate key for the appropriate hashtable. This key is a string that defines a particular environmental variable. For example, to access the current bandwidth metric, the key "hardware.network.bandwidth" would be used. This key gives access to the actual value, which can be used to determine the validity of a Relationship Component. Section 6.3.5 details the *Environment* object, in particular how it is used by the System Grader.

The return type from each method is a Boolean type. Thus, if *true* is returned, the scenario for this Relationship Component is valid. If *false* is returned, the scenario for this Relationship Component is invalid.

6.2.3 *RelationshipComponent* class

The foundation of any Relationship Component is the *RelationshipComponent* class. All Relationship Components are built on this base class. The *RelationshipComponent* class is an abstract base class, meaning that it can never exist as an independent instance unless it is inherited.

The *RelationshipComponent* class not only includes the functionality to house the Relationship Scope and Consistency Model implementations, but also the functionality associated with Clone List management and setting up Relationship Component metadata.

Clone List Management

The Clone List is separated from the actual Relationship Component to allow easy modification. This means that the Clone List is actually a XML file containing the relationships between particular files and Relationship Components. Figure 6.3 shows an extract of one such file and table 6.1 details each XML tag.

```
<CloneList>
  <RelationshipComponent name="Outlook Connection RC"
                         id="OutlookConnectionRC">

    <Clone localpath="c:\temp\outlook.pst"
           name="outlook file" id="outlook" type="file" />
  </RelationshipComponent>
  <RelationshipComponent name="Connection" id="444444">
    <Clone localpath="C:\temp\test1.txt"
           name="test1" id="test1" type="file" />
  </RelationshipComponent>
</CloneList>
```

Figure 6.3: Clone List XML File

The structure of the Clone List contains a parent tag that defines a Relationship Component. Child nodes under the parent tag represent files or encapsulated types for which this Relationship Component can govern. The resulting XML file is thus a description of the Clone-to-Relationship Component relationship, which can be modified independently (via any text editor) of the Relationship Component.

The Clone List is used by an already implemented method supplied with the *RelationshipComponent* base class. Its purpose is to determine if a particular

Clone Type	Tag	Description
Relationship Component	`<RelationshipComponent name="Tester1" id="Tester1"/>`	This tag encapsulates all Clone types. It directly references an existing Relationship Component via the ID.
Volume	`<Clone name="PIM Volume" id="vol1" type="volume" />`	This tag is used to bind both directories and files, and is used solely for ordering and Clone bundling
Directory	`<Clone localpath="c:\temp" name="Temp" id="temp" type="directory" filter="*.mpg" Recursive="false" />`	This tag is used to reference files within a directory. The filter defines the extension of files that should be included. The recursive attributes indicates weather to include all sub directories or not (*true* or *false*).
File	`<Clone localpath="c:\temp\test.txt" name="test.txt" id="test21" type="file" />`	This tag represents the actual file.

Table 6.1: Clone List Tags

Relationship Component can govern a particular Clone. The *IsSuitable* method is called by the Relationship Component Repository during the *Parade* process (section 5.2.6) on each Relationship Component. The Clone ID of a particular Clone is passed in. The Clone List is then searched with the results of the search informing the Relationship Component Repository if the particular Relationship Component can govern the particular Clone. Thus, all the functionality associated with managing the Clone List is implemented within the *RelationshipComponent* base class.

Relationship Component Metadata

The process of creating a Relationship Component not only consists of creating and setting up the three major sub-components (the Consistency Model, Relationship Scope and Clone List), but also setting up the metadata to describe the Relationship Component itself. This is performed by forcing the constructor of the base class to accept an object containing all the appropriate metadata for the Relationship Component. The *RCInformation* object encapsulates all the details describing the Relationship Component. These details include:

- **Name.** A literal string of the name of the Relationship Component.
- **Author.** Name of the author of the Relationship Component.
- **ID.** A literal string of a unique identifier for the Relationship Component.
- **InstanceType.** An enumerated type that describes the two instance types an Relationship Component can be (section 4.4.1).
- **ConcurrencyState.** An enumerated type that describes the three threading models that a Relationship Component can be (section 4.4.2).
- **Shutdowntime.** An integer value that describes how long the *CleanEndCM* method is allowed to run in milliseconds (section 4.4.3).

The creation of the Consistency Model and Relationship Scope classes, coupled with setting up the Clone List and metadata information results in a fully functional Relationship Component. This can be added to into the GLOMAR middleware layer, ready to be used when required.

6.3 GLOMAR Middleware Layer Implementation

6.3.1 Local Operation Interface

The implementation of the Local Operation Interface consists of three modules, including a filter driver (transparent) and two bridges (non-transparent) for COM (Box 1998) and .NET (figure 6.4).

Filter Driver

The transparent module used in the GLOMAR implementation is a Windows NT/2000 filter driver (Nagar 1997), that intercepts file operations from the

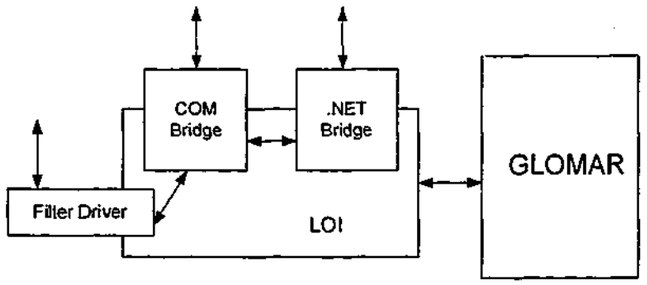

Figure 6.4: Local Operation Interface Implementation

input/output subsystem. These operations are then forwarded to the COM bridge, which in turn forwards them on to the Executive (via the .NET bridge). Once the Executive finishes with these operations, they are returned, with the filter driver re-entering the operations back into the input/output subsystem. The filter driver prototype demonstrated the feasibility of the transparent module. However, for the sake of the GLOMAR implementation, attention was focused upon the non-transparent modules.

COM Bridge

The purpose of the COM bridge is to allow non .NET applications (and the filter driver) to pass operations to the Local Operation Interface (and Remote Operation Interface). The COM bridge consists of a .NET component, which exposes four methods[3] (*LocalOperation32*, *RemoteOperation32*, *LocalOperation64* and *RemoteOperation64*) via COM. In addition, the COM bridge exposed a number of specially designed classes that can be used by COM compliant applications. These methods process parameters passed in and make appropriate calls to the .NET bridge, via the .NET remoting infrastructure (Srinivasan 2001).

.NET Bridge

.NET applications (and the COM bridge) communicate with GLOMAR using the .NET bridge (figure 6.5). The .NET bridge interface consists of a single

[3]32 bit and 64 bit interfaces are required because some COM compliant languages (e.g. VB6) do not natively support 64 bit integers

method that receives eight parameters and returns the *OperationStatus* enumerated type. The eight passed in parameters include:

- **filename.** The name of the file being access.
- **application.** The application ID.
- **operation.** The operation being performed (this is a string representation, e.g. "read") .
- **length.** The length of the data within the buffer.
- **file_attr.** The file attribute details (by reference).
- **offset.** The offset within the buffer.
- **buffer.** The actual buffer (by reference).
- **tag.** The optional tag parameter (only used when required).

```
Public Function LocalOperation(ByVal filename As String, _
                               ByVal application As String, _
                               ByVal operation As String, _
                               ByVal length As Long, _
                               ByRef file_attr As FILE_ATTR, _
                               ByVal offset As Long, _
                               ByRef buffer As Long, _
                               Optional ByVal tag As String = "")
                                    As OperationStatus
```

Figure 6.5: Local Operation Interface Entry Point

To communicate with the .NET bridge, an application must use the .NET remoting infrastructure. The URI to this particular remoted service is *tcp://localhost:9999/loi*. The .NET bridge functionality consists of processing the operation passed to it and then forwarding it onto the Executive. This call to the Executive is synchronous.

6.3.2 Remote Operation Interface

The Remote Operation Interface supplies an entry point into GLOMAR via the .NET remoting infrastructure (the URI is *tcp://hostname:9999/roi*). The Remote Operation Interface differs from the Local Operation Interface due to

its ability to accept two additional parameters. These additional parameters define the ID of the Relationship Component (*rcid*) that made the request and the ID of the Clone (*cloneid*) that the operation is intended. Figure 6.6 details the Remote Operation Interface implementation interface.

```
Public Function RemoteOperation(ByVal rcid As String, _
                                ByVal cloneid As String, _
                                ByVal appid As String, _
                                ByRef buffer As Long, _
                                ByRef attr As FILE_ATTR, _
                                ByVal length As Long, _
                                ByVal offset As Long, _
                                ByVal fileoperation As FileOperation, _
                                Optional ByVal tag As String = "") _
                                    As OperationStatus
```

Figure 6.6: Remote Operation Interface Entry Point

Since the *rcid* and *cloneid* are passed in as parameters, there is no need to enter the remote operation into the Executive, as this information makes the Executive's evaluation process redundant. As a result, the Remote Operation Interface directly invokes the remote interface of the specific Relationship Component.

The implication of bypassing the Executive is that an operation is only able to use the *singleton* instance of a Relationship Component (as only the Executive exploits a Relationship Component's instantiation model). Thus, sharing state information between remote operations is unsafe, as other remote operations (from other processes) might access the same instance.

6.3.3 Clone Distribution Manager

The Clone Distribution Manager processes the Clone List and provides an API for querying this information. This service is used by other GLOMAR middleware layer services and Relationship Components.

During initiation of the Clone Distribution Manager, the location of the Clone List is passed in. This XML file is opened and read, with an in-memory Clone List (XML DOM) created. Some additional processing of the Clone List is done, prior to making it available via the Clone Distribution Manager's API. This processing includes:

- **Expanding Directories.** All Clone tags specified as directories are read (the volume tags are ignored, as they are an encapsulating type with no direct reference to Clones). Based on the *filter* and *recursive* attributes, a list of files is returned. For example, if **.txt* is defined as the filter, then the list of files for this directory would include all *.txt* files. For each returned file, a unique ID is created. This ID is prefixed with the ID of the directory, thus directly binding the file to any event that affects the directory.

- **Determine Clone Reference Type.** Determining the reference type for each Clone depends on the source of its reference. Thus, each file entry within the Clone List is extended to include a property indicating its reference type (*explicitly* or *implicitly*).

The Clone Distribution Manager's API facilitates the querying of this in-memory Clone List (table 6.2). It is implemented as a singleton (Gamma et al. 1995), with it exposed via the .NET remoting infrastructure. The URI of this remoted service is *tcp://localhost:9999/clonemanager*. A benefit of using the .NET remoting infrastructure is that since the service is available via TCP/IP, it can be used by remote nodes.

6.3.4 Service Manager

The implementation of the Service Manager provides the ability to implement any additional functionality external to a Relationship Component. This is done by the Service Manager reading an XML file (Service Manager XML File; figure 6.7) containing the physical location and instantiation type (a *remoted*, an *instantiated* and an *external* service) of all services. Once read, the services are dynamically instantiated and started by using the .NET *Reflection* library.

All services (excluding the external services) are required to implement the *IGlomarService* interface (figure 6.8). This interface consists of a *StartService* method and a *StopService* method. The motivation for this interface is to allow any service to be created. The Service Manager starts all services by asynchronously invoking the *StartService* method of each implementation. When GLOMAR is stopped, the Service Manager stops all services by invoking the *StopService* method.

To add a new service to the Service Manager XML file requires adding a service node. The attributes of this node are:

Interface	*Parameters*	*Return Type*	*Description*
GetCloneList	Relationship Component ID	Array of Clone Details	This method returns a list of all Clones that are Relationship Component govern
GetCloneDetails	Clone ID	Clone Details	This method gets the Clone Details from the Clone ID
GetCloneDetails	Clone ID and Relationship Component ID	Clone Details	This method gets the Clone Details from the Clone ID and the Relationship Component ID
ClonenameToCloneID	Clonename	Clone ID	This method gets the Clone ID from the Clone Name
IdToLocation	Clone ID	Clone Local Path	This method gets the Clone Local path from the Clone ID
IdToClonename	Clone ID	Clone Name	This method gets the Clone name from the Clone ID

Table 6.2: Clone Distribution Manager's API


```
<services>
   <service name="transaction_log" source="trans.dll"
            classname="glomar.transaction"
            remoted="true" />
   <service name="updater" source="update.dll"
            classname="glomar.update"
            remoted="false" />
   <service name="proprogater" source="prop.exe"/>
</services>
```

Figure 6.7: Service Manager XML File

```
Public Interface IGlomarService
    Public Sub StartService()
    Pubic Sub StopService()
End Interface
```

Figure 6.8: *IGlomarService* interface

- **name.** The name of the service.
- **source.** This attribute is the physical location of the service, which is relative to the Service Manager directory.
- **classname.** This attribute is the class name of the service. This attribute should include any namespaces this class exists within.
- **remoted.** This attribute is a Boolean value, indicating whether the service is to be remoted on the GLOMAR communication channel or not. If *true* is set, the resulting URI for this service is *tcp://localhost:9999/<name>*. If *false* is set, the service starts, but is not added to the GLOMAR communication channel.

As GLOMAR aims to support legacy systems, there is a need to support external functionality not directly coupled to the concurrency control and consistency maintenance mechanism. These might take the form of stand-alone executables. To allow for such an implementation (non .NET object) within GLOMAR, the System Grader allocates and runs the executable within a new process. The Service Manager determines this is an executable by the *classname* attribute being set to null. Else, all other services run within GLOMAR's process.

6.3.5 System Grader

The System Grader implementation is responsible for two tasks: constructing the *Environment* object and running the process to update the contents of the *Environment* object.

Constructing the Environment Object

The unique approach to dynamically describing a scenario is achieved by the *Environment* object defining three hashtables, namely, *UserProfileInfo*, *FileProfileInfo* and *SystemProfileInfo*. In other words, these are the in-memory representation of the information used in defining the context of a Relationship Component. Each profiles directly maps to each element of the predefined context defining taxonomy.

Upon the start-up of the System Grader, these hashtables are initiated to store the environmental information defined within the System Grader XML File (figure 6.9). To define the environmental information available via the hashtables and detailing how they are to be updated is done by adding nodes to the System Grader XML File. This ability allows new and unique mechanisms to be added to GLOMAR to describe existing and future scenarios. This in collaboration with Relationship Components provides the capabilities to not only describe scenarios, but also apply concurrency control and consistency maintenance mechanisms to them.

UserProfileInfo

To insert a new user, a node is added to the *resource* tag defined as *user*. The *user* node consists of four attributes, including:

- **username.** The username must be unique.
- **name.** For display purposes only.
- **type.** This attribute reflects the user type. The classification of the user type can be user-defined and has no direct linkage to the user types defined within an operating system. For example, a user can be "Admin", "Workgroup", etc. This means the classification of user types can be more detailed than what might be offered by an operating system.
- **operation.** This attribute reflects the user role. For example, a user might be defined as a "Power User", thus have a higher level of privilege than a "Passive User". This classification of user role is user-defined and is highly dependent on the specific constraints, but independent of an operating system.


```
<resource type="System">
  <hardware>
    <pref name="processor" >

       <pref name="speed" source="SysInfo.dll"
            method="getCpuSpeed" object="ManagedSysInfo"
            poll="1000"/>
    </pref>
    <pref name="storage" value="2000000"/>
    <pref name="network">
       <pref name="connection" source="pMetric.dll"
             method="connection"  object="prefMetric"
             poll="1000"/>
       <pref name="type" value="Wireless"/>
       <pref name="bandwidth" value="234234" />
       <pref name="packet_loss" value="2342342"/>
    </pref>
  </hardware>
  <software>
      <pref name="filesystem" value="NTFS5.0"/>
  </software>
  <location/>
</resource>

<resource type="User">
      <user username="simonc" name="Simon Cuce"
            type="ADMIN" operation="High"/>
</resource>

<resource type="File">
      <file id="test" operationtype="READWRITE"
                      operationaction="stream" />
      <type id="12312" username="simon"
            description="multimedia_simon" extension=".mpg"
            operationtype="READ"
            operationaction="stream" />
</resource>
```

Figure 6.9: System Grader XML File

During start-up of the System Grader, information about the currently logged in user is extracted from the operating system and matched to the user defined within the System Grader XML File. When a match is found, the details are extracted and inserted in the *UserProfileInfo* hashtable. If, however, no match is found, the contents of the *UserProfileInfo* hashtable are set to null. The hashtable key to the actual values are detailed in table 6.3.

FileProfileInfo

The *resource* tag defined as *file* allows for two types of entries. One type defines details about a particular Clone, the other defines details about the Clone's

Entry Attribute	Key
username	username
name	displayname
type	usertype
operation	operation

Table 6.3: *UserProfileInfo* Keys

type. Details relating to a particular Clone are defined by adding a *file* node. The attributes of the *file* node are:

- **id.** This attribute must correspond to the ID of a Clone within, at least, one Clone List.
- **operationtype.** This attribute defines the type of access expected for a Clone. For example, *readonly*, *readwrite* or *update*. This attribute can be extended to create access types more intuitive to the specific needs of the system.
- **operationaction.** This attribute defines how a Clone is used by an application. For example, a Clone is read as a *stream* (as is the case with multimedia files). This attribute can be extended also to suit a particular need.

The second type of entry that can be added is the *type* node. This defines similar attributes as the *file* node, however defines additional details relating to the Clone type. For example, all .jpg files (which are image files) can have a single set of *operationtype* and *operationaction* attributes assigned to them.

In addition, the *type* node allows a *username* attribute to be added. This allows particular users to specify *type* nodes relating to Clone types. For example, *user1* uses *.txt* files differently from *user2*. The *username* attribute allows this to be expressed, as different details relating to a Clone type can be assigned to different users. This feature allows additional flexibility for describing details relating to Clones or Clone types. The attributes of the *type* node are:

- **id.** This attribute must be unique.
- **username.** This attribute is used to couple a particular type with a particular user. This is a non-mandatory attribute.

Entry Attribute	Key
File Entry	
id	filename (converted internal)
operationtype	operationtype
operationaction	operationaction
Type Entry	
id	*value not available*
username	*value not available*
description	description
operationaction	ext.operationaction
operationtype	ext.operationtype
extension	*not available*

Table 6.4: *FileProfileInfo* Keys

- **description.** This attribute is a description of the type.
- **extension.** This attribute contains the file extension. In many file systems, the file extension is used to detail the type of a file.
- **operationtype.** This attribute defines the type of access expected for a Clone. For example, *readonly*, *readwrite* or *update*. This attribute can be extended to create access types more intuitive to the specific needs of the system.
- **operationaction.** This attribute defines how an application uses a Clone. For example, a Clone is read like a *stream* (as is the case with multimedia files). This attribute can be extended also to suit a particular need.

When updating the contents of the *FileProfileInfo* hashtable, there is no need to have every different Clone and Clone type available. Rather, a Relationship Scope is only concerned with the Clone and Clone type that is being accessed. For this reason, only the details that refer to the current Clone being accessed are stored. This means a "pickup" is done on a per operation basis within the Executive (section 6.3.7), updating the *FileProfileInfo* hashtable to reflect the current Clone. Table 6.4 details keys for accessing Clone and Clone type information.

SystemProfileInfo

The *resource* tag defined as *system* is responsible for containing all the environmental information relating to the *hardware*, *software* and *location* of the

system. However, unlike the *file* and *user* nodes, the structure of the *system* node is not predefined. Rather, static environmental information, parental tags and Context Providers can be added into the *system* node as needed.

Defining environmental information consists of adding a *pref* node to the *hardware*, the *software* or the *location* nodes. The *pref* tag can have three states: a reference to static environmental information, a parental tag or a reference to a Context Provider. The simplest of these tags is a reference to static environmental information. Such environmental information remains constant for the duration of a GLOMAR session, for example, the version of an operating system. The attributes of the *pref* tag to define static environmental information are:

- **name.** The name of the static environmental information.
- **value.** The value of the static environmental information.

GLOMAR facilitates easy classification of different environmental information by allowing parental tags to be defined for the *system* node. This allows different environmental information to be encapsulated within a single node. An example of a parental tag can be found in figure 6.9. The attribute of the *pref* tag to define a parental tag is:

- **name.** The name of the parental tag.

Since Context Providers are external modules that generate environmental information independent of GLOMAR. The *pref* tag must contain enough information so that GLOMAR can create an instance of the module (via the .NET *Reflection* library) and extract the resulting data. Thus, the attributes of the *pref* tag to define a Context Provider are:

- **name.** The name of the Context Provider.
- **source.** This attribute references the actual file containing the Context Provider. Only a relative address is required, as by default, all Context Providers reside within a *providers* directory.
- **object.** This attribute references the actual object within the file. This value should include any namespaces that this object exists within.
- **method.** This attribute defines the method to be invoked. The results from this method call will be the environmental information.

Provider Name	Key
Processor Speed	hardware.processor.speed
Hardware Storage	hardware.storage
Network Connection	handware.network.connection
Network Type	hardware.network.type
Network Bandwidth	hardware.network.bandwidth
Network Packet Loss	hardware.network.packet_loss
File System	software.filesystem

Table 6.5: *SystemProfileInfo* Keys based on figure 6.9

- **poll.** This attribute defines when this Context Provider is invoked. If the value is 0, the method will be called once, with the resulting value remaining static for the duration of a GLOMAR session. If, however, the poll value is greater than 0, this then defines in milliseconds the frequency for which this method should be invoked.

Environmental information is addressed within the *SystemProfileInfo* hashtable by flattening the *system* node XML structure and separating nodes with full stops. For example, in figure 6.9, the Processor Speed Context Provider would have a key "hardware.processor.speed". This addressing approach allows for any structural variation within the System Grader XML file, regardless of the depth and defined structure of the node. Table 6.5 details the keys generated as a result of processing the System Grader XML File illustrated in figure 6.9.

Updating Values

The other activity of the System Grader is to constantly update values within the *Environment* object with information retrieved from the Context Providers. During the initiation of the System Grader, all Context Providers are passed onto an additional subsystem that manages their invocation.

Each Context Provider is allocated a thread (independent of the Executive's thread) within which a delegate (Watkins, Hammond, and Abrams 2003) is assigned. These delegates point to defined methods for each Context Provider. When the threads are started, an infinite loop is initiated, calling the delegate. Once the call is made and the results from the invocation are added to the *Environment* object, the thread is made to sleep for the duration defined within the poll attribute. This process continues for the duration of a GLOMAR session.

6.3.6 Relationship Component Repository

The purpose of the Relationship Component Repository is to manage Relationship Components, determine the validity of a Relationship Component and supply the Executive with Relationship Component instances. The Relationship Component Repository does this by dividing processing into three distinct areas:

- Relationship Component Initiation
- Relationship Component Processing
- Relationship Component Implementation

Relationship Component Initiation

All Relationship Components are initiated when the Relationship Component Repository is started. This process involves a number of steps, exploiting the dynamic invocation functionality of the .NET *Reflection* library. These include:

- **Finding all Relationship Components.** The initial step is to inspect the Relationship Component directory for all *.dll* files. This results in a list of files being returned.
- **Select only .NET files.** The next step determines which files are .NET compliant components. The reason for this step is that many component implementations share the *.dll* extension (e.g. COM and Dynamically Link Libraries).
- **Finding the Relationship Components.** Once the list of files has been reduced to only .NET components, then each is searched to find the classes that are inherited from the *RelationshipComponent* class. This involves a recursive process that traverses the inheritance hierarchy of each class stopping when the *RelationshipComponent* class is found. The reason for this rigorous process is to allow for inheritance from existing Relationship Components. As long as the *RelationshipComponent* class is found within the inheritance hierarchy, the class is deemed a Relationship Component.
- **Creating and Storing instances of the Relationship Components.** Once the Relationship Component is found, the .NET *Reflection* library is used to dynamically create a new instance of the Relationship Component. These instances are added to an in-memory storage container and used from that point onwards by the Relationship Component Repository.

Relationship Component Processing

All the complexity defined within section 5.2.6 regarding *Relationship Component Processing* is encapsulated into a single method. For the Executive to request a valid Relationship Component, requires calling the *Evaluate* method, which takes in a single parameter (defining the current operation and scenario) and returns a string (defining the "most valid" Relationship Component). The parameter passed in is called the *CloneOperationDetails* and contains the following members:

- **AppId.** This is the current application ID of the originator of the operation.
- **Status.** This refers to the *Environment* object, with the *UserProfileInfo*, *FileProfileInfo* *SystemProfileInfo* hashtables filled by the Executive.
- **Clone.** This object represents the Clone targeted by this operation.
- **Operation.** This enumerated value represents the operation being performed.
- **AssignmentList.** This object contains a list of existing Relationship Component instances that currently govern this Clone. In most cases, there will only be a single entry, as usually only a single Relationship Component will govern a Clone at anytime.

Upon receiving the *CloneOperationDetails* parameter, the *Evaluate* method firstly ensures that the existing Relationship Component is still valid. This requires extracting the currently implemented Relationship Component for the Clone from the *AssignmentList*. The Relationship Component currently being evaluated is found within this list. If no Relationship Component is found in this list, then the *Parade* method is called. This returns a Relationship Component ID, which is then passed back to the Executive.

If, however, a Relationship Component is found within the *AssignmentList*, its validity is firstly checked. The method *EvaluateRules* which is a member of the Relationship Scope is called (section 6.2.2), passing in the current *Environment*

object. If the result from this method is *true*, then the component is still valid for this scenario. This results in the string "current" being returned to the Executive.

If the Relationship Component is invalid, as indicated by a *false* being returned, the Relationship Component Repository calls the *Parade* method. The result from this method is an ID of the most appropriate Relationship Component based on the current scenario. This Relationship Component is evaluated to determine if the current threading model and instantiation type defined are compatible for its implementation. A failure of this process results in the string "failed" being returned. Success prompts the return of the Relationship Component ID to the Executive.

The Parade Method

The implementation of the *Parade* method (section 5.2.6) determines which Relationship Component should govern a specific Clone. This is done by calling the *IsSuitable* method for each Relationship Component. The successful results from this query are added to a list of suitable Relationship Components. Depending upon the count of this list:

- **Count equal to zero.** The string "failed" is returned to the Executive.
- **Count equal to one.** An actual Relationship Component ID is returned to the Executive.
- **Count greater than one.** A list of Relationship Components is passed onto the *Investigation* method.

The Investigation Method

The functionality of the *Investigation* method firstly involves receiving a list of Relationship Components from the *Parade* method. The list of suitable Relationship Components is reduced by querying the *EvaluateRules* method of each Relationship Component's Relationship Scope. This determines which of the resulting Relationship Components can exist within the current scenario.

The count of the resulting list is tested. If the count is zero, the Relationship Component Repository returns "failed" to the Executive. If the count is equal to one, then the Relationship Component ID is returned to the Executive. If, however, the count is greater than one, the list of suitable Relationship Components is passed to the *AdvancedInvestigation* method, which attempts to find the most suitable.

The purpose of the *AdvancedInvestigation* is to determine the priority of the selected Relationship Components based on the referencing of a particular Clone. Each Clone can be referenced explicitly, meaning it is singularly referenced to a particular Relationship Component or implicitly, meaning it is part of an aggregate collection of references to a particular Relationship Component. Two lists are created, one for explicitly referenced Clones and one for implicitly referenced Clones. Each suitable Relationship Component is evaluated and assigned to the appropriate list. When complete, the first item in the explicit list is returned. If, however, the count of the explicit list is zero, then the first item in the implicit list is returned.

Relationship Component Implementation

When the Executive needs to invoke a Relationship Component, the Relationship Component Repository is used. It supplies two methods to return a Relationship Component instance, one for a *new instance* and one for a *singleton* Relationship Component. The approach employed is similar to a factory pattern (Gamma et al. 1995).

If the Executive needs a new instance of a particular Relationship Component, it calls the *GetRCinRCID* method. This method takes the ID of the required Relationship Component and returns a new instance of that component. Before a new instance of the component is returned, the Relationship Component Repository's list of stored components is searched. When found, the properties are queried to determine if this Relationship Component can exist as a *new instance*. If so, a new instance is dynamically created using the .NET *Reflection* library. This instance is then returned to the Executive.

If, however, a *singleton* instance of the Relationship Component is needed, the Executive calls the *GetRCinRCIDStatic* method. The Relationship Component Repository's list of stored Relationship Components is searched using the Relationship Component ID. When found, a reference from the Relationship Component Repository's list is returned to the Executive.

6.3.7 Executive

The primary purpose of the Executive is to manage file operations from the Local Operation Interface and coordinate this with the Relationship Component Repository, so that it can dispatch operations to appropriate Relationship

Components. This is performed within the Executive by the method *FileSystemEvent*. As this method only receives primitive information from the Local Operation Interface, its first task is to refine this information in preparation for its usage.

Initially, the absolute address of the target Clone is resolved via the Clone Distribution Manager. If this fails, then it is assumed that this Clone is not meant for governing by GLOMAR. This happens early within the *FileSystemEvent* method to ensure that operations not destined for GLOMAR incur minimal cost. If a reference is found within the Clone List, then the details for that Clone are retrieved.

Next, a list of existing Relationship Components (in relation to the Clone) is retrieved. All relationships between Clones and implemented Relationship Components are recorded within an *AssignmentList*. This *AssignmentList* contains:

- **ApplicationID.** This is one of the index values.
- **CloneID.** This is the other index value.
- **RelationshipComponent.** This is a collection of references to Relationship Component instances that are currently governing this application and Clone.

Following this, the *FileProfileInfo* hashtable within the *Environment* object is set. As stated in section 5.2.5, file information is set on a per operation basis. This requires making a call to the System Grader and passing in the Clone ID. The details stored within the System Grader XML File are examined and the appropriate information is set within the *Environment* object.

The final stage involves invoking the *Evaluate* method of the Relationship Component Repository. As stated (section 6.3.6), the results can either be, "current", "failed", or a Relationship Component ID. If "failed" is returned, then the operation is aborted. If "current" is returned, then the operation is allowed to continue using the locally stored Relationship Component instance fetched from the *AssignmentList*.

However, if a Relationship Component ID is returned (indicating a change in governing Relationship Components), then a series of steps to manage the lifecycle of the existing Relationship Component are implemented. The first of these steps is to shutdown the existing Relationship Component. The implementation of this is done by asynchronously calling the *CleanEndCM* method of the existing Relationship Component. While this is occurring, the current

thread is made to sleep for the duration defined by the shutdown time within the metadata of the Relationship Component. When the thread is awoken, it checks whether the asynchronous call to the *CleanEndCM* method has completed. If so, the Executive continues. If however the thread is still busy, because the method has yet to complete, then the thread is terminated, regardless of the state of the *CleanEndCM* method. The Relationship Component is consequently removed, with control returned to the Executive.

The next step is to retrieve the new Relationship Component instance from the Relationship Component Repository and invoke the *OpenCM* method on the Consistency Model object. This is performed prior to any other operation and is only done when a Relationship Component has been replaced. The final step is to update the *AssignmentList* of the changes.

Once the processing of the Executive is complete, the operation can be directed to the appropriate method within the Relationship Component. This involves entering the operation into an error handling loop (section 5.2.7). The results from the method determine if this operation is to be replayed or not. This loop is however only performed once, regardless of the number of failed results. Once complete, the results from the method are returned to the Local Operation Interface.

6.3.8 Administration Console

The Administration Console (figure 6.10) serves two purposes, to view and to edit elements of the actual GLOMAR implementation via a graphic user interface. It includes the ability to view the details of all the invoked Relationship Components, all the Context Providers and Clones. It also supports the creation of new Clones (including Volumes, Directories and Files), defining their relationship with each other, assigning these Clones to Relationship Components (via a drag-and-drop) and updating GLOMAR's settings (including port number and directories).

6.4 Running the GLOMAR System

There exists two versions of GLOMAR, the Console Application (figure 6.11) and the Windows Service Application (figure 6.12). The purpose of the Console Application is to provide a debuggable version of the implementation, specifically suited to development of Relationship Components. The Windows Service

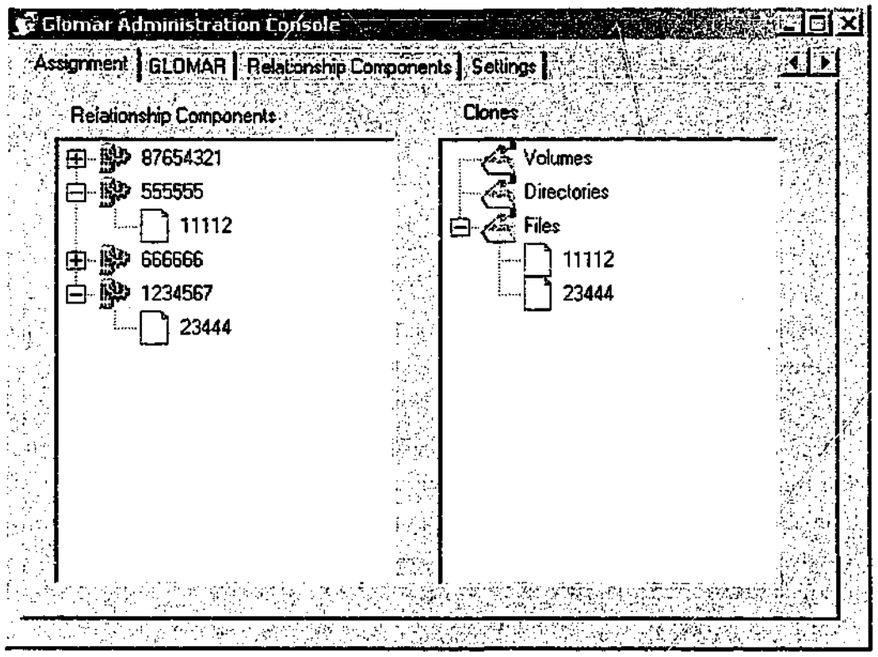

Figure 6.10: GLOMAR Administration Console

Application on the other hand allows for a transparently running implementation of the GLOMAR system, suited to production implementations.

The actual implementation for both vetsions is the same (some additional code has been added to enable it to run as a Windows Service and it was not compiled in debug mode). The structure consists of instantiating all middleware components, registering a channel with the .NET remoting infrastructure (by default port 9999) and assigning the Remote Operation Interface, Local Operation Interface and Clone Distribution Manager to that channel. Also all middleware components are installed into the Global Assembly Cache (GAC) (Macdonald 2001). This is due to the necessity of the COM bridge to be registered in a universally accessible location.

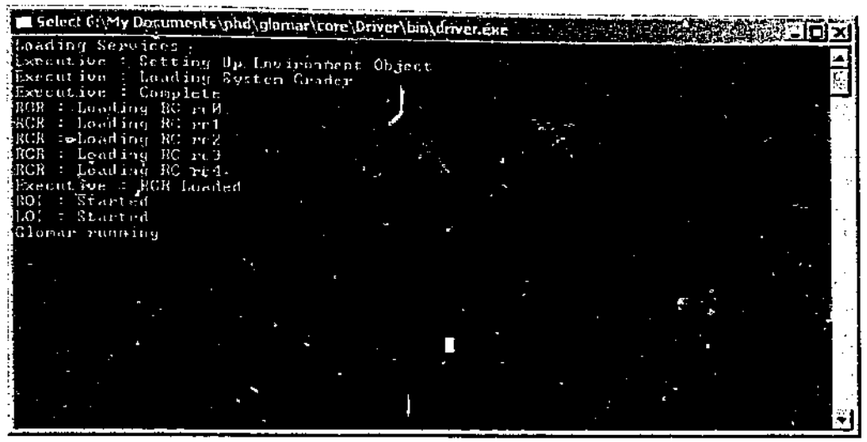

Figure 6.11: GLOMAR's Console Driver

6.5 Summary

This chapter has detailed the actual implementation of GLOMAR, discussing the implementation of the Relationship Component and the GLOMAR middleware layer. This implementation was written in .NET using a mixture of languages and techniques. The approach targeted the Windows platform, but was designed so that much of the underlying techniques are platform independent. However, where platform specific elements were used, they were kept to a minimum.

The next chapter details three Relationship Component implementations. Each example illustrate through implementation the flexibility of GLOMAR. The first implementation is an approach to handling disconnected operations. The second is a full scale implementation of an existing concurrency control mechanism targeted at a mobile environment. The third and final implementation provides fine grain consistency maintenance for a Personal Information Manager.

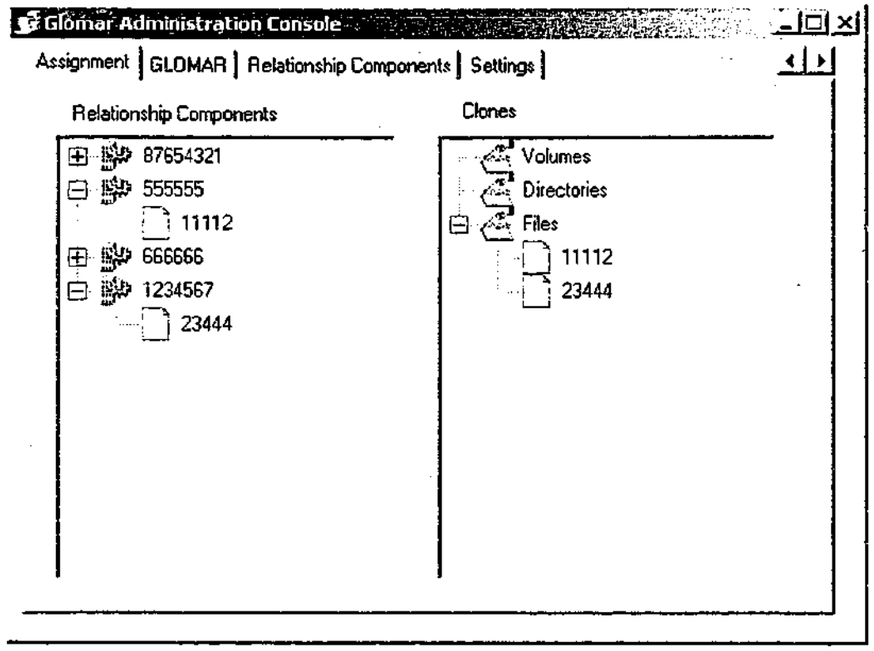

Figure 6.10: GLOMAR Administration Console

Application on the other hand allows for a transparently running implementation of the GLOMAR system, suited to production implementations.

The actual implementation for both versions is the same (some additional code has been added to enable it to run as a Windows Service and it was not compiled in debug mode). The structure consists of instantiating all middleware components, registering a channel with the .NET remoting infrastructure (by default port 9999) and assigning the Remote Operation Interface, Local Operation Interface and Clone Distribution Manager to that channel. Also all middleware components are installed into the Global Assembly Cache (GAC) (Macdonald 2001). This is due to the necessity of the COM bridge to be registered in a universally accessible location.

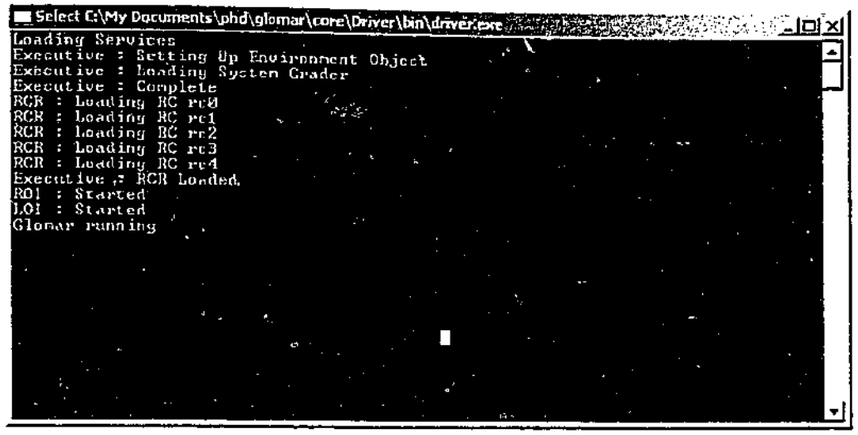

Figure 6.11: GLOMAR's Console Driver

6.5 Summary

This chapter has detailed the actual implementation of GLOMAR, discussing the implementation of the Relationship Component and the GLOMAR middleware layer. This implementation was written in .NET using a mixture of languages and techniques. The approach targeted the Windows platform, but was designed so that much of the underlying techniques are platform independent. However, where platform specific elements were used, they were kept to a minimum.

The next chapter details three Relationship Component implementations. Each example illustrate through implementation the flexibility of GLOMAR. The first implementation is an approach to handling disconnected operations. The second is a full scale implementation of an existing concurrency control mechanism targeted at a mobile environment. The third and final implementation provides fine grain consistency maintenance for a Personal Information Manager.

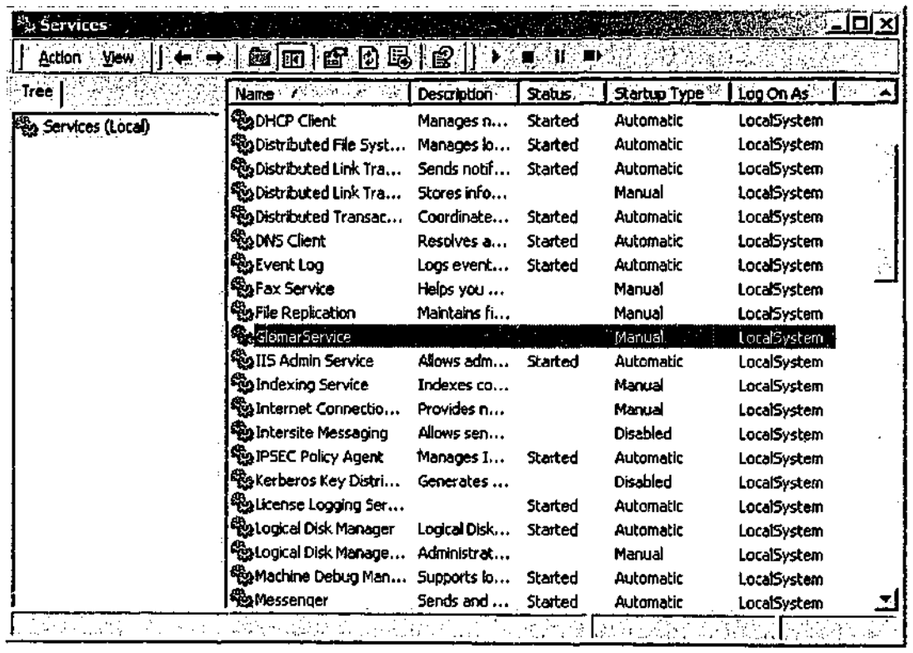

Figure 6.12: GLOMAR's Windows Service

Chapter 7

Case Studies

GLOMAR's major contribution is the ability to support flexibility, by allowing multiple consistency maintenance and concurrency control mechanisms to be concurrently invoked at run-time. This chapter demonstrates this flexibility via three different implemented Relationship Components (case studies). The first implementation (notepad application) demonstrates a simplified consistency maintenance mechanism that handles disconnected operations. The second implementation (the Twin Transaction Model (Rasheed 1999)) demonstrates a more complex model for handling mobility in a transaction based processing system. The final implementation (Outlook 2002) demonstrates a Relationship Component that handles the maintenance of data for a Personal Information Manager (PIM).

7.1 Aims

The chapter aims to:

- Demonstrate the creation process for a consistency maintenance and concurrency control mechanism.
- Demonstrate flexible and application-specific solutions within Relationship Components.
- Demonstrate effective support for heterogeneity issues within a DFS.

Evaluating effective support for flexibility and heterogeneity is usually associated with a criterion that has no uniform basis, that may introduce bias and lacks quantitative metrics. Therefore, analysing and observing Relationship

Component implementations for different classes of scenarios are considered sufficient in this dissertation. As a result, the ability to service different concurrency and consistency needs of a device, application and user are illustrated within this chapter through actual Relationship Component implementations.

7.2 Notepad Relationship Component

The first implementation of a Relationship Component manages connected and disconnected operations for a text editor. In other words, this implementation supports a seamless transition from a connected to disconnected state for a notepad like application, preventing lost updates upon distributed replicas.

The main aim is to illustrate how more than one Relationship Component can be used to govern a single Clone. Within this implementation, the two connectivity states are divided into two external Relationship Components, one for connected and another for disconnected.

7.2.1 Notepad Relationship Components Design

Two approaches were designed to illustrate different methods of handling disconnected operations. These include an optimistic approach (*Get Latest*) and a pessimistic approach (*Read Once, Write All*). Since much of the design and implementation of this case study is restricted, it primarily demonstrates the feasibility of GLOMAR in handling consistency maintenance and concurrency control within a DFS.

Get Latest

The *Get Latest* approach has two modes, *disconnected* and *connected*. When disconnected, all operations are performed locally, regardless of their consistency requirements. When connected, prior to a file being opened, the timestamp of all connected nodes is checked. Whichever node has the latest version of the file (based on a timestamp) is considered the most up-to-date. If the local node contains the most up-to-date file, then it is used. However, if a remote node contains the most up-to-date file, then it is copied over the network (figure 7.1) and made available locally.

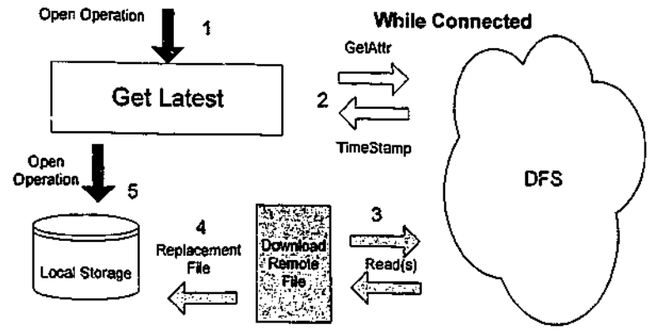

Figure 7.1: Notepad *Get Latest* Relationship Component

Read Once, Write All

The *Read Once, Write All* (ROWA, section 2.3.1) approach focuses on the handling of disconnected operations in a pessimistic manner. Again the model is split into two modes, *connected* and *disconnected*. When disconnected, rather than allowing operations to proceed unchecked, all modification (*write*) operations are written to a persistent log. Within each entry, the details of the operation are included (length, buffer and offset).

Upon reconnection, the contents of the log are replayed prior to allowing normal operations to resume. As each logged operation is successfully performed remotely, it is removed from the log. This allows a disconnection to occur during log play back, without the risk of losing logged operations. Normal modification operations that occur while connected must be propagated to all remote nodes prior to being committed locally. Figure 7.2 details the ROWA design which is similar to Coda's approach (Kistler and Satyanarayanan 1991), though less sophisticated.

7.2.2 Notepad Relationship Components Implementation

The implementation of these two approaches (*Read Once, Write All* and *Get Latest*), requires:

- The implementation of the notepad application.
- The creation of the Context Providers to determine connectivity.
- The creation of the Relationship Scope for each Relationship Component.

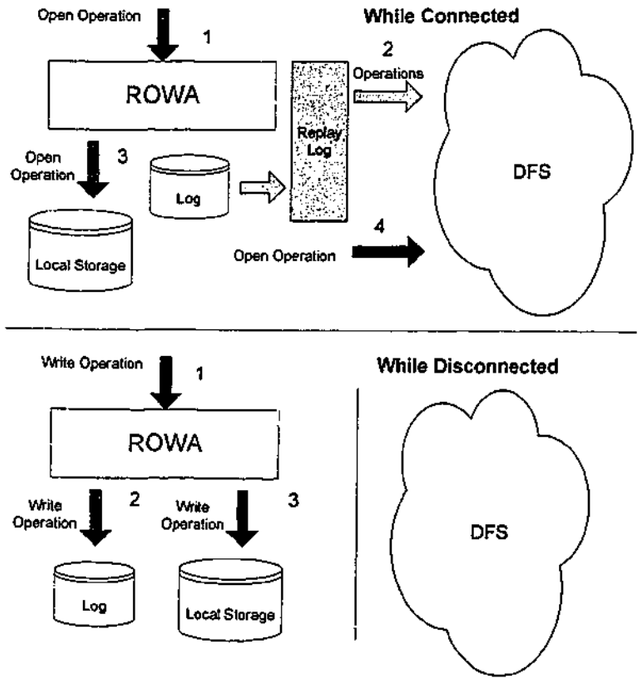

Figure 7.2: Notepad *ROWA* Relationship Component

- The implementation of an XML Web Service used to communicate information between GLOMAR nodes.
- The implementation of the Consistency Models for both the *Get Latest* and *ROWA*.

Notepad Application

The notepad application supports the opening, editing and closing of ASCII plain text files (figure 7.3). However, the unique element of this application is associated with how operations are managed.

Traditional text editor applications follow session semantics (Coulouris, Dollimore, and Kindberg 2001) to persist data. The developed notepad application exhibits a more interactive approach. For example, existing text editors adjust an in-memory representation of the data, only saving the changes to disk when all modifications are finished. Within the developed notepad application,

keystroke events are captured (extracting the keyboard item and its location within text string), then immediately saved on disk. Using this level of granularity means the application exhibits the behaviours of a traditional file system (frequent and unstructured operations), making it a suitable test bed.

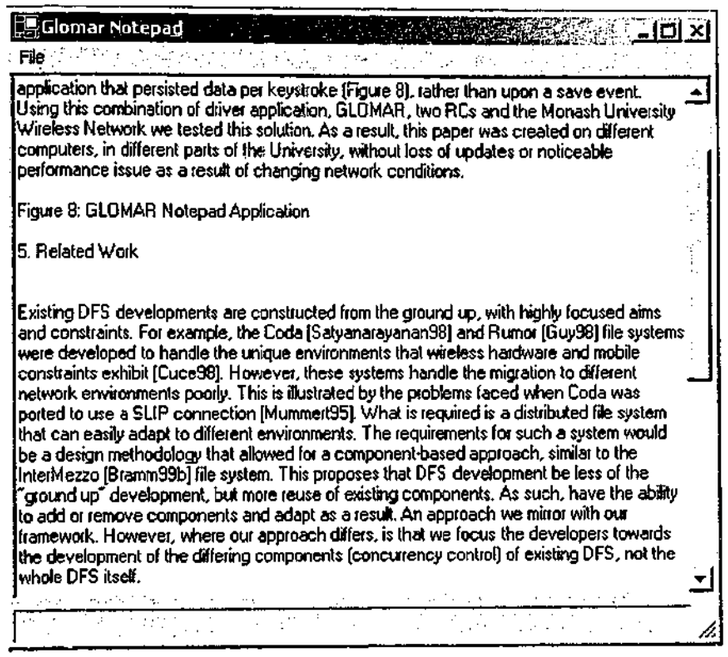

Figure 7.3: Notepad Application

The other unique aspect of the notepad application is that all operations are dispatched to GLOMAR. Thus, all operations including *open*, *close*, *write* and *read* are redirected to the Local Operation Interface's .NET bridge (section 6.3.1). Depending upon the results from GLOMAR, the operation will or will not be completed.

Context Provider and Relationship Scope

The implementation of the Context Provider consists of a single method that derives the number of packets per second sent from remote nodes. This is done by calling the Windows Performance Counter (Microsoft 2000). If the current result from the counter is zero, then connection is assumed lost. However, if

the result is greater than zero, then connection is assumed active. The Context Provider extracts this information into a form that can be used by the Relationship Scope. Thus, the results from this Context Provider can be either, *"connected"* or *"disconnected"*. See figure 7.4 for the source code of the Relationship Scope.

```
Public Class No_Connection_RelationshipScope

    Implements IRelationshipScope

    Public Function EvaluateRules(ByVal info As Environment)
            As Boolean Implements IRelationshipScope.EvaluateRules
        Return SystemProfileRule(info.m_SystemProfileInfo)
    End Function

    Public Function SystemProfileRule(ByVal spinfo As Environment.SystemProfileInfo)
             As Boolean Implements IRelationshipScope.SystemProfileRule
        If spinfo.Item("hardware.network.connection") = "disconnected" Then
            Return True
        Else
            Return False
        End If
    End Function
End Class

Public Class Connection_RelationshipScope
    Implements IRelationshipScope

    Public Function EvaluateRules(ByVal info As Environment)
            As Boolean Implements IRelationshipScope.EvaluateRules
        Return SystemProfileRule(info.m_SystemProfileInfo)
    End Function

    Public Function SystemProfileRule(ByVal spinfo As Environment.SystemProfileInfo)
             As Boolean Implements IRelationshipScope.SystemProfileRule
        If spinfo.Item("hardware.network.connection") = "connected" Then
            Return True
        Else
            Return False
        End If
    End Function
End Class
```

Figure 7.4: Notepad Relationship Scopes

XML Web Service

To facilitate the communication between GLOMAR nodes, an XML Web Service (Caron 2002) was created (figure 7.5). This XML Web Service exposes the Remote Operation Interface (section 6.3.2), using SOAP envelopes over HTTP. For a Relationship Component to use this XML Web Service, a proxy class is required.

The motivation for using a XML Web Service to facilitate communication was to illustrate the flexibility and capability of Relationship Component implementations to house different and unique implementations. In addition, it offered an elegant approach, whereby much of the details of communication infrastructure were hidden from the developer.

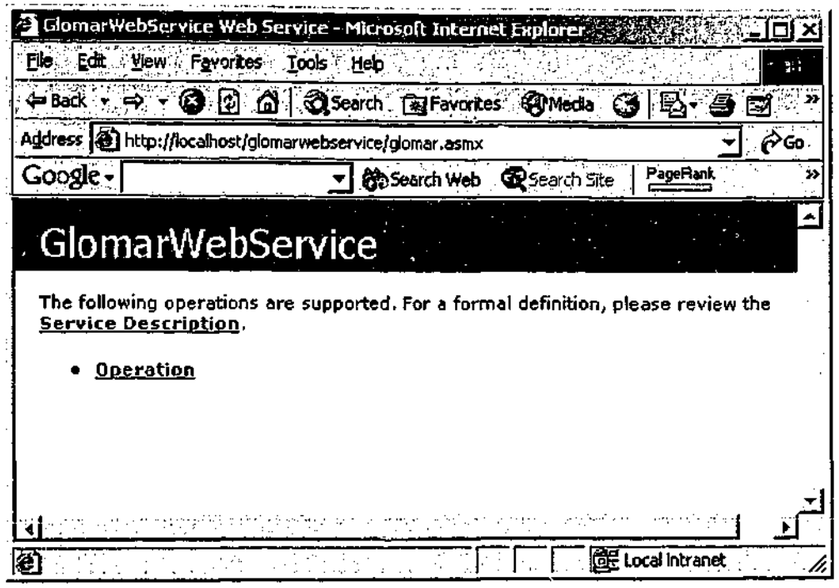

Figure 7.5: GLOMAR's XML Web Service

Get Latest Relationship Component Implementation

With the *Get Latest* implementation, all the functionality is contained within the Consistency Model responsible for connected operations. The reason the Consistency Model responsible for disconnected operations has no implementation is that no additional value adding processing is required when disconnected.

Within the Consistency Model responsible for connected operations, the bulk of the complexity is found within the *open* method. The implementation of this method consists of firstly calling the *GetAttr* method on the remote node. The *RemoteGetAttr* method queries the file system to derive the timestamp of the "last write access". Once the timestamp of the remote version of the file is retrieved, then a comparison is performed with the local version's timestamp.

If the local version's timestamp indicates it is the most up-to-date, then the *open* method is finished and control returned to the system.

However, if the remote version's timestamp is the most up-to-date, then the *open* method downloads the contents of the remote version of the file. This is done by calling the *Read* method of the remote node (in other words *RemoteRead* method). The previously called *GetAttr* method not only retrieves the timestamp, but also the length of the file (the number of ASCII characters within the file). Thus, the download process consists of multiple calls to the *RemoteRead* method. With each call, a single character is downloaded. The position of this character within the file is passed as the offset value. The results from each subsequent request are incorporated in a temporary file. Once all characters of the file have been successfully downloaded, then the newly created file replaces the local version. When complete, control is returned to the system.

ROWA Relationship Component Implementation

The implementation of the *ROWA* approach consists of capturing all modification operations within a log while disconnected, then posting the contents of the log and subsequent operations while connected. The implementation of the Consistency Model responsible for disconnected operations intercepts all *write* operations, saving them to disk. This is done by writing a 192-bit value (64-bit x 3) that stores the length, offset and buffer values. Each entry is added to the tail of the file, implementing a queue structure.

When the scenario changes and the Consistency Model responsible for connected operations is invoked, GLOMAR firstly invokes the Consistency Model's *OpenCM* method. Within this method, operations captured within the log are replayed. This involves posting these operations to the remote node via GLOMAR's XML Web Service. This method loops through the log, reading the length, offset and buffer of each operation. On the remote node, these operations are processed by the *RemoteWrite* method. The result from the *RemoteWrite* method is returned to the *OpenCM* method. If successful, the posted operation is removed from the log. The benefit of this approach is that even if a disconnection occurs during a log replay, the stored operations are not lost. When the log has been fully replayed (the queue is empty), the original operation that invoked GLOMAR is performed. While connected all *open*, *close* and *write* operations are posted via GLOMAR's XML Web Service for remote

processing. Only success on the remote node will result in the operation being committed locally.

7.2.3 Analysis of Notepad Relationship Components

The aim of this implementation was to create a number of Relationship Components that mask the issues associated with disconnected operations, such that operations on Clones were not lost due to a loss of connectivity. The results of this implementation demonstrated this, as operations were not lost due to disconnections.

Get Latest

In particular the *Get Latest* Relationship Component allows for a number of plain text file (in this case Latex files) to be shared between a desktop and laptop. As the usage pattern was such that only a single user was ever editing a file at one particular time, there were no version conflict issues. However, this Relationship Component did allow the user to move seamlessly between processing units, ensuring the latest copy was available.

This approach did have some issues resulting from the lack of sophistication and the reliance upon the operating system's clock for determining a file's validity. For example, in one particular episode, the user was unaware that the laptop's clock had been turned back. As a result (and as expected), modifications were lost during the synchronisation stage, as data that was more timely was not assigned the appropriate timestamp.

ROWA

The results from the *ROWA* Relationship Component illustrated an approach that handled the constraints of a disconnection environment well. *ROWA's* pessimistic nature meant scaling was improved compared to that of the *Get Latest*. This was due in part to the reduced likelihood of concurrent operations causing conflicts. However, when conflicts did arise, the *ROWA* implementation poorly managed conflict resolution.

There were additional limitations relating to some of the implementation choices made. The usage of a 64-bit structure to house a single character meant that communication was inefficient. This inturn then affected the responsiveness of the notepad application.

The results from both Relationship Component implementations illustrated the aims of GLOMAR, namely, the handling of a semantically similar approach to a file system and the flexibility to implement a unique solution for consistency maintenance and concurrency control.

One of the benefits of this implementation is the idea that normal applications could be converted into distributed applications with little modification. Within this example, the notepad application was designed to act much like a normal text editor (in stand-alone mode). However, with only minimal code to integrate it into GLOMAR, it became distributed. This can be directly attributed to the decoupling of the concurrency control and consistency maintenance mechanism from the application and/or operating system.

The reason for this can draw parallels with the beneficial qualities of object-oriented middlewares in the creation and implementation of a distributed system. As the concurrency control functionality was decoupled into separated components, managed by the GLOMAR middleware layer, the notepad application was only required to interact with GLOMAR via a predefined interface, to become a distributed application. This demarcation and delegation of functionality meant that to achieve the look and feel of a distributed application only required creating the linkage between GLOMAR and the notepad application. As shown, this linkage between GLOMAR and the notepad application was trivial and relatively seamless.

This case study demonstrated two simple solutions for providing a suitable level of consistency within a DFS that handles disconnected operations. While the feasibility to handle this situation was illustrated, it was recognised that to demonstrate fully the true potential of GLOMAR required a more complex and sophisticated system. The next case study demonstrates such an implementation.

7.3 Twin Transaction Model Relationship Component

To enhance support for handling the constraints of a mobile environment (beyond the previously demonstrated case study) and demonstrating a sophisticated Relationship Component, the *Twin Transaction Model (TTM)* (Rasheed 1999; Cuce, Zaslavsky, Hu, and Rambhia 2002) Relationship Component was developed. This implementation illustrates two of GLOMAR's aims, flexibility and the ability to port existing consistency models to the GLOMAR framework.

The TTM implementation offers a means of demonstrating the flexibility of a Relationship Component and evaluating the process of porting a system that was never initially intended for GLOMAR.

Another motivation for implementing TTM, was that much of the design is based upon transaction semantics within a Distributed Database Management System. TTM implementation within GLOMAR offers a suitable case study to demonstrate how transaction semantics are implemented within a DFS. This factor is important, as many concurrency control and consistency maintenance mechanism are based on transaction semantics and are designed for Distributed Database Management System environments.

7.3.1 TTM Relationship Component Design

The TTM defines transaction execution mechanisms to cater for connected and disconnected modes of operation (Rasheed 1999). A defined resynchronisation mechanism achieves a consistent state on reconnection of a mobile host. The TTM consists of transaction executions/management, concurrency control and resynchronisation parts. These different parts work together to maintain the consistency of the local (a mobile host) and global (all mobile hosts) system states. A brief summary of the TTM follows.

Transaction Execution/Management

TTM implements transaction management using a set of *mobile transaction managers (MTMs)*, as well as a *fixed transaction manager (FTM)*. The MTMs are responsible for handling transaction requests on each of the mobile hosts, while FTM handles requests from hosts on the fixed network and from MTMs. A set of *global reconciliation algorithms* is used between MTMs and FTM to detect any conflicts/inconsistencies that may arise and resolve them.

Unlike classical transactions, TTM has no simple correctness criterion. Each MTM decides what is its appropriate behaviour (serialisable or not). The *correctness* of execution of transactions is relative to these individual behaviours. The consistency of FTM (and MTMs) is tentative most of the time (unless all the MTMs have connected and have gone through the resynchronisation process).

TTM relies on resynchronisation process and transaction conflict resolvers to keep data consistent. The resynchronisation process is necessary to maintain the consistency of data. The local state of MTM and local state of FTM evolves

along their own courses (different transactions take them to different states from an initial consistent state). The resynchronisation process is responsible for the combination of the two local states and making the data consistent. To achieve this goal, the resynchronisation process needs a record of transactions that are executed on both MTM and FTM. This record is termed as *Transaction History Log*. Apart from transaction history log, the transactions in *pending* state on MTM and FTM are also needed to ensure that no inconsistent data access was allowed.

The resynchronisation process is executed whenever an MTM connects with FTM and information exchange takes place (pending transactions and their transaction execution history). On re-connection, the resynchronisation process will make sure that replicated data items on MTM reflect the updates on the FTM. The resynchronisation process can be divided into the following tasks:

- Propagation of resolved transactions from FTM to MTM, which transit transactions from *tentative-commit* to *commit* state.
- Propagation of pending transactions from MTM to FTM for resolution.

TTM defines a twinning process, which is applied to each transaction. The application of the twinning process to a transaction creates two transactions, called twin-transactions. Thus, for a transaction T, two twin transactions Tα and Tβ are created (thus the name Twin Transaction Model). The twinning process is applied in both connected and disconnected modes of operation. The α-twin of the transaction is executed locally and β-twin is executed on FTM. Detection and resolution of conflicts is performed using both twins of a transaction. The twinning of a transaction is implicit and is performed for each transaction. Explicit twinning of a transaction is also allowed, where the application can submit the two twin transactions (α and β). This is not implemented in the Relationship Component implementation.

The two twins of a transaction undergo a number of states before reaching their final state. The transition of states is dependent on connected or disconnected mode of operation. These states and their valid transitions in connected and disconnected modes are depicted in figure 7.6.

The twin transaction execution is different in connected and disconnected mode. In the disconnected mode:

- Twinning process generates two transactions Tα and Tβ.

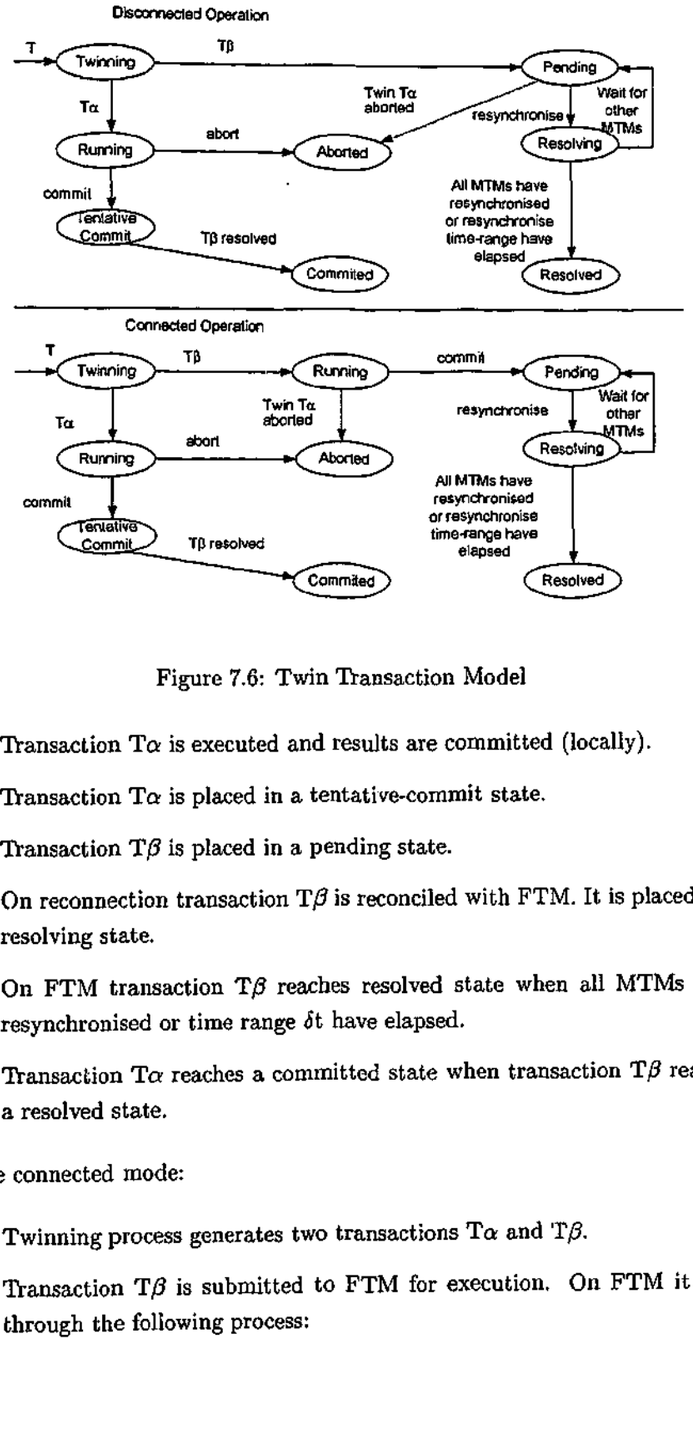

Figure 7.6: Twin Transaction Model

- Transaction Tα is executed and results are committed (locally).
- Transaction Tα is placed in a tentative-commit state.
- Transaction Tβ is placed in a pending state.
- On reconnection transaction Tβ is reconciled with FTM. It is placed in a resolving state.
- On FTM transaction Tβ reaches resolved state when all MTMs have resynchronised or time range δt have elapsed.
- Transaction Tα reaches a committed state when transaction Tβ reaches a resolved state.

In the connected mode:

- Twinning process generates two transactions Tα and Tβ.
- Transaction Tβ is submitted to FTM for execution. On FTM it goes through the following process:

  - Twinning process generates α and β twins of transaction Tα. The MTM that submits transaction Tβ explicitly defines α and β twins to be semantically equivalent to α and β twins of original transaction.
  - The α-twin of transaction Tβ is executed. It is placed in a tentative-commit state and results are committed on FTM.
  - The β-twin of transaction Tβ is placed in a pending state.
  - Transaction Tβ reflects the state of it's β-twin.
  - On FTM β-twin of transaction Tβ reaches a resolved state when all MTMs have resynchronised or time-range δt has elapsed

- Transaction Tα is executed and results are committed (locally). Transaction Tα is placed in a tentative-commit state.
- Transaction Tα reaches a committed state when transaction Tβ reaches a resolved or committed state.

Any subsequent transactions that operate on results produced by a pending transaction T will become dependent on the success of transaction T. There can be multiple pending transactions creating possible dependencies among themselves. This inter-dependency of transactions (twin-transactions) must be captured to eliminate any inconsistent operations/transactions. This will include *read* and *write* dependencies. The *read* dependency occurs when a transaction T' reads a data item written by a pending transaction T and write dependency occurs when a transaction T' writes a data item written by a pending transaction.

A transaction T' reaches its resolved state when all the transactions on which transaction T is dependent have reached their resolved states. Since, MTMs are operating in disconnected mode, the transaction T cannot reach its resolved state unless all the MTMs have connected with FTM and have gone through the resynchronisation process. This is necessary to ensure that every transaction that was executed during disconnection has been taken into account. However, it is possible that some MTMs will not connect. In such a case, the FTM will always be in a tentative state and none of the transactions will reach their resolved state. A time out period for each MTM is required to avoid waiting for connection indefinitely. If an MTM connects after the time out period, the transactions executed on that MTM are resynchronised after all the resolved transactions in the FTM are complete.

Consistency Model

TTM has relaxed the ACID (Haerder and Reuter 1983) properties of a transaction for the following reasons:

- Transactions are kept in a *pending* state.
- The results produced by a transaction in a *pending* state are made visible to other transactions.
- The transactions produce results that are *tentative* until the transactions on which these results are dependent are reconciled/synchronised.
- A transaction in a *pending* state is resynchronised and reaches its resolved state during the resynchronisation process. The transaction might be aborted or compensated during this process.

The consistency of data items is guaranteed if a transaction transforms a data item from one consistent state to another consistent state. In the TTM, transactions undergo different states before reaching their final resolved state. Among these states, the states *pending* and *resolved* are important states to consistency management. Post the *pending* state, the results of the α-twin are made visible locally and after the *resolved* state, the results are made visible globally. Thus, consistency of a data object must be maintained locally and globally. That is, transactions executed on an MTM must be consistent before they are checked against global consistency (if they are not consistent locally, they will not be consistent globally). Thus to achieve consistency, TTM employs two layers of consistency, *local* and *global*.

The local consistency layer requires that α-twin of a twin transaction must be locally serialisable with other pending or ongoing transactions executed on the same unit. Locally serialisable ensures that the local interleaved transaction executions are always equivalent to the serial execution. This is necessary for the pen ling transactions to present a locally consistent view of their results.

Since each manager executes transactions locally and these transactions are resolved at a later stage, an optimistic concurrency control mechanism to maintain consistency is the natural choice. The global consistency model is defined by the resynchronisation process, which defines how transaction histories are to be merged.

The TTM defines two levels of global consistency models, that is two types of resynchronisation mechanisms (two ways to detect conflicts during merging of histories).

In the TTM, it is left to the application to decide what level of consistency is required. The two levels are:

- **Global One-Copy Serialisability.** If a disconnected transaction T's result is copied to the server as is, T must be 1SR (One-Copy Serialisable) with all previously committed transactions.
- **Global Certification.** Global certificates require that if a disconnected transaction T's result is copied to the server as is, T must be not only serialisable with but also serialisable after all the previously committed transactions.

The classical durability property requires that once a transaction completes successfully, its results must be able to survive. This also implies that once the results of a transaction are made available (to other transactions) it must remain a permanent part of the system state until modified by later transactions. In the TTM, durability of transactions is only guaranteed when they have been resolved (classified as a successful completion of a transaction). This is because a committed transaction (in pending state) on a manager might be aborted/compensated during the resynchronisation process.

Transaction History

Each mobile host executes transactions in connected and disconnected modes. These transactions need to be synchronised with all transactions executed by all other hosts to achieve a global consistent state. For that purpose, each twin-transaction manager creates a transaction history log. The transaction history log is used to synchronise transactions during the resynchronisation of the transaction from an MTM to FTM. The conflict detection (consistency validation checks) cannot be performed if the transaction system does not record the history of the disconnected transaction executions (transaction history). Certain strategies are adopted to keep the size of the log to a minimum.

Concurrency Control

TTM uses an optimistic concurrency control mechanism. It allows transactions to proceed, with conflicts being detected at a later stage. TTM assumes that

β-twin will be successfully executed on FTM and the results produced by both twin transactions (α and β) will be the same.

Figure 7.6 shows that the commit of β-twin is dependent on that of α-twin and the transaction reaches the resolved state after the resynchronisation process. Although, the transaction is executed in connected mode, the resynchronisation process still needs to be executed and the transaction will go through the pending state before reaching the resolved state. This is due to the fact, that there are transactions that might be executed on other disconnected TTMs, which need to be resynchronised with transactions executed in the connected mode.

The advantages of having twins of a transaction in a running state during the connected mode are:

- The communication between the two TTMs is minimised.
- Only the results are compared and if different the results produced by α-twin are discarded and results produced by β-twin are replicated to MTM.
- If MTM that submitted β-twin of a transaction disconnects while in the middle of a transaction, FTM can safely execute β-twin and the result can be resynchronised when that MTM connects again.
- Thus MTM does not need to wait for the completion of a transaction.

The TTM uses replication to provide a private workspace for the execution of the α-twin. The public space is kept on FTM where all β-twins are committed and the results and the transaction history logs are maintained. The execution of a transaction and its validation on FTM and MTM is done in two phases.

Phase 1:

- Initially the two twins Tα and Tβ are executed on MTM and FTM respectively.
- Transaction Tβ on FTM goes through the twinning process. It reaches pending state when its α-twin is in a tentative-commit state. The results produced by its α-twin are checked against the public space for conflicts.
- The results produced by transaction Tβ in pending state are sent back to MTM.

- MTM checks the results against that of Tα. If conflicts are detected, the results produced by Tα are discarded and received results are tentatively committed.
- If Tα is aborted on MTM, Tβ is also aborted on FTM.

Phase 2: Once the transaction is validated by optimistic concurrency control in phase 1, the next phase of validation, which is required by the TTM, begins:

- The results committed on FTM can only be finalised when all other disconnected MTMs have resynchronised with FTM or resynchronise time range (δt) for all TTMs have elapsed.
- Due to this reason, the results that are committed by MTM, even in connected mode, are kept in a pending state.

When a transaction fails validation during resynchronisation, although it was committed and was validated by optimistic concurrency control on FTM (during connected mode), the class of transaction decides the outcome of such failures. This situation is triggered by resynchronisation process of MTMs that are connecting with FTM.

The transactions executing on different disconnected clients cannot read results produced by other clients (during disconnected or connected modes). This necessitates that even the results produced by connected clients must also be kept in a pending state (for a specified amount of time, or until all clients have resynchronised).

The optimistic concurrency control mechanism takes care of concurrency among transactions executed by different managers. A second level of concurrency control mechanism is required to enforce a local consistency model. This is required to make sure that concurrent execution of α-twins of transactions on a manager is one-copy serialisable. These transactions must be synchronised so that the replicated copies of data remain consistent for each of them.

In current implementation strict two-phase locking (2PL) protocol is used for local concurrency control. The advantages of 2PL are:

- Simplicity
- Reasonable performance (when data sharing is infrequent)

Since a single user typically operates each MTM, the likelihood of concurrent transactions operating on the same data (read/write sharing) is low. In addition, 2PL is simple to implement.

The twin-transaction system running on an MTM that is disconnected has a number of responsibilities. The results produced by local transactions are tentative and are dependent on validation by FTM, once reconnected. To achieve a consistent state with FTM, it needs to perform a number of tasks to ensure that enough information about the executed transactions is made available to FTM. These tasks include:

- Maintaining local consistency.
- Recording transaction history information.
- Detecting redundant disconnected transactions.
- Probabilistic Success/Failure calculations.

In disconnected mode of operation, the twin-transaction manager has no knowledge of the state of other managers. In such a situation, an optimistic approach to concurrency control is adopted. The transaction is allowed to execute under the local concurrency control mechanisms and conflict detection is done during the resynchronisation process. To decrease the number of conflicts during the resynchronisation process TTM proposes a probabilistic conflict detection mechanism. The mechanism relies on computing probability of success/failure of a transaction that is being executed in disconnected mode. This is not currently implemented in this version of the Relationship Component.

Resynchronisation - TTM State Propagation

The resynchronisation process is executed whenever an MTM connects with FTM and passes its pending transactions and their execution history log to FTM. The resynchronisation process is executed even if the transaction history log is empty. This is because the local state of MTM and the local state of FTM evolves along their own courses (different transactions takes them to different states from an initial state synchronised at last connection). On re-connection, the resynchronisation process ensures that replicated data items on MTM reflect the updates performed on FTM.

In the synchronising state, new transactions can be executed, but these transactions will be executed in mixed mode (connected/disconnected). If the new transaction is dependent on any of the unresolved transaction, then the transaction is executed in disconnected mode and it will become a pending transaction

to be resolved on the server. On the other hand, if it is not dependent on any of the unresolved transactions, then it is executed in connected mode.

The propagation of TTM state during the resynchronisation process can be divided into the following tasks;

- Propagation of resolved transactions from FTM to MTM to transit transactions from tentative-commit to commit state.
- The propagation of pending transactions from MTM to FTM for resolution.

While an MTM is disconnected from FTM, the state of FTM can change due to two factors, other MTMs connect and synchronises with FTM and connected MTMs execute transactions.

In both these cases, some transactions will transit from pending to resolved state. Some of these transactions might update the data items that are replicated on the disconnected MTM. Due to that reason, when an MTM connects and goes through the resynchronisation process it is imperative that the state of MTM should also be checked against FTM. This is to see if any data item is no longer valid and to check whether some pending transactions on MTM updated any invalid data item. In such a case, the pending transaction (and its siblings) must go through the resynchronisation phase.

The validation of the state of MTM against that of FTM is done in two phases. In the first phase, all the data items that are not accessed by transactions on MTM are regarded as immediately resynchronised with the FTM upon reconnection. Those data items that are accessed by transactions on MTM are considered re-synchronised only after all the transactions have resynchronised on FTM (resulting in either a pending or a resolved state on FTM).

For this validation, each data item is marked with a version (that can contain Twin Transaction Manager Identification that last updated it). The version is changed on each update operation for that data item. On reconnection, the resynchronisation process compares the versions of every replicated data item. If both versions are identical, the data item is marked as valid again, otherwise it is marked as invalid. Any subsequent access to that data item, while the MTM is connected, will cause the manager to refetch the new version of data item from FTM. While the MTM is connected, this validation of state of MTM is performed periodically to ensure the status of data items is up-to-date. Before disconnection or on user demand, the twin transaction implementation provides a mechanism to refetch all invalidated data items.

On reconnection, apart from the need to resynchronise data items that have changed on FTM, the transactions executed on MTM are also propagated to FTM (and their results to other MTMs). This is completed in the second phase of the resynchronisation process when the transaction log of MTM is checked against that of FTM. During this resynchronisation process, conflicts are detected and resolved.

The TTM guarantees a global one-copy serialisable execution schedule. This guarantee requires programmers to resolve conflicts either by programming application-specific resolvers or by manually repairing invalidated transactions. The programmers are also burdened with application-specific conflict estimation agents that are required to calculate the probability of success or failure in the disconnected mode operation under certain circumstances.

Since application-specific resolvers and conflict estimation agents operate on a transaction-by-transaction base, it is wise to carry out the resynchronisation process of MTM transaction log with that of FTM in the same way. This ensures inconsistency and conflict scope are minimised and the resolver has just one invalidated transaction to work with at a time. In addition, the resolver can concentrate on the effects of the transaction on MTM and changes on FTM, during disconnection. Therefore, it is able to make decisions without worrying about interference from other transactions.

The resynchronisation algorithm for a transaction that is in conflict consists of two steps. The first step invokes the resolver, which is either application-specific or automatic re-execution. If no resolver exists, the transaction is aborted. The second step is the successful resolution of a transaction, which often requires adjusting the state of those transactions that read from the resolved transaction (read data written by the resolved transaction).

After the synchronisation process, the FTM has the option to perform a check on all those transactions that are waiting (to transit to a resolved state) for this MTM (last) to connect and perform resynchronisation. Transactions that are in such a state, transit to a resolved state.

7.3.2 TTM Relationship Component Implementation

Two Relationship Components were created to cater for the TTM within GLOMAR, *Connected Twin Transaction* and *Not Connected Twin Transaction*. Each Relationship Component represents a connectivity state that is defined within the TTM approach. The communication mechanism used within this

implementation was GLOMAR's XML Web Service and the test bed application used was the notepad application (section 7.2.2). The architecture of the TTM implementation is shown in figure 7.7.

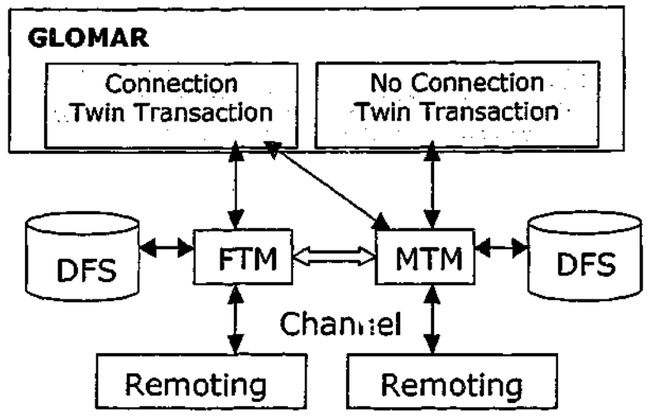

Figure 7.7: Twin Transaction Model Implementation Architecture

The notepad application first calls the *Open* operation of the *Connected Twin Transaction* Relationship Component. Before opening the file, the Relationship Component determines if resynchronisation is required. If any transactions were logged while in a disconnected state, the log is replayed to the server. This process is always the first operation to occur for this Relationship Component, regardless of the operation being invoked.

After the log has been replayed, the *Connection Twin Transaction* Relationship Component then proceeds by invoking the *RemoteOpen* operation on the server. The file is opened on the remote server and a new transaction Tα is created within the Transaction Server (this is an external user-specific service). After this has succeeded, the local cached file is opened and transaction Tβ (which is the same as Tα) is created locally. The states of Tα and Tβ are set as *Running*. Subsequent *Write* operations generated by the notepad application are performed remotely (*RemoteWrite*) and locally, with changes recorded using immutable files (Coulouris, Dollimore, and Kindberg 2001).

When the notepad application has finished with editing a file, it calls the *Close* operation. Since *Open* and *Close* operations are used as surrogate transaction boundaries, this operation triggers the Relationship Component to commit (or abort) all the modification operations and resolve the immutable files. Within

the Relationship Component, this process involves calling the *RemoteClose* operation of the server, where transaction Tα state is changed to *Tentative commit*. If this is successful, the local cached file is closed and the transaction Tβ state is also changed to *Tentative commit*. Once the transactions (Tα and Tβ) are resolved, then they are committed, with each being changed to a *Committed* state.

In the *Not Connected Twin Transaction* Relationship Component, the process is simpler as there is no connectivity. The *Open* operation opens the file and creates a new transaction, which in turn creates a transaction history to record the operations. The transaction is then set to a *Running* state. All modification operations are performed on the immutable file for that transaction and recorded within the log. Finally, the *Close* operation closes the file and changes the transaction's state to *Tentative commit*. The posting of operations occurs when the *Connection Twin Transaction* Relationship Component assumes control.

Beyond the Relationship Components created, additional user-specific services were created to assist in the running of the TTM. These consist of a MTM and FTM, which provide the transaction processing part of the TTM. The major user-specific services provided by MTM and FTM servers are as follows:

- **Add new transaction.** The new transactions are added into the transaction vector on the MTM and FTM. There are two vectors on the MTM corresponding to the connected and the disconnected vector respectively. These transactions are also written into the transaction history file.
- **Change transaction state.** Changing transaction's state on the transaction vector and writing the result into the transaction history. If the state of the transaction is changed into *Abort* and *Commit*, the transaction is removed from the vector.
- **Resynchronisation.** When the MTM connects with FTM from disconnected mode and with transactions in the disconnected transaction vector, the resynchronisation operation should be performed for file system consistency. The transaction conflicts are resolved by comparing timestamps of the transactions. This is done by browsing the transaction vector on the FTM and the transaction history on the MTM and FTM. The transaction's state is changed according to the resolved result. Conflict resolution is only provided by FTM.

- **File manipulating.** There are many situations where files are transferred between MTM and FTM. For example, when a transaction is resolved, it should be propagated to other MTMs. This is done by first reading the file into a file stream and then either uploading or downloading the stream to the remote node.

7.3.3 Analysis of TTM Relationship Component

The results from this implementation primarily focus on demonstrating GLOMAR's flexibility and its ability to house complex concurrency control and consistency maintenance mechanisms. This implementation has demonstrated both of these, as the TTM implements a complex series of steps and exploits the GLOMAR infrastructure, whilst maintaining a sufficient level of consistency within a mobile environment.

This implementation also illustrates that regardless of whether the consistency model was designed for GLOMAR or not, it does not constrain whether it can be implemented. For example, the original design of the TTM was not intended to be built for a Relationship Component.

Another important aim demonstrated from this implementation is the ability to support a pseudo-transaction (section 2.4.1). The implications of being able to support a pseudo-transaction are wide ranging. One implication is that many of the existing consistency models are geared towards using transactions. The TTM implementation has shown that transactions can be effectively accommodated within GLOMAR. The second implication is that transaction semantics can be imposed upon a stateless file system. Rather than being restricted by the actual implementation of the file system, consistency models can be created that use a transactional approach. However, this ability could affect implementation issues. For example, the TTM had to overload certain operations to provide a pseudo-transactional approach.

7.4 Outlook 2002 Relationship Component

To illustrate an application-specific Relationship Component implementation, a consistency model for a Personal Information Manager (PIM) was developed. Rather than creating a home made PIM that lacks complex functionality and is explicitly designed for GLOMAR (and its methodology), Microsoft's Outlook 2002 (XP) (Byrne 2001) was chosen instead. Outlook provides the complexity

and functionality required to illustrate the feasibility of GLOMAR in dealing with application-specific concurrency control and consistency maintenance mechanisms.

The Outlook Relationship Component also illustrates GLOMAR's ability to convert a stand-alone DBM application into a distributed application in a transparent way. In addition, application-specific processing within this Relationship Component further illustrates the flexibility of GLOMAR, as different mechanisms are used throughout the implementation (e.g. client-server, peer-to-peer, optimistic or pessimistic).

The Outlook Relationship Component focuses on maintaining data for Outlook folders. The folders that are governed include, *Sent Mail*, *Draft Mail*, *Inbox*, *Calendar*, *Contacts* and *Tasks*.

The motivation of the consistency models chosen for each of the folders where based on two criteria. Firstly, the appropriateness of the consistency model to the data being governed and secondly, the overall effect of having multiple different consistency models all concurrently existing and operating on Outlook data.

7.4.1 Sent Mail, Draft Mail and Inbox Consistency Model

The three email-related folders within Outlook are governed by the Outlook Relationship Component. These include;

- **Sent Mail.** Contains a copy of all emails sent by an Outlook user.
- **Draft mail.** Contains all emails that have been written, though not sent.
- **Inbox.** Contains all incoming emails.

The three mail folders each use nine data items (table 7.1), referring to the actual email and all other email related details.

The three mail folders are governed with the same type of consistency model, which uses a *preferential remote update approach* [1] within a peer-to-peer environment. When an operation (*update*, *new* and *delete*) occurs, it is added to a queue (persistent store), prior to being performed. An additional process is then invoked periodically to propagate the stored operations to each available node. Only when the operation within the queue has been committed on all

[1] *Preferential* and *mandatory remote update* are our terms used to describe the consistency models used within the Outlook Relationship Component

Data	Description and Purpose
Body	Contains text associated with this email
Subject	Contains the subject line of the email
Time Received	When the email was written
Priority	The priority of the email (In outlook there are levels, HIGH, LOW and NORMAL)
To Address	Contains the email address of the addressee (This can contain more than one email address, however it must be delimited)
From Address	Contains the email address of the sender
CC Address	Contains the email address of carbon copy recipients (This can contain more than one email address, however it must be delimited)
Sent	Whether this email has been sent or not
Unread	Whether this email has been read or not

Table 7.1: Sent Mail, Draft Mail and Inbox Data Items

remote nodes, will the entry within the persistent store be removed. Currently, the propagation of operations for the *Sent Mail* folder is performed every 12 hours, the *Draft Mail* folder every 3 hours and the *Inbox* folder once an hour [2].

7.4.2 Calendar Consistency Model

Within Outlook, the Calendar folder is used to record time-based events for a user. The Calendar information consists of nine data items (table 7.2), referring to the actual appointment, the associated date, time and other details.

The type of consistency model chosen for the Outlook Calendar was a Uniform Majority Quorum Consensus approach (section 2.3.3), implemented within a peer-to-peer environment. This balance between availability (via the peer-to-peer model) and consistency (using quorum consensus) is highly suited to Calendar information.

The crux of this consistency model is determining the dominance of replicated Calendar information, through the use of version vectors (Parker et al. 1983). For example, when necessary, version vectors for each available node (ensuring

[2]The time periods defined for each mail folder are based on the perceived consistency needs based on personal experience.

Data	Description and Purpose
Body	Contains text associated with this event
Subject	Is a shorted description of the event
Location	Displays the location of the event
Time Received	When the appointment was created by the user (this includes date and time)
Start Time	When the event starts (This includes date and time)
End Time	When the event ends (This includes date and time)
Priority	The priority of the event (In outlook there are three levels HIGH, LOW and NORMAL)
All Day Event	Whether the event is all day
Recurring	Whether the event is recurring (Daily, Weekly, Monthly and Yearly)

Table 7.2: Calendar Data Items

that quorum of at least 50% has been achieved) are compared. From this, the dominant replica is found and its Calendar information used.

7.4.3 Contacts Consistency Model

Within Outlook, the Contacts folder in used to store information about individual contacts. The Contact information used within this implementation is detailed in table 7.3 and is only a subset of the available information used by Outlook.

For the maintenance of Contact information, a client-server model was chosen, using both the *mandatory* and *preferential remote update approaches*. When Outlook performs a *new*, an *update* or a *delete* operation, the operation is instantly propagated to a contact list on a server. Failure of the operation on the server side results in the failure of the operation on the client side. Changes (because of client operation) to the contact list on the server are not instantly propagated to all other available replicas. Rather, every 12 hours a process updates each available replica with the contents of the contact list.

Data	Description and Purpose
Body	Contains text associated with a contact
Subject	Is the display name of the Contact
Time Received	The time the Contact was created by a user
Priority	The priority of the Contact (In Outlook there are three levels HIGH, LOW and NORMAL)
Title	Contact's business title
Last name	Surname of Contact
First name	First name of Contact
Middle name	Middle name of Contact
Title Prefix	Title of Contact
Email	Email address of Contact
Home number	Home phone number of Contact
Business number	Business phone number of Contact
Business fax	Business fax of Contact
Mobile number	Mobile phone number of Contact
Home address	Home address of Contact
Company	Company name for Contact
Business address	Business address of Contact
Birthday	Birthday of Contact
Anniversary	Wedding Anniversary of Contact

Table 7.3: Contact Data Items

7.4.4 Tasks Consistency Model

Within Outlook, the Task folder stores tasks to be completed by a user. The Task information used within this implementation consists of seven data items (table 7.4), referring to the actual task to be performed and its associated dates.

The type of consistency model chosen for maintaining Task information uses the *mandatory remote update approach* within a client-server environment. All operations like *open*, *new*, *update* and *delete* are performed on a remote server prior to being performed locally on the client. If the remote operation fails, then the associated local operation is forced to fail as well.

7.4.5 Outlook 2002 Relationship Component Implementation

The implementation of the Outlook Relationship Component consists of consistency models for each of the Outlook folders and a dispatcher to bridge

Data	Description and Purpose
Body	Contains a detailed description of the task to be completed
Subject	A shortened description of the task
Time Received	The time this task was created by a user
Due Date	The due date of the task
Start Date	The start date of the task
Priority	The priority of the task (In Outlook there are three levels HIGH, LOW and NORMAL)
Status	Indicates the status of the task (In Outlook there are five levels, NOT_STARTED, IN_PROGRESS, COMPLETED, WAITING_ON_SOMEONE_ELSE and DEFERRED

Table 7.4: Task Data Items

GLOMAR with the folder-specific consistency models instances. In addition, an Outlook COM Add-In is built to intercept Outlook operations and forward them to GLOMAR (figure 7.8).

Outlook COM Add-In

By creating a COM (Box 1998) object that implements the interface *IDTExtensibility2* and using the Outlook Object Model 10.0, a module can be installed into Outlook (figure 7.9) to intercept operations and perform any additional processing (Rice 2000)[3].

The implementation of this COM Add-In used the Visual Basic 6.0 template and designer classes supplied with the book (Byrne 2001). Within this COM Add In, each specific Outlook folder has the following operations intercepted:

- **OpenFolder.** This intercepts the Outlook Explorer selecting a specific folder. The result of this operation is that the specific folder is opened with the contents visible within the Outlook Inspector. This is not cancellable.

[3]Microsoft recently released a .NET Add-In template class to Visual Studio.NET, however this was unavailable at the time of development

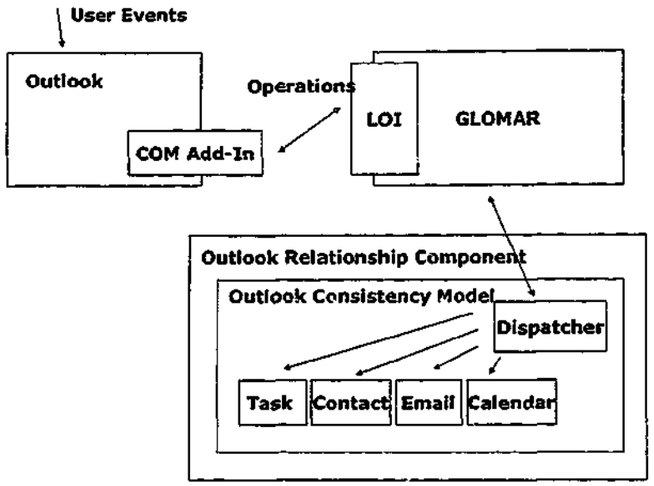

Figure 7.8: Outlook Relationship Component Design

- **CloseFolder.** This intercepts the Outlook Explorer deselecting a specific folder. The result of this operation is that the current specific folder is removed from the Outlook Inspector. This is not cancellable.
- **ItemAdd.** This intercepts when an item has been added to a specific folder. This is not cancellable. However, if the adding of a new item fails, then it is deleted automatically.
- **BeforeDelete.** This intercepts an operation prior to an item being deleted (This is new to Outlook 2002). This is cancellable.
- **OpenItem.** This intercepts when an item is opened within the Outlook Inspector. This is cancellable.
- **CloseItem.** This intercepts when an item is closed within the Outlook Inspector. This is cancellable.
- **WriteItem.** This intercepts when an item is saved within the Outlook Inspector. This is cancellable.

When an operation has been intercepted, the COM Add-In performs additional processing before forwarding it to GLOMAR. This results in information being pinned to an operation so that a dispatcher, within the Outlook Relationship Component, can direct the appropriate Outlook operation to the appropriate

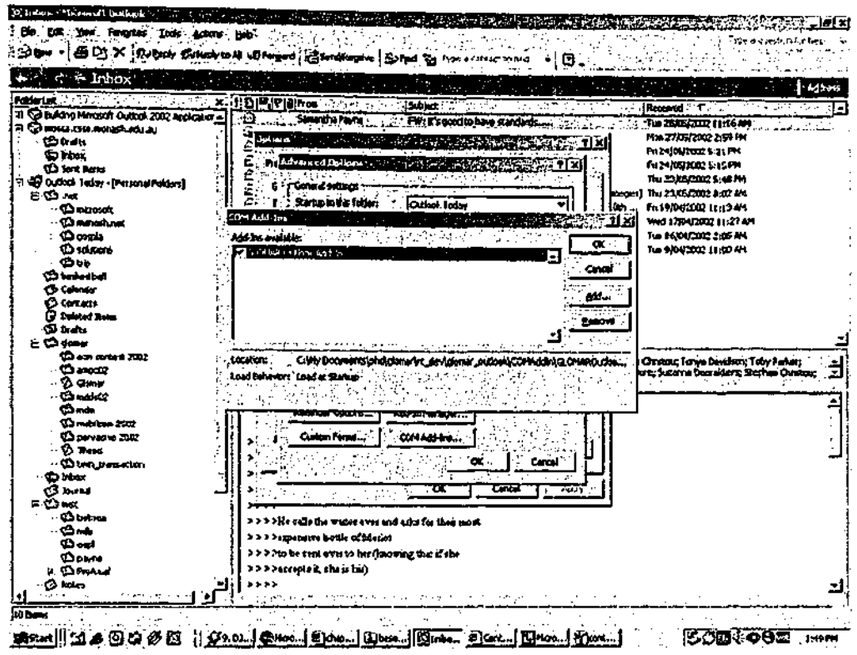

Figure 7.9: Outlook COM Add-In

consistency model and appropriate method. This means the *tag* variable is filled with Outlook specific data, detailing not only the type of operation, but the folder type, item type, as well as other data (table 7.5). The resulting call is then made to the COM bridge via the 32-bit API (as the COM Add-In is written in VB 6.0).

Depending upon the returned results, the operation within Outlook is either committed or cancelled. A cancelled operation also results in a dialog box appearing, indicating an error has occurred (figure 7.10).

Dispatcher

The generic nature of the Consistency Model interface does not apply itself well to the requirements of the Outlook Relationship Component. This is due to the type of operations that can occur and the number of folders that must be individually governed. For this reason, rather than changing the Consistency Model interface (which would require a major reworking of GLOMAR, and reduce the DFS's flexibility), an additional interface was individually implemented within

Data	Description and Purpose
Entry Id	Contains the GUID of the item
Storage Id	Contains the GUID of the folder
Item Type	Whether the operation is occurring on a folder or item
Folder	The folder the operation is occurring within (DRAFTMAIL, SENTMAIL, INBOXMAIL, CONTACT, APPOINTMENT, ETC...)
Operation	The type of operation. These are different to the standard operations (e.g. CREATENEWITEM, CHANGEITEM, OPENITEM, CLOSEITEM, WRITEITEM, REMOVEEXISTINGITEM)
Storage String	Can contain any user-defined string, eg XML.

Table 7.5: Outlook Information passed via the *tag* paraemter

a single Relationship Component, specifically targeting Outlook. However, the Dispatcher approach as defined here is not always necessary for all scenarios. Rather, it can be seen as one method of bridging highly specific operations and requirements with the generic interface defined by GLOMAR.

The interface *IOutlookProcessing* (figure 7.11) was purposely built to handle Outlook operations. To allow for multiple implementations of this interface for each folder type within the Outlook Relationship Component, a dispatcher was built. The purpose of the dispatcher is to direct operations from the Outlook Relationship Component's Consistency Model to the appropriate folder-specific Consistency Model implementation. In other words, as operations occur, an additional process determines the functionality to invoke based on information passed to it via the *tag* parameter. For example, when an *Open* operation occurs in Outlook, the COM Add-In makes the request to GLOMAR, filling the *tag* parameter indicating, the data item type (folder or item), the operation type and the details of the actual item. Once the request enters the Outlook Relationship Component's Consistency Model, the dispatcher determines the correct folder specific Consistency Model implementation to forward the request to and then what specific Outlook operation it should invoke. Figure 7.12 depicts the UML based diagram of the Outlook Relationship Component.

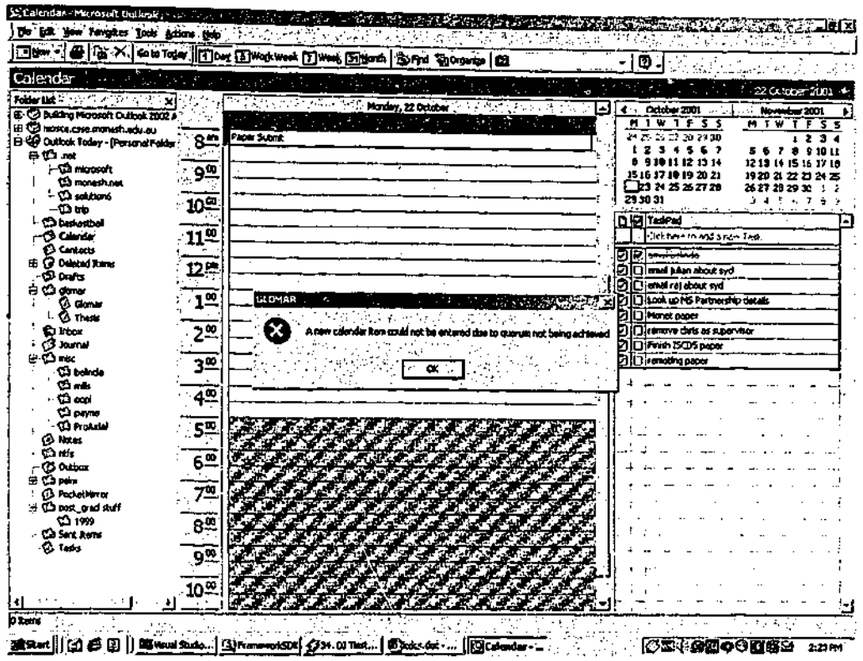

Figure 7.10: GLOMAR Operation Failing within Outlook

Sent Mail, Draft Mail and Inbox Component Implementation

The implementation of all Mail-related Consistency Models is similar, with the *RemoveItem*, *CreateItem* and *ChangeItem* operations written into a XML file prior to local execution. For example, when an Outlook user deletes an item, the details of that operation and item are stored within a XML file. Figure 7.13 details the Mail log XML file.

Depending upon the period set, which is different for each mail folder implementation, a user-specific service is activated (installed within the Service Manager). This service firstly reads the XML file and produces a list of operations to propagate. Then an attempt is made to contact each node and propagate the associated operations.

Communication between nodes is facilitated as each service exposes via the .NET remoting infrastructure three methods, *AddMail*, *UpdateMail* and *DeleteMail*. The local service contacts the equivalent service on the remote node and uploads

the specific operations via these methods. Once each operation is successfully uploaded and performed on all remote nodes, it is removed from the XML file.

Calendar Component Implementation

The Calendar-related Consistency Model is the most complex of the folder-specific Consistency Models to implement. This is due to the complexity associated with determining if quorum is achieved, determining which Calendar information is dominant and the process of populating Outlook with the most up-to-date events.

When the Calendar folder is opened, the *OpenFolder* method is called. This method firstly determines if quorum can be achieved, by calling the *IsQuorum* method on the *WebServiceList* object. This object contains a list of XML Web Service proxies that represent each node of the network. The *IsQuorum* method proceeds to contact each node, recording if the node was contactable or unreachable. Once complete, the ratio of available and unavailable nodes is evaluated to determine if quorum was achieved (>50%).

If quorum is achieved, then the *OpenFolder* method contacts each available node and requests its version vector. As only GLOMAR's XML Web Service is used, this information needs to be marshalled into a number of *Read* calls (the *GetAttr* method is used to gain the total number of *Read* calls required). This involves marshalling the version vector into 64 bit integer blocks on the remote side and reconstituting a version vector instance on the local side. All version vectors (downloaded and local) are then compared, resulting in the dominant node being determined.

If a remote node is deemed to be dominant, then the *OpenFolder* method downloads the node's events locally. Similar to how the version vectors were gained, the *Read* method of the remote node is called multiple times. This results in a number of 64 bit integer blocks returned. When marshalled, the results represent the dominant node's list of events. Once complete, the local list of events is removed and the downloaded list is inserted in its place. Only when this is successful, is the local version vector adjusted.

For operations that result in the *RemoveItem*, *CreateItem* and *ChangeItem* methods being invoked, the *IsQuorum* method is firstly called. This attempts to determine if quorum is achieved. Failure to achieve quorum aborts the operation, were as success attempts to propagate the operation.

The process involved in propagating operations to all available nodes is based on a two-phase commit protocol (Lampson and Sturgis 1976) (figure 7.14). The first phase uploads a serialised representation of the operation and event itself to each available node. Packaged with the operation is an ID number (randomly generated number that is unique only to this operation). This operation is then written to a temporary store and hashed on the ID number, rather than immediately performed.

When all nodes have received the operation, then the second phase starts. A commit message is sent to each available node. Packaged with this message is the ID number of the operation to execute. When the commit message reaches the remote node, the operation is retrieved from the store (the store is persistent) and executed. Once the two-phase commit protocol is completed (success or failure), the version vector for both the local and remote nodes are updated.

Contact Component Implementation

The central element of this implementation is the Contact XML Web Service (figure 7.15). This service exposes four methods; *GetAllContacts*, *ChangeExistingContacts*, *AddNewContacts* and *DeleteContacts*. As the name suggests, each method provides the functionality to access and modify a central store of Contacts.

As *RemoveItem*, *CreateItem* and *UpdateItem* operations occur, they are posted to the Contact XML Web Service. Only when successfully performed by the Contact XML Web Service, will the operation execute locally.

Contact information is downloaded from the XML Web Service via a user-specific service. This process periodically queries the XML Web Service for a list of all contacts (calling the *GetAllContacts* method), then downloads them, replacing the existing Contact information with the new list.

Tasks Component Implementation

The Task-related Consistency Model forwards the Outlook operations *OpenFolder*, *RemoveItem*, *CreateItem* and *ChangeItem* to the Task XML Web Service. The Task XML Web Service exposes four methods (figure 7.16), *GetAllTasks*, *ChangeExistingTask*, *AddNewTask* and *DeleteTask*. This XML Web Service encapsulates an XML file of Tasks, providing an interface to access and modify them.

This approach is similar to the Contact-related Consistency Model (section 7.4.5). However, all Tasks are downloaded when the folder is opened, not periodically downloaded as with Contacts. Thus, when the *OpenFolder* method is called, it first downloads from the Task XML Web Service the current list of tasks. Once complete, the local list of Tasks is then replaced by the downloaded list of Tasks.

For each *RemoveItem*, *CreateItem* and *ChangeItem* operation intercepted, the Task-related Consistency Model posts the operation to the Task XML Web Service. Only when the operation is completed successfully by the XML Web Service, does the operation execute locally.

7.4.6 Analysis of Outlook 2002 Relationship Component

The Outlook Relationship Component demonstrates fine grain consistency maintenance and concurrency control for Outlook data items. Unlike other case studies implementations (*TTM*, *Get Latest* and *ROWA*) which provide a generic solution, the Outlook Relationship Component shows how an application-specific Relationship Component is created and serviced by GLOMAR.

Using Outlook as the test application, illustrates how only minor modifications to the actual application are required to exploit the benefits of GLOMAR. In other words, the processes and functionality of Outlook were not affected to accommodate GLOMAR. Only some minór modifications were required so operations could be intercepted and forwarded to GLOMAR.

The Outlook Relationship Component illustrates the flexibility of GLOMAR, as each Consistency Model used a different approach and mechanism (for example, client-server, peer-to-peer, GLOMAR to XML web service communication). Regardless of the data structure or functionality required to service the consistency and concurrency requirements of Outlook, the Relationship Component was able to support it.

As stated in Chapter 3, GLOMAR provides a way of balancing resource usage. The Outlook Relationship Component is an illustration of this accomplishment. Rather than propagating the entire contents of the Outlook data store (which can be 60 to 100 mb[4]) every time a modification operation occurs, only the active data and operations are propagated. Traditionally, if a user wanted to

[4]This figure was determined from a brief survey of postgraduate students and academics using Outlook within the School of Computer Science and Software Engineering, Monash University

replicate Outlook data, whole file propagation was the only solution. This adversely affects bandwidth and degrades performance in a constrained bandwidth environment. With the Outlook Relationship Component, there is a reduced reliance upon the bandwidth, as operations are propagated at appropriate times, without greatly compromising the consistency requirements of Outlook's data.

In addition, because of the Outlook Relationship Component, Outlook has been simply converted into a distributed application. Thus, the Outlook Relationship Component demonstrates that by externalising the mechanism for handling consistency maintenance and concurrency control, an additional level of complexity and functionality can be added, without affecting the original application.

7.5 Aggregated Analysis of the Case Studies

To illustrate the run-time benefits of the multiple consistency model approach, all of the implemented Relationship Components were instantiated concurrently. Thus, while the notepad application was running, its Relationship Component was governing text files using *ROWA*, *Get Latest* and *TTM*. Simultaneously, operations generated by Outlook were handled by its Relationship Component. As a result, multiple concurrency control and consistency maintenance mechanisms were simultaneously running within a single DFS.

The case studies also resulted in the development of a large library of Context Providers. A library of C++ (Lippman and Lajoie 1998) based hardware metric methods (Wendt 2002) were wrapped by a number of Context Providers. Table 7.6 details the environmental information that is available.

The case studies illustrate GLOMAR has potential far beyond merely encapsulating concurrency control and consistency maintenance functionality. Rather, it can be used for other tasks that require transparent redirection of file operations. For example, there are software systems that support write operations on read-only media, like a CD-ROM (DirectCD (Roxio 2002) and Windows XP (Microsoft 2002a)). This is made possible by redirecting operations away from the CD to a write store. When a user reads the CD, the actual contents visible to the user represent an amalgamation of both the contents of the CD and that of the write store. GLOMAR was never intended to provide the framework to do this, but due to its flexibility, it is capable of this. Thus, GLOMAR can be applied to solutions where some additional "value adding" to an operation is required.

Type	Description
CPU	Speed
	ID
	Family
	Model
	Number of Processors
Drive	Free Space
	Name
	Total Space
Memory	Get Available Page File Size
	Get Available RAM Size
	Get Available Virtual Size
	Get Total Page File Size
	Get Total RAM Size
	Get Total Virtual Size
Operating System	Description
Socket	Domain Name
	IP Address

Table 7.6: Supplied Context Providers

All the case study implementations highlighted one area of concern relating to the handling and management of Relationship Components. For example, an application has two Relationship Components, one for disconnected and one for connected. Both these components share information regarding the structure and location of a cache. A new Relationship Component (with no knowledge of the location or structure of the cache) could very easily replace an existing Relationship Component. As a result, the life-cycle defined by the two original Relationship Components is comprised. This issue is further addressed in section 9.2.

7.6 Summary

This chapter has illustrated the encapsulation of different concurrency control and consistency maintenance mechanisms within a single DFS, through actual Relationship Component implementations. The first implementation demonstrated a semantically similar approach to a traditional file system in a disconnected environment. The second implementation demonstrated a Relationship Component that handles the constraints of a mobility-enabled DFS,

using a transactional model. The final implementation demonstrated a consistency model that services the specific needs of a single application, in this case Outlook 2002.

Each example demonstrates GLOMAR's flexibility to support consistency maintenance and concurrency control mechanisms. It also shows that a balance can be achieved between consistency and resource usage. The next chapter evaluates and discusses the cost of the GLOMAR middleware layer, primarily to determine if the multiple consistency model approach is efficient.

```
Public Interface IOutlookProcessing

  Public Function OpenFolder(ByVal handle As FHANDLE,
                             ByVal details As OutlookOperationDetails)
                             As Glomar.OperationStatus
  Public Function CloseFolder(ByVal handle As FHANDLE,
                              ByVal details As OutlookOperationDetails)
                              As Glomar.OperationStatus
  Public Function OpenItem(ByVal handle As FHANDLE,
                           ByVal details As OutlookOperationDetails)
                           As Glomar.OperationStatus
  Public Function CloseItem(ByVal handle As FHANDLE,
                            ByVal details As OutlookOperationDetails)

                            As Glomar.OperationStatus
  Public Function RemoveItem(ByVal handle As FHANDLE,
                             ByVal details As OutlookOperationDetails)
                             As Glomar.OperationStatus
  Public Function CreateItem(ByVal handle As FHANDLE,
                             ByVal details As OutlookOperationDetails)
                             As Glomar.OperationStatus
  Public Function ChangeItem(ByVal handle As FHANDLE,
                             ByVal details As OutlookOperationDetails)
                             As Glomar.OperationStatus
  Public Function GetAttrFolder(ByVal handle As FHANDLE,
                                ByRef fileattr As Glomar.FILE_ATTR,
                                ByVal details As OutlookOperationDetails)
                                As Glomar.OperationStatus
  Public Function ReadFolder(ByVal handle As FHANDLE, ByVal length As Long,
                             ByVal offset As Long, ByRef buffer As Long,
                             ByVal details As OutlookOperationDetails)
                             As Glomar.OperationStatus


  Public Function RemoteOpenFolder(ByVal handle As FHANDLE,
                                   ByVal details As OutlookOperationDetails)
                                   As Glomar.OperationStatus
  Public Function RemoteCloseFolder(ByVal handle As FHANDLE,
                                    ByVal details As OutlookOperationDetails)
                                    As Glomar.OperationStatus
  Public Function RemoteReadFolder(ByVal handle As FHANDLE, ByVal length As Long,
                                   ByVal offset As Long, ByRef buffer As Long,
                                   ByVal details As OutlookOperationDetails)
                                   As Glomar.OperationStatus
  Public Function RemoteGetAttrFolder(ByVal handle As FHANDLE,
                                      ByRef fileattr As Glomar.FILE_ATTR,
                                      ByVal details As OutlookOperationDetails)
                                      As Glomar.OperationStatus
  Public Function RemoteOpenItem(ByVal handle As FHANDLE,
                                 ByVal details As OutlookOperationDetails)
                                 As Glomar.OperationStatus
  Public Function RemoteCloseItem(ByVal handle As FHANDLE,
                                  ByVal details As OutlookOperationDetails)
                                  As Glomar.OperationStatus
  Public Function RemoteRemoveItem(ByVal handle As FHANDLE,
                                   ByVal details As OutlookOperationDetails)
                                   As Glomar.OperationStatus
  Public Function RemoteCreateItem(ByVal handle As FHANDLE,
                                   ByVal details As OutlookOperationDetails)
                                   As Glomar.OperationStatus
  Public Function RemoteChangeItem(ByVal handle As FHANDLE,
                                   ByVal details As OutlookOperationDetails)
                                   As Glomar.OperationStatus
End Interface
```

Figure 7.11: *IOutlookProcessing* Inverface

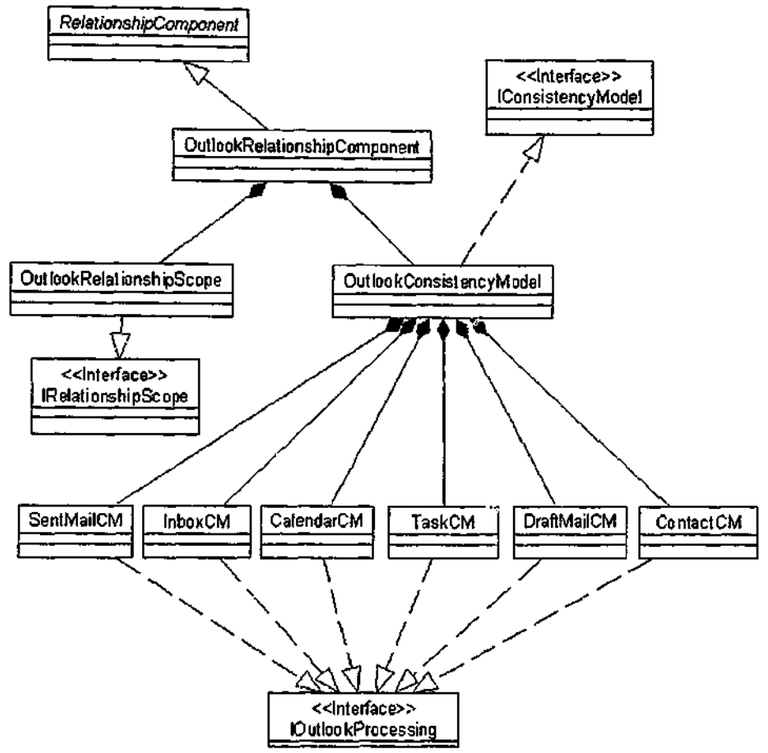

Figure 7.12: Outlook Consistency Model

```
<sentmaillist>
      <ip address="130.194.0.1">
         <add>
            <sentmail>
                  <subject>Hello Simon</subject>
                  <body>Testing Relationship Component</body>
                  <toaddress>foo@bar.com</toaddress>
            </sentmail>
         </add>
      </ip>
      <ip address="130.194.0.2">
         <add>
            <sentmail>
                  <subject>Hello Simon</subject>
                  <body>Testing Relationship Component</body>
                  <toaddress>foo@bar.com</toaddress>
            </sentmail>
         </add>
      </ip>
</sentmaillist>
```

Figure 7.13: Mail Log XML File

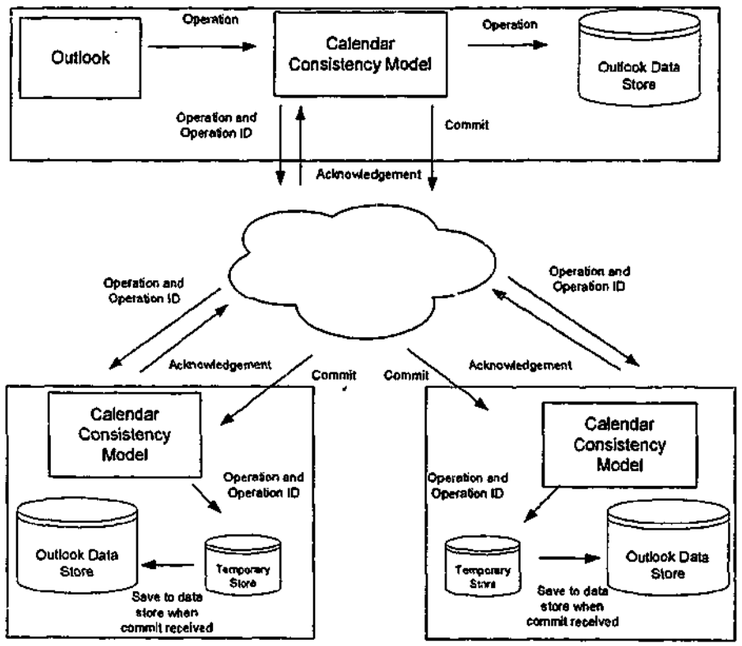

Figure 7.14: Calendar Two Phase Commit Protocol

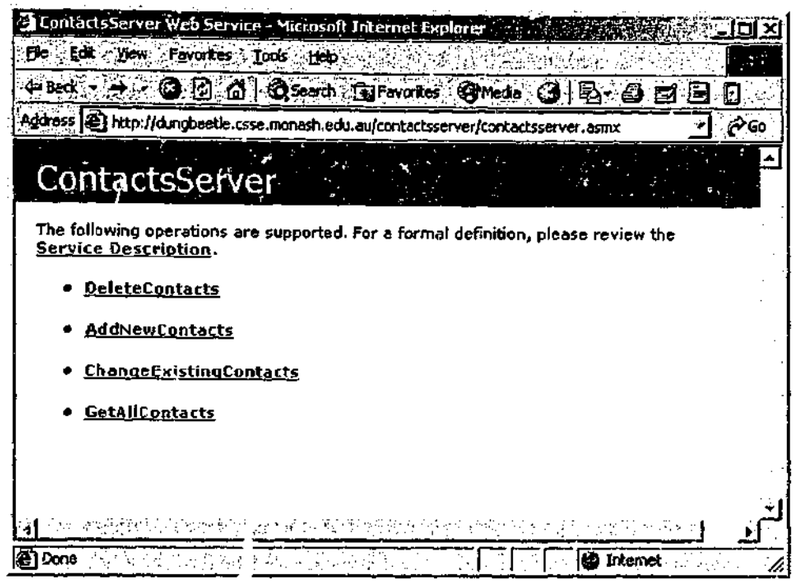

Fig .re 7.15: Contact XML Web Service

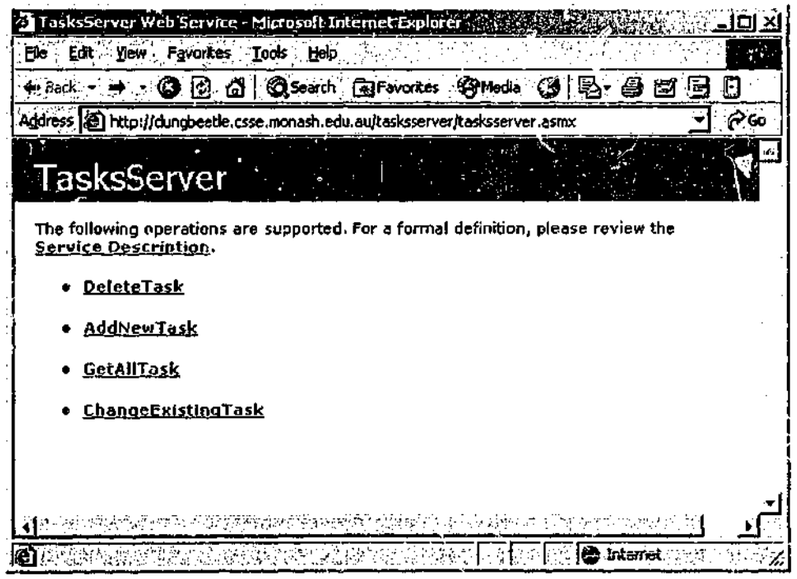

Figure 7.16: Task XML Web Service

Chapter 8

GLOMAR Evaluation

This chapter evaluates the GLOMAR middleware layer for the purpose of determining the efficiency of the multiple consistency model approach. Firstly, this chapter discusses the evaluation methodology, focusing on the aims and difficulties associated with evaluating GLOMAR as a middleware. Secondly, a detailed analysis of performance and resource usage, primarily focusing on the inner workings of the GLOMAR middleware layer is presented. Finally, this chapter concludes with a discussion on the evaluated results.

8.1 Introduction

The previous chapter (Chapter 7) demonstrates the effectiveness of the GLOMAR middleware layer in supporting the multiple consistency model approach by qualitatively analysing case studies. This chapter in turn quantitatively evaluates the efficiency of the GLOMAR middleware layer.

All experiments proposed and described in this chapter are targeted at the cost of using the GLOMAR middleware layer only. Therefore, the initiation of the GLOMAR middleware layer and the cost of processing an operation (excluding Relationship Component functionality) are measured as part of all experiments.

The evaluation of individual Relationship Components is omitted from this chapter to highlight the generic nature of the GLOMAR middleware layer, rather than focusing on application-specific issues.

The resulting implementation of GLOMAR combines elements of a DFS and a traditional middleware. This is because it hides the complexity of managing different concurrency mechanisms via a simplified pseudo standard interface. For the purpose of evaluation, this description specifies GLOMAR implementation

capabilities and target domain, rather than the whole GLOMAR framework and methodology. This combination is the strength of GLOMAR. However, the comparison of GLOMAR to either a DFS and/or middleware is inappropriate. This is because elements of GLOMAR's middleware affect performance, when compared with other DFSs. On the other hand, GLOMAR's DFS elements adversely affect performance comparison with traditional middlewares. From the literature reviewed, no relevant comparison methodology has been found for comparing a DFS with a middleware.

8.2 Aim and Experimental Methodology

This chapter aims to prove that the associated algorithms within the GLOMAR middleware layer provide a suitable basis for implementing the multiple consistency model approach. In other words, the benefits of GLOMAR are not outweighed by the cost associated with the management of Relationship Components.

This evaluation analyses the scalability of the GLOMAR middleware layer. As much of the processing is associated with the managing of Relationship Components and Clones, this chapter discusses the impact of varying the number of Relationship Components and Clones. As a by-product of evaluating scalability, a suggested number of Relationship Components and Clones will be produced.

The experiments evaluate both the initiation and the processing of operations within the GLOMAR middleware layer (figure 8.1). For each experiment, the number of Relationship Components and Clones are assumed as input parameters. The experiments performed have been grouped and are outlined below:

- **Initiation of the GLOMAR middleware layer.** These experiments aim to determine the time and resource usage of the initiation of the GLOMAR middleware layer. These experiments include:
  - **Scaling the number of Clones.** This experiment aims to determine the time and memory consumption when the number of Clones is varied.
  - **Scaling the number of Relationship Components.** This experiment aims to determine the time and memory consumption when the number of Relationship Components is varied.

- **Processing of Operations within the GLOMAR middleware layer.** These experiments aim to determine the time of processing an operation within the GLOMAR middleware layer. These experiments include:
  - **Scaling the number of Clones.** This experiment aims to determine the average time per operation when the number of Clones is varied.
  - **Scaling the number of Relationship Components; Average time per Operation.** This experiment aims to determine the average time per operation when the number of Relationship Components is varied.
  - **Scaling the number of Relationship Components; Time of *Relationship Component Processing*.** This experiment aims to determine the time taken by *Relationship Component Processing* when the number of Relationship Components is varied.
  - **Scaling the number of Relationship Components; Average time of partial *Relationship Component Processing*.** This experiment aims to determine the time taken by partial *Relationship Component Processing* when the number of Relationship Components is varied. This is to illustrate the effects within a real life situation, when not all processes of the *Relationship Component Processing* are invoked.
  - **Scaling the number of Relationship Components; Average time of Instantiation.** This experiment aims to determine the time to instantiate different Relationship Components when the number of Relationship Components is varied.

The input parameters for each experiment are detailed in table 8.1. These values represent a broad range of input parameters to achieve the aims of the experiments detailed in this chapter. They represent both the likely range of Clones and Relationship Components and the potential limit of concurrent Clones and Relationship Components experienced within a DFS. Table 8.2 details the run-time environmental configuration of these experiments.

In order to generate the Relationship Components and Clones used in the experiments, an auxiliary application (*Relationship Component Generator*) was built. Its purpose was to create the skeleton code, Clone List and a *VB.NET makefile* to represent the targeted number of Relationship Components. As

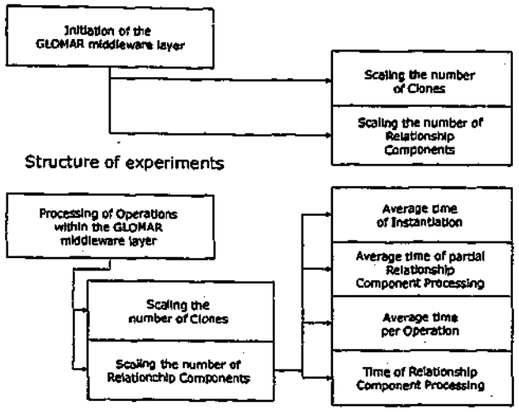

Figure 8.1: Experiment structure

Input Parameters	Number
Clones	1, 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000
Relationship Components	1, 5, 10, 50, 100, 500, 1000

Table 8.1: Experimental Input Parameter

these Relationship Components are used in the testing of the GLOMAR middleware layer, they contain no actual consistency maintenance code. Rather, they contain a minimal implementation of the Consistency Model interface.

8.2.1 Evaluating the Initiation of the GLOMAR middleware layer

Three aspects of the initiation of the GLOMAR middleware layer were evaluated. These were the overall memory consumption, the overall time taken and the initiation time of each stage.

The memory consumption is derived by recording the total memory allocated by the .NET garbage collector on all three heaps (Gen0, Gen1 and Gen2 (Watkins et al. 2003)) using the application *GlowCode.NET* (GlowCode.Com 2002).

.NET version	1.0.3705
Processor	Intel Pentium 434Mhz
Memory	256 Mb
OS	Windows 2000 (5.00.2195)
Storage	7.8 Gb

Table 8.2: Experimental Environment Configuration

The time to complete each different stage is determined by outputting the current system time at the end of each stage. The stages evaluated within the initiation are (in order):

- **Port Setup.** After the .NET remoting infrastructure has setup the listening port.
- **Clone Distribution Manager.** After the Clone Distribution Manager has been initiated, including reading in data from the Clone List.
- **Executive.** After the Executive has been initiated.
- **Relationship Component Repository.** After the Relationship Component Repository has been initiated, including the installing and instantiation of all Relationship Components.
- **Remote Operation Interface.** After the Remote Operation Interface has been initiated.
- **Local Operation Interface.** After the Local Operation Interface has been initiated.

This evaluation consists of two experiments, one where the number of Relationship Components are varied and another where the number of Clones are varied.

8.2.2 Evaluating Processing an Operation

The time taken to process an operation within the GLOMAR middleware layer is evaluated within these experiments. To extract the time per operation, each experiment (excluding the time of *Relationship Component Processing*) consists of 100 operations (*write* operations) being passed to GLOMAR. These results are then averaged.

For each experiment evaluating the average time per operation and time of *Relationship Component Processing*, three types of Clone Lists and three types of Relationship Components were produced by the *Relationship Component Generator*. These include the *best*, *average* and *worst* case scenarios.

For Clones, the *best* case scenario is when one access is required to gain a Clone from within the Clone List. The *average* case scenario is when accessing a Clone requires N/2 accesses (N is the total number of Clones in the Clone List). The *worst* case scenario is when accessing a Clone requires N accesses.

For Relationship Components, the *best* case scenario is when one access is required to gain a Relationship Component from within the list of instantiated Relationship Components. The *average* case scenario is when accessing a Relationship Component requires N/2 accesses (N is the total number of Relationship Components in the list of instantiated Relationship Components). The *worst* case scenario is when accessing a Relationship Component requires N accesses.

For each of these *best*, *average* and *worst* case scenario Relationship Components, two instantiation types are defined, *singleton* or *new instance*. The motivation for allowing the two instantiation types is to isolate the cost of implementing the two types, independent of other processing costs.

The average time per operation experiment determines the average time taken to process an operation. As a result, the 100 operations absorb the time taken by *Relationship Component Processing*. Whereas, the time of *Relationship Component Processing* experiment results in the actual time taken by the *Relationship Component Processing* to process an operation.

The experiments (average time per operation and time of *Relationship Component Processing*) only invokes *Relationship Component Processing* once. This is because the scenario is static for the duration of the experiment (all 100 operations). Thus, this type of experiment is analogous to a system that exists on a stable network.

To evaluate GLOMAR in a more volatile system, a *random* case scenario was defined. When a Relationship Component is defined as *random* case, the scenario (thus Relationship Scope) is randomly validated and invalidated. This case is used to determine the cost of instantiating Relationship Components of different types as well as determining the time when only partial *Relationship Component Processing* is implemented.

8.3 Initiation of the GLOMAR middleware layer: Results and Discussion

As earlier stated (section 3.3.1), GLOMAR incurs additional cost per operation. An effort was made to avoid this by allocating costly activities to more appropriate times. For example, rather than initiating Relationship Components when required (just in time), an instance is available in-memory. This additional memory usage is offset by the benefit of having the Relationship Component available immediately. This particular experiment records the time and memory consumption associated with the initiation of the GLOMAR middleware layer.

8.3.1 Scaling the number of Clones

Memory consumption during the initiation of the GLOMAR middleware layer as the number of Clones vary are detailed in figure 8.2. These results show a linear relationship between the consumption of memory and the number of Clones ($y = 0.121x + 149.21$ with a confidence value of $R^2 = 1$). Noticeably, the amount of memory consumed remains relatively insignificant (approximately 150 Kbytes to 160 Kbytes) until 100 Clones or more are instantiated (table 8.3). However, even with a very large number of Clones (10,000), GLOMAR's memory footprint remains acceptable (approximately 1.5 Mbytes). However, when 50,000 Clones or more are instantiated, memory consumption becomes more noticeable.

Number of Clones	Memory (Kbytes)
1	149.2139
5	149.6982
10	150.3037
50	155.1475
100	161.2021
500	209.6396
1,000	270.1865
5,000	754.5615
10,000	1360.03
50,000	6200.204

Table 8.3: Initiation of the GLOMAR middleware layer Memory Consumption Table (Clones)

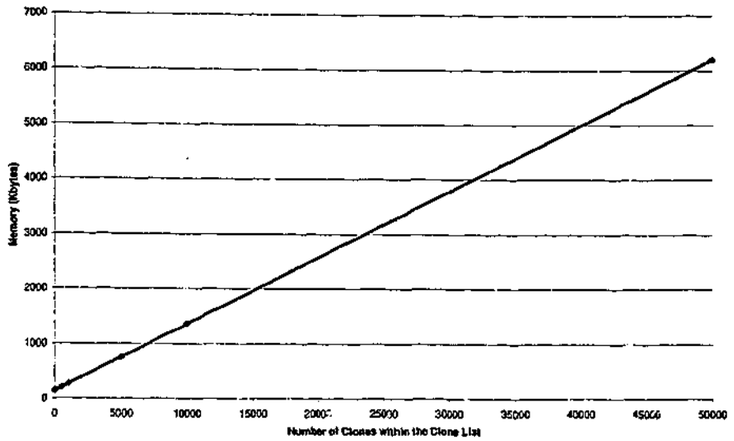

Figure 8.2: Initiation of the GLOMAR middleware layer Memory Consumption (Clones)

Table 8.4 details the time taken to initiate the GLOMAR middleware layer. From one to 500 Clones, the time remains unchanged (approximately 2.5 seconds), as the majority of the time is spent opening the Clone List (the XML file) for reading, rather than the reading of the actual data itself. When the number of Clones is greater than 5,000, the time to open the Clone List is less than the time to read the Clone List. Thus, GLOMAR has the potential to manage one to 1,000 Clones with no major difference in time.

Number of Clones	Time (seconds)
1	2.253299
5	2.343398
10	2.323402
50	2.303296
100	2.313306
500	2.533606
1,000	2.874099
5,000	6.329101
10,000	9.093107
50,000	116.5276

Table 8.4: Initiation of the GLOMAR middleware layer Times (Clones)

The initiation time of the GLOMAR middleware layer (table 8.4) has a linear relationship to the actual number of Clones ($y = 0.0023x + 0.2889$ with a confidence value of $R^2 = 0.9785$).

When evaluating the initiation time based on the stages within the GLOMAR middleware layer, the majority of the time (as expected) is spent initiating the Clone Distribution Manager. However, as figure 8.3 details, the effects of scaling the number of Clones is seen at the Clone Distribution Manager initiation stage. This illustrates that the effects of scaling the number of Clones have been successfully isolated to the most appropriate stage, in this case, the initiation of the Clone Distribution Manager.

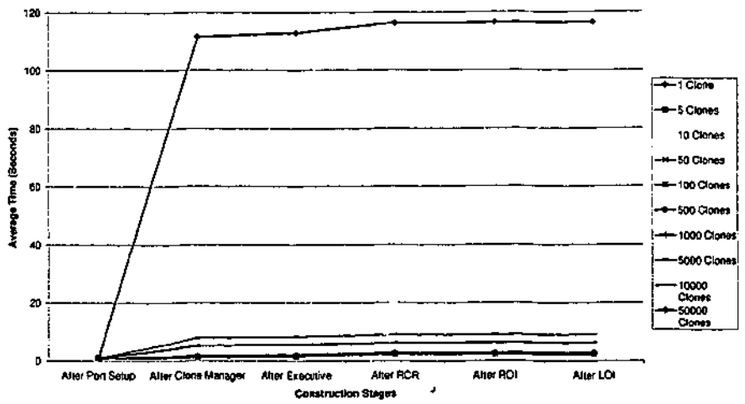

Figure 8.3: Initiation of the GLOMAR middleware layer Times, Based on Stages (Clones)

8.3.2 Scaling the number of Relationship Components

Memory consumption of the initiation of the GLOMAR middleware layer as the number of Relationship Components vary is detailed in figure 8.4. With no Relationship Components, the memory footprint of the GLOMAR middleware layer is approximately 118 Kbytes. This is an approximate value as the .NET garbage collector fluctuates memory consumption over time.

As figure 8.4 shows, as the number of Relationship Components increases, the consumption of memory also proportionally increases. This relationship is linear ($y = 0.7572x + 143.47$) with a confidence value of $R^2 = 0.9986$.

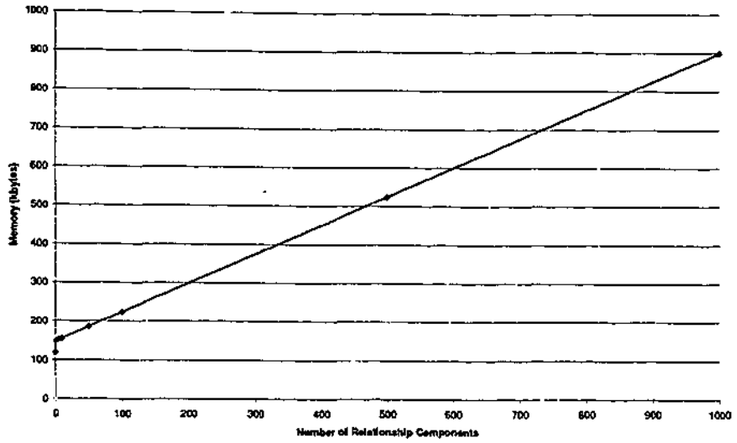

Figure 8.4: Initiation of the GLOMAR middleware layer Memory Consumption (Relationship Components)

Table 8.5 details the time taken to initiate the GLOMAR middleware layer as the number of Relationship Components varies. With no Relationship Components installed, the time taken is 4.2 seconds. From one to 50 Relationship Components, the time taken does not vary greatly (average 4.7 seconds). When 100 or more Relationship Components are instantiated, initiation of the middleware becomes noticeable.

Number of Relationship Components	Time(seconds)
0	4.226099
1	4.887002
5	4.576602
10	4.876992
50	4.816998
100	5.347699
500	9.503706
1,000	12.8185

Table 8.5: Initiation of the GLOMAR middleware layer Times (Relationship Components)

A surprising observation is the time taken to instantiate one to 50 Relationship Components is approximately the same. This is because the actual number of

Relationship Components has no direct reflection on the actual time to initiate the GLOMAR middleware layer. This is particularly shown in the fact that the time to instantiate 25 Relationship Components is less than the time to instantiate two Relationship Components. This experiment was repeated multiple times, with the same results recorded. The explanation for this anomaly can only be attributed to the influence of the .NET garbage collector. It would seem that the implementation of the ArrayList (Watkins, Hammond, and Abrams 2003) is customised towards larger numbers of items. Thus, when implementing a small number of components within the ArrayList, a more aggressive approach to memory management is implemented by the .NET garbage collector.

Figure 8.5 illustrates the initiation of the GLOMAR middleware layer broken into stages as the number of Relationship Components is varied. The time taken by each stage during initiation of the GLOMAR middleware layer (except for the Relationship Component Repository) is similar regardless of the number of Relationship Components instantiated. Figure 8.5 shows as the number of Relationship Components increases, so does the time taken to initiate the Relationship Component Repository. This illustrates that the impact of scaling the number of Relationship Components is isolated to the initiation of the Relationship Component Repository.

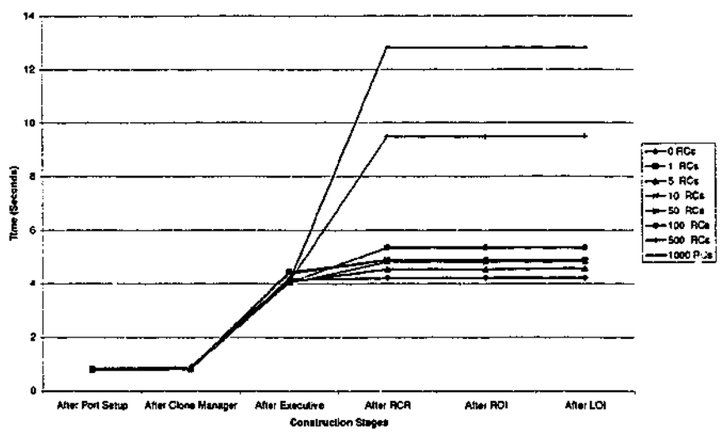

Figure 8.5: Initiation of the GLOMAR middleware layer Times Based on Stages (Relationship Components)

8.4 Processing an Operation: Results and Discussion

The cost of performance and resources within the initiation of the GLOMAR middleware layer is less critical than the cost imposed upon processing an operation. Much of the benefits of the multiple consistency model approach discussed in previous chapters are dependent upon the premise that only a small amount of time is added to a file operation. This series of experiments investigates the performance overhead (time) the GLOMAR middleware layer places upon an operation, focusing upon how scaling the number of Relationship Components and Clones affects internal processes.

8.4.1 Scaling the number of Clones

The result of the average time per operation, when scaling the number of Clones is illustrated in figure 8.6. The average time per operation when one to 5,000 Clones are implemented is 0.02 to 0.04 seconds. This result is independent of the position of the Clone within the Clone List (*best*, *worst*, *average* case scenario). However, once the number of Clones is equal to or greater than 5,000, the performance overhead experienced by the GLOMAR middleware layer is increased, in particular for *worst* and *average* case scenarios.

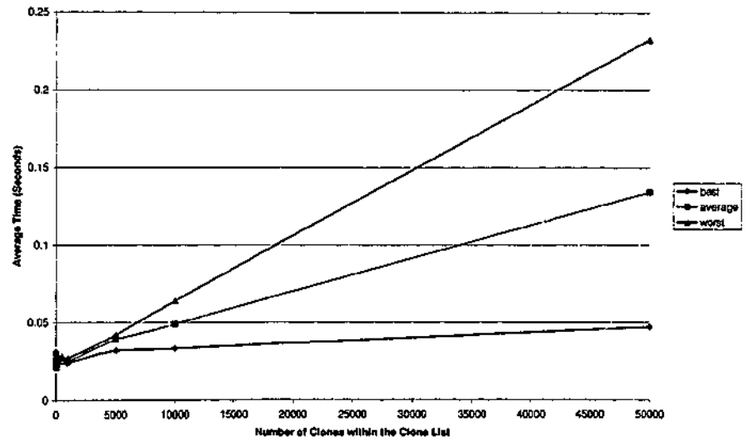

Figure 8.6: Average Time per Operation (Clones)

Figure 8.6 also illustrates that performance is not greatly affected when 50,000 *best* case Clones are implemented. The average time per operation of the *best* case scenario is 0.047 seconds, compared to the *average* case scenario which is 0.134 seconds and *worst* case scenario which was 0.232 seconds. This observation indicates that processes involved in the managing of an operation are not affected by the number of Clones present when the *best* case scenario is applied. Performance overheads are only incurred when necessary, as with *worst* and *average* case scenarios.

The average time per operation as the number of Clones is varied, can be expressed as a linear function. The linear functions and confidences for *best*, *average* and *worst* case scenarios are detailed in table 8.6.

Case	Equation	Confidence
Best	$y = 5E-07x + 0.024$	$R^2 = 0.8815$
Average	$y = 2E-06x + 0.0259$	$R^2 = 0.9958$
Worst	$y = 4E-06x + 0.0252$	$R^2 = 0.9975$

Table 8.6: Average Time per Operation Linear Functions (Clones)

To further analyse the impact upon performance, the average time per operation by the Clone Distribution Manager to search and retrieve a specific Clone from within the Clone List was measured. The average time per operation of the Clone Distribution Manager is illustrated in figure 8.7. The results also show a linear relationship between the time and number of Clones (table 8.7).

Case	Equation	Confidence
Best	$y = 5E-08x + 0.0018$	$R^2 = 0.8219$
Average	$y = 9E-07x + 0.024$	$R^2 = 0.9967$
Worst	$y = 2E-06x + 0.0019$	$R^2 = 0.9998$

Table 8.7: Average Time per Operation for the Clone Distribution Manager Linear Function (Clones)

To further analyse the effect of the Clone Distribution Manager on the overall average time per operation, the percentage of time spent by the Clone Distribution Manager is compared to the total time taken to process an operation (figure 8.8). The results indicate that the impact upon overall performance is minimal (about 10% to 12%). Only when a very large number of Clones (5,000

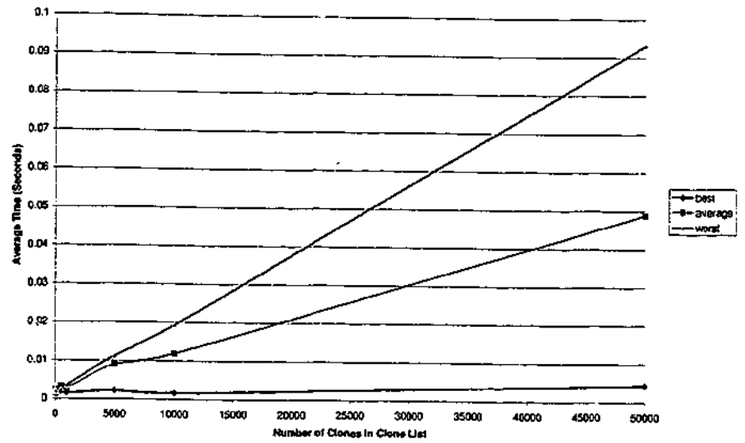

Figure 8.7: Average Time per Operation for the Clone Distribution Manager (Clones)

or greater implemented as *average* or *worst* case scenario) does the impact on overall performance become apparent.

8.4.2 Scaling the number of Relationship Components

Average Time per Operation

The results of the average time per operation when scaling the number of Relationship Components (including *best*, *average* and *worst* case scenarios, implementing both *singleton* and *new instance*) are detailed in table 8.8 and figures 8.9 and 8.10. These results indicate that the overall performance impact of multiple Relationship Components is exponential (table 8.9 details their equations and confidence values).

This exponential effect is only really observed when 50 or more Relationship Components are implemented, regardless of the Relationship Component's instantiation type. When one to 50 Relationship Components are observed in isolation, a combination of linear and constant times results. This is best represented when reviewing the time taken to process one *best* case *new instance* Relationship Component (0.023 seconds), compared to 50 *best* case *new instance* Relationship Components (0.027 seconds). This indicates that the data structure used to store and find Relationship Components is tailored to a smaller

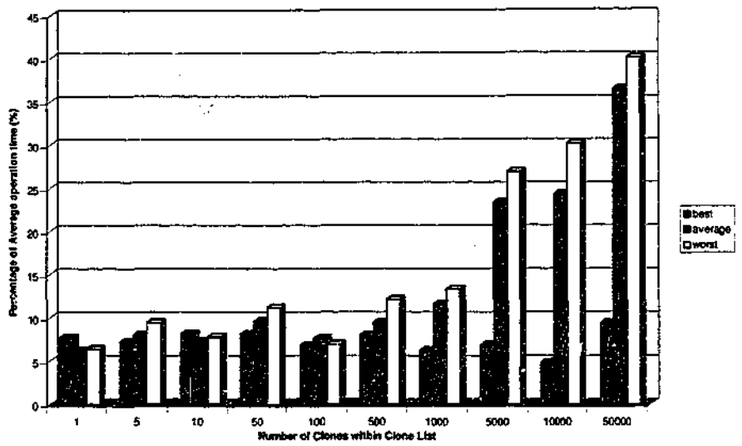

Figure 8.8: Percentage of Average Operation Time taken by the Clone Distribution Manager

No. of	Singleton			New Instance		
RCs	best	average	worst	best	average	worst
1	0.02	0.0271	0.0238	0.0213	0.027	0.026
5	0.0233	0.026	0.023	0.0254	0.0246	0.0239
10	0.0206	0.034	0.0259	0.023	0.0397	0.0254
50	0.0274	0.0406	0.0522	0.0276	0.0431	0.044
100	0.0384	0.0631	0.0777	0.0428	0.0627	0.0751
500	0.3604	0.49	0.5605	0.3728	0.4731	0.6097
1000	1.375	1.736	1.869	1.367	1.734	1.953

Table 8.8: Average Time per Operation for *Singleton* and *New Instance* Relationship Components

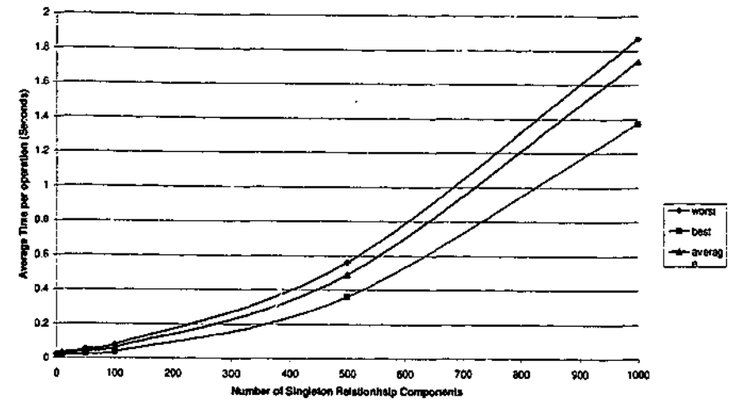

Figure 8.9: Average Time per Operation for *Singleton* Relationship Components

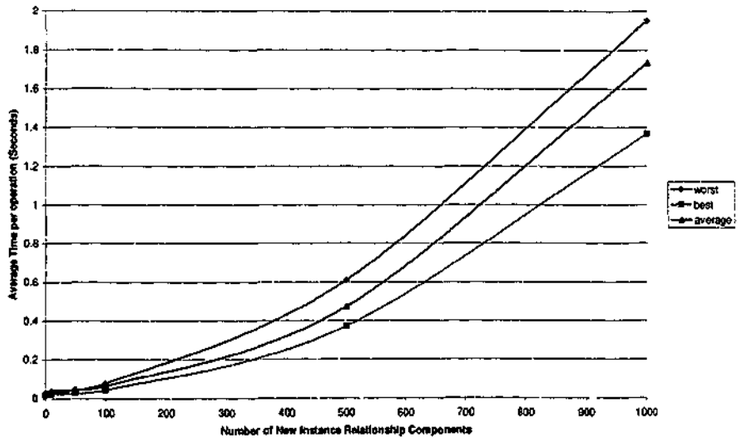

Figure 8.10: Average Time per Operation for *New Instance* Relationship Components

Instantiation Type	Case	Equation	Confidence
New Instance	Best	$y = 0.0249e^{0.0043x}$	$R^2 = 0.9719$
	Average	$y = 0.034e^{0.0042x}$	$R^2 = 0.9637$
	Worst	$y = 0.032e^{0.0045x}$	$R^2 = 0.9446$
Singleton	Best	$y = 0.023e^{0.0044x}$	$R^2 = 0.973$
	Average	$y = 0.033e^{0.0043x}$	$R^2 = 0.9653$
	Worst	$y = .00325e^{0.0044x}$	$R^2 = 0.9391$

Table 8.9: Average Time per Operation Exponential Functions (Relationship Components)

subset of elements. When the number of elements increases, the inadequacy of this data structure is illustrated via a relative increase in processing times.

Time of *Relationship Component Processing*

To further evaluate the average time per operation, the time associated with *Relationship Component Processing* is isolated. The resulting time of *Relationship Component Processing* (table 8.10), show a similar exponential effect (table 8.11) as the average time per operation. These values have not been averaged over 100 operations as with the previous experiment. Instead, they directly illustrate the actual time taken by *Relationship Component Processing*.

No. Of	Singleton			New Instance		
RCs	best	average	worst	best	average	worst
1	0.4307	0.4807	0.3304	0.3304	0.3605	0.6090
5	0.3305	0.37	0.4006	0.3505	0.3705	0.3905
10	0.3405	0.3805	0.4306	0.3705	0.4	0.4507
50	0.7411	0.831	0.8913	0.7711	0.8813	0.8912
100	1.9228	2.163	2.1231	2.042	2.0936	2.2933
500	34.43	39.877	39.076	35.921	38.275	42.6613
1000	135.935	154.45	156.031	135.174	156.334	162.1932

Table 8.10: Time of *Relationship Component Processing* for both *Singleton* and *New Instance* Relationship Components

Unlike the average time per operation, the exponential effect is not exhibited until 100 or more Relationship Components are instantiated. From 50 to 100 Relationship Components, the time taken only doubles, in accordance with the proportional increase in Relationship Components (linear). However, from 100 to 500 Relationship Components, the time taken increases by approximately a factor of 20 (Relationship Components only increase by a factor of 5).

Instantiation Type	Case	Equation	Confidence
New Instance	Best	$y = 0.5244e^{0.0062x}$	$R^2 = 0.9217$
	Average	$y = 0.5628e^{0.0062x}$	$R^2 = 0.927$
	Worst	$y = 0.666e^{0.0061x}$	$R^2 = 0.927$
Singleton	Best	$y = 0.5264e^{0.0062x}$	$R^2 = 0.9274$
	Average	$y = 0.5902e^{0.0062x}$	$R^2 = 0.926$
	Worst	$y = 0.57395e^{0.0062x}$	$R^2 = 0.926$

Table 8.11: Average Time of *Relationship Component Processing* per Operation, Exponential Functions (Relationship Components)

Time of Partial *Relationship Component Processing*

As stated in section 5.2.6, the *Relationship Component Processing* was designed to impose additional cost only when necessary. So that if the current scenario of an existing Relationship Component is valid, then the *Relationship Component Processing* stops early. Figures 8.11 and 8.12 illustrate the average time per operation when only partial *Relationship Component Processing* for both *singleton* and *new instance* Relationship Components is usded. In addition to *best*, *average* and *worst* case scenarios, a *random* case scenario is added. A *random* case scenario illustrates a Relationship Component that randomly changes its validity. It is primarily used to highlight partial *Relationship Component Processing*.

The results in table 8.12 show that partial *Relationship Component Processing* are either linear (*worst* and *average* case scenarios) or constant (*best* and *random* case scenarios). These results are critical in illustrating the scalability of the GLOMAR middleware, as they illustrate that even with a large number of input parameters, the effect of GLOMAR is minor (in a stable environment).

Time of Instantiation

This experiment determines the effect of the different instantiation types (*singleton* and *new instance*) upon performance. A comparison of different instantiation types for the *best*, *average* and *worst* case scenarios as seen in earlier experiments fails to determine any performance impact associated with average

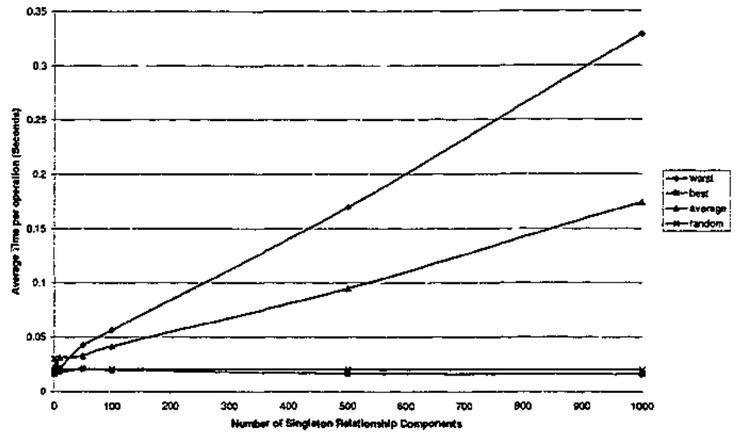

Figure 8.11: Average Time per Operation for *Singleton* Relationship Components when only Partial *Relationship Component Processing* is invoked

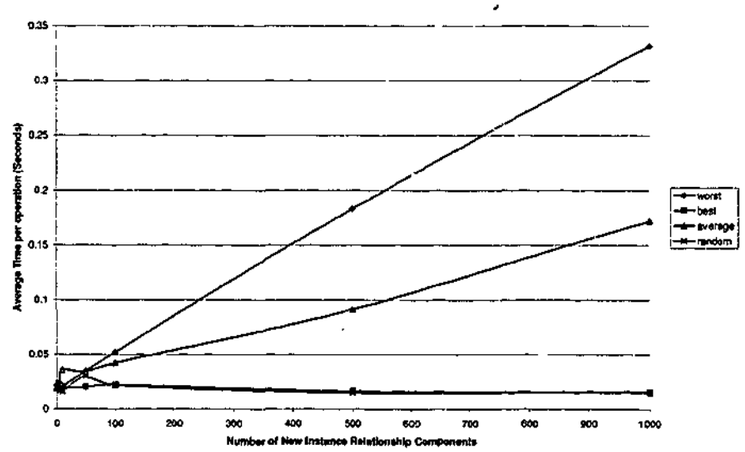

Figure 8.12: Average Time per Operation for *New Instance* Relationship Components when only Partial *Relationship Component Processing* is invoked

Instantiation Type	Case	Equation	Confidence
New Instance	Best	$y = 0.0191$	
	Average	$y = 0.0001x + 0.0263$	$R^2 = 0.991$
	Worst	$y = 0.0003x + 0.0198$	$R^2 = 0.9994$
	Random	$y = 0.0194$	
Singleton	Best	$y = 0.0178$	
	Average	$y = 0.0001x + 0.0251$	$R^2 = 0.9975$
	Worst	$y = 0.0003x + 0.029$	$R^2 = 0.9986$
	Random	$y = 0.0217$	

Table 8.12: Average Time per Operation Equations when only Partial *Relationship Component Processing* is invoked

time per operation. Primarily, this is because the *Relationship Component Processing* is called only once by the first operation, with the remaining operations absorbing the cost (as the cost is averaged over 100 operations).

However, with the *random* case scenario, the impact becomes more noticeable. Figure 8.13 demonstrates that as the number of Relationship Components increases, the cost associated with creating a *new instance* type, impacts greatly upon performance compared to that of a *singleton* type. However, this impact is only observed when the number of Relationship Components is greater than 100.

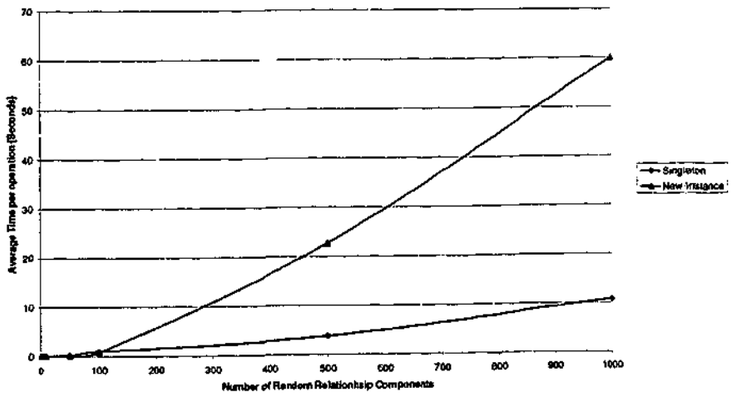

Figure 8.13: Instantiation Time with *Singleton* and *New Instance* Relationship Components within a *Random* Case Scenario

8.5 Conclusion

The results from all these experiments determine the cost of using the GLOMAR middleware layer. The following section outlines the suitability of the proposed and implemented algorithms, as well as the effect of scaling the number of Relationship Components and Clones. Finally, a threshold for Clones and Relationship Component numbers is determined.

8.5.1 Initiation of the GLOMAR middleware layer

The linear relationship (section 8.3) for both memory consumption and time of the initiation of the GLOMAR middleware layer would suggest that GLOMAR could adequately handle many Clones and Relationship Components. Assuming memory is abundant [1] and waiting time during the initiation of the GLOMAR middleware layer is acceptable, then a recommended limit on the number of Clones and Relationship Components is not necessary. Thus, GLOMAR's initiation is a suitable mechanism that does not greatly affect the performance of the DFS.

However, it should be noted that Clones and Relationship Components do not affect performance and resource usage equally, as Relationship Components impact greater on the overall performance than Clones. For example, 50 Clones add 2.3 seconds to initiation, whereas 50 Relationship Components add 4.8 seconds. In addition, these results only reflect Relationship Components that have minimal actual consistency maintenance functionality packaged inside. Realistically, Relationship Component implementations would have additional application-specific code that would effect the time and memory consumption of the GLOMAR middleware layer during its initiation.

8.5.2 Processing an Operation

The initiation of the GLOMAR middleware layer is less important to the overall usage of GLOMAR when compared to processing of operations. Unlike the initiation of the GLOMAR middleware layer where waiting for extended periods is acceptable, the ramification of time delays upon the processing of an operation is critical to GLOMAR's efficiency.

In relation to the scaling of the number of Clones, the results would suggest that one to 5,000 Clones have an equal effect upon performance (section 8.4.1). For

[1] This is based on the assumption that GLOMAR exists on a desktop or server type device

any number larger than 5,000 Clones, there is a noticeable impact on performance. Realistically, the cost is only minimal, with 50,000 Clones only adding approximately 0.9 seconds per operation. However, 5,000 or less Clones would be a recommended quantity.

The effects of scaling the number of Relationship Components have far wider implications for average time per operation than with Clones. This is because the cost of using a Relationship Component is far greater than that of a Clone. This is since the cost associated with processing a Relationship Component is more than that of a Clone.

The performance of the average time per operation (section 8.4.2) changes from linear to exponential when the number of Relationship Components exceeds 50 to 100 implementations. Implementing one to 50 Relationship Components shows no quantifiable difference in performance, as the time in these cases is approximately 0.03 seconds per operation. Thus, 50 or less Relationship Components would be a recommended quantity. Implementing 50 to 100 Relationship Components has a small impact upon performance, but in certain situations, this might be acceptable (the cost is approximately 0.06 seconds per operation). From 100 to 1,000 Relationship Components, the average time per operation outweighs any benefits had by the multiple consistency model approach. In this case, the average time per operation for a *best* case scenario with 1,000 Relationship Components is 0.37 seconds. This performance degradation impacts the scalability of the GLOMAR middleware layer and is unacceptable for most DFS implementations. However, the likelihood of 100 Relationship Components or more being simultaneously implemented is very low.

These recommended quantities for Clones and Relationship Components are because operations result in all events within the *Relationship Component Processing* being invoked. In reality, the *Relationship Component Processing* has been designed to only implement as much functionality as necessary to fulfil the task of selecting the appropriate Relationship Component. As a result, and depending on the situation, additional Relationship Components and Clones could be implemented without incurring noticeable cost. This was illustrated in the fact that the impact of partial *Relationship Component Processing* is minimal (section 8.4.2).

The cost associated with instantiation types shows an additional cost coupled with the implementation of a *new instance* Relationship Component (section 8.4.2). However, this cost increases considerably in an environment where the

scenario is constantly changing (and thus Relationship Components are constantly stopped and started). Rather, if the scenario is relatively stable, then there is no major impact on performance because of the different instantiation types. However, if the scenario is unstable, then the implication for performance is noticeable.

Currently, the implementation targets a mid range of input parameters, primarily focusing on demonstrating the feasibility of the multiple consistency model approach. However, the GLOMAR middleware layer can be fine-tuned to focus on performance. This is not to say that the current implementation fails to address performance issues. However, different implementers might choose to implement different data structures and container types more attune to the requirements of a specific implementer or system. An illustration of this is the poor performance seen as a result of a large number of Clones. An obvious solution would be a more appropriate data structure than a serial list (XML) used to represent Clones.

8.6 Overall

The results from these experiments show the efficiency of the GLOMAR middleware layer, in the face of varying numbers of Relationship Components and Clones. These quantifiable results, in conjunction with the qualitative analysis from Chapter 7 demonstrate GLOMAR as both an efficient and effective implementation of the multiple consistency model approach.

8.7 Summary

This chapter has discussed the cost issues associated with the .NET implementation of the GLOMAR middleware layer. The experiments showed how scaling of two input parameters (the number of Clones and the number of Relationship Components) affect the GLOMAR middleware layer. From these experiments, the cost of the initiation of the GLOMAR middleware layer and the cost of processing an operation were determined. The results outline the potential for GLOMAR to scale (up to a particular point), a recommended number of Relationship Components and Clones and validates the proposed and developed algorithms to support the multiple consistency model approach. The next chapter concludes this thesis, outlining the major contributions and future work.

Chapter 9

Conclusion

This dissertation discusses the motivation, conceptual architecture, design, implementation and evaluation of a component-based framework for maintaining consistency of data objects within a heterogeneous DFS, called GLOMAR.

The motivation for GLOMAR stems from a number of issues associated with the implementation of consistency maintenance and concurrency control within a DFS.

Existing approaches primarily focus on servicing the consistency and concurrency needs of a single scenario only, whether that being user, hardware or application-specific. However, current DFS environments exhibit more than a single scenario. Rather, they are an amalgamation of multiple scenarios coexisting simultaneously. For this reason, an approach targeted specifically at a single scenario to manage consistency maintenance and concurrency control does not service the needs of current or future DFSs.

This dissertation proposes and implements a component-based framework to facilitate the servicing of multiple scenarios within a single DFS implementation concurrently. This is primarily achieved by abstracting the consistency maintenance and concurrency control mechanisms from the operating system and/or application, then re-implementing them as components. Depending upon the scenario experienced by the DFS, the appropriate consistency maintenance and concurrency control mechanism is invoked. This not only provides of level of adaptation for the DFS, but enough flexibility to allow additional consistency maintenance and concurrency control mechanisms to be added, when necessary.

The elements of the GLOMAR framework includes the component to house the consistency maintenance and concurrency control mechanism (Relationship Components) and the middleware layer used to manage them at run-time.

This dissertation also analysed both the Relationship Component and middleware layer to determine the efficiency and effectiveness of the implementation. This analysis consisted of three case studies (two generic Relationship Components and one application-specific Relationship Component), each tailored for a different scenario. These case studies, in conjunction with an analysis of the efficiency of the GLOMAR middleware layer, demonstrate the flexibility and feasibility of the GLOMAR framework in relation to the management of different consistency maintenance and concurrency control mechanisms.

9.1 Contribution of this Dissertation

From review of the literature, this dissertation is the first attempt to propose, design and implement a middleware layer, coupled with a component-based framework, to manage consistency maintenance and concurrency control, as implemented within a DFS, using the multiple consistency model approach. The contribution of this work is summarised below.

The contribution of GLOMAR includes:

- **A multiple consistency model approach to support diverse applications and scenarios running on top of a DFS was proposed.** The GLOMAR framework illustrates how different scenarios and multiple solutions could be defined and serviced. This met the aim of supporting multiple concurrency control and consistency maintenance within a heterogeneous environment.

- **Consistency maintenance and concurrency control functionality was specified and abstracted into a Relationship Component.** Primarily, the case studies in Chapter 7 showed how different consistency models were implemented within the Relationship Component. All three case studies were unique, whether being simple, complex, or application-specific. However, each consistency maintenance and concurrency control mechanism and associated elements were successfully encapsulated within the Relationship Component. This met the aim of being able to streamline and thus encourage the development of Relationship Components such that new and unique implementations were achieved.

- **The flexible and effective design of the Relationship Component enabled current and future consistency maintenance and concurrency control functionality to be supported.** The varying

nature and structure of the case studies detailed in Chapter 7, illustrate the flexibility and effectiveness of the design of the Relationship Component. Not only were the scenarios for which they were valid able to be defined, but their scope and functionality were not restricted. This is illustrated in the *ROWA* Relationship Component implementation, as it was targeted to be a generic concurrency control and consistency maintenance mechanism. This is opposed to the Outlook Relationship Component implementation, which was targeted at a single application and a specific data set. The Twin Transaction Model Relationship Component illustrated the implementation of an existing consistency model, showing how it could be encapsulated as a Relationship Component. This met GLOMAR's aim of flexibility, configurability and the motivation to support fine grain concurrency control and consistency maintenance mechanisms within a DFS.

- **The GLOMAR framework demonstrates how software engineering practices can be integrated into the construction of consistency maintenance and concurrency control functionality.** The process of constructing consistency maintenance and concurrency control functionality was streamlined via well-defined components, interfaces and tools. This impacted positively on Relationship Component creation, as code was shared across implementations and Relationship Components were deployed simply. A direct illustration of this was the sharing of the Relationship Scope for the Twin Transaction Model Relationship Component and the *ROWA* Relationship Component. Thus, the aim of exploiting software engineering practices was illustrated via the many Relationship Component implementations.

- **The feasibility of the multiple consistency model approach was demonstrated with a full-scale implementation.** The GLOMAR framework was used to govern the consistency of files within a real life system. It was actively used to ensure that replicas of files where up-to-date within a small DFS.

- **The flexibility of GLOMAR in regards to diverse Relationship Component implementations was demonstrated.** Each Relationship Component implementation used different techniques and different functionality to achieve their concurrency control and consistency maintenance requirements. Of particular interest was the wide range of techniques used by all the Relationship Components. These including, web

service technologies (SOAP and WSDL), client-server and peer-to-peer architectures and application-specific functionality (using the Outlook COM object). This demonstrated how the aim of flexibility and heterogeneity were supported, as the implementation possibilities were not restricted.

- **A balance has been achieved between consistency and resources usage through implementation of appropriate consistency models.** The ability to balance consistency and resource usage stems from the motivation to provide fine grain support for concurrency control and consistency maintenance. The Outlook Relationship Component demonstrated this with an application-specific Relationship Component, exploiting application-specific data and events. This resulted in a very fine grain level of consistency maintenance being achieved, without greatly compromising resource usage (in this case bandwidth).

- **Analysis of the scalability of the GLOMAR middleware layer was demonstrated.** The evaluation within Chapter 8 illustrated that the perceived cost of implementing the middleware, in turn the multiple consistency model approach was negligible. The results showed that when implementing a realistic number of Clones (5,000) and Relationship Components (50), the GLOMAR middleware layer does not greatly impact performance such that the beneficial qualities of the multiple consistency model approach were compromised.

- **To support the Relationship Component development methodology (thus the creation of consistency maintenance and concurrency control mechanisms), class libraries, types and Administration Console have been created.** The class libraries, types and Administration Console supplied with the GLOMAR framework made the creation and editing of Relationship Components more efficient. This usage was illustrated by the Relationship Component implementations defined in Chapter 7, as new and existing concurrency control and consistency maintenance functionality were easily added to GLOMAR. Thus, by defining types and classes to be used, a development methodology and specification for the creation of consistency maintenance and concurrency control mechanisms resulted.

- **To provide the infrastructure to support Relationship Components.** An aim of GLOMAR was to focus development on the differing aspects of a DFS. This was achieved via two mechanisms. The first was via the Relationship Component, and its specific scoping, to only service

concurrency control and consistency maintenance. The second was via the services found within the GLOMAR middleware layer. The unique aspect of the GLOMAR middleware layer was the ability to extend easily these services, such that generic services and/or Relationship Component specific services could be implemented. As a result, this provided the mechanisms to support Relationship Component implementations. As shown in Chapter 7 and was specifically illustrated in the Twin Transaction Model Relationship Component, Relationship Component specific services were added to enhance the capabilities of the Twin Transaction Model Relationship Component.

9.2 Future Work

Much of the possible future work arises from applying the GLOMAR framework to platforms other than a DFS. For example, if the multiple consistency model approach were applied to other platforms (DDBMS or DSM) then expansion work would require overhauling some of GLOMAR's major design features. The lack of support for transaction semantics is one case in point. Thus, an area of future work would involve porting GLOMAR's approach to different distributed systems platforms.

The steps within GLOMAR's *Relationship Component Processing* (section 5.2.6) intends to provide a balance between cost (time) and Relationship Component selection. However, in certain situations, the focus upon performance affects the Relationship Component selection to the detriment of the application and/or user. Thus future work might consider improving *Relationship Component Processing*, such that a balance can be structured between time imposed to process a selection and the appropriateness of that selection.

One example of the limitations of *Relationship Component Processing* is the lack of structure between related Relationship Components. The main concern is since no formal structure has been defined between related Relationship Component (for example, a connected and disconnected consistency model), there is no way to ensure that appropriate components interact. A solution could be to redesign *Relationship Component Processing* and extend the Relationship Component's metadata, such that Relationship Components can be loosely coupled. In other words, so when it is time to select an appropriate Relationship Component to implement, preferential treatment is given to any related components. However, an issue like how this is to be implemented and

how this relationship is represented, such that flexibility is not reduced, makes this a difficult undertaking.

Finally, areas of future work could also include improving the current Context Provider implementations and implementing GLOMAR using a native code compiler. The purpose of both approaches would serve only to improve the performance of GLOMAR and nothing more. However, this benefit is unsubstantiated at this point.

9.3 Final Remarks

This dissertation has shown an effective and efficient mechanism to support true heterogeneity (whether being hardware, software or user), using a scenario based approach that encapsulates multiple concurrency control and consistency maintenance mechanisms concurrently. This dissertation has also proposed, developed and illustrated the ability to apply a component-oriented architecture to concurrency control and consistency maintenance functionality within a DFS, such that the scope and granularity of consistency can be adjusted accordingly.

Glossary

.NET Remoting Infrastructure The Remoting Infrastructure enables communication between objects in different application domains or processes using different transportation protocols.

API Application program interface.

Availability The process used to improve the availability of data objects.

COM Microsoft's Component Object Model

COM Add-In A COM based module used to extending existing applications. For example, extending any of the Microsoft Office applications.

Component A binary unit of independent production, acquisition and deployment that interact to form a functioning system (Szyperski 1997).

Concurrency Control Concurrency control is the process used to identify and resolve updates to data that are made by multiple users simultaneously.

Consistency Maintenance Consistency maintenance is the process of ensuring that events that manipulate data on one replica are visible on all others, thus making them correct.

consistency model (lower case) The consistency model is the concurrency control and consistency maintenance mechanism.

Consistency Model (upper case). This sub-component contains a single concurrency control and consistency maintenance mechanism (in other words a consistency model).

Context Provider The mechanisms used to derive specific information about the current status of the system.

Clone Within GLOMAR, data objects and encapsulating types are referred to as Clones. The reason for using this term is that GLOMAR views distributed data objects as clones of each other

Clone Distribution Manager The GLOMAR middleware layer service that manages all Clone related activities. These activities mainly include name resolution.

Clone List This sub-component defines the governed data objects.

DDBMS *See* Distributed Database Management System.

DFS *See* Distributed File System.

DSM *See* Distributed Shared Memory.

Distributed Database Management System A Distributed Database Management System is a database management system that replicates data objects to improve availability and performance.

Distributed File System A Distributed File System is a file system that replicates data objects to improve availability and performance.

Distributed Shared Memory A Distributed Shared Memory is a memory system that replicates data objects to improve availability and performance.

Distributed System A Distributed System comprises a number of computing nodes connected via a communication network, each with a supportive operating system. Messages are then passed between nodes to facilitate the sharing of resources

Environmental Object The Environmental object contains a collection of scenario types and actual values, which are then passed into the Relationship Scope of each Relationship Component. It is also referred to as the shared data structure.

Executive The GLOMAR middleware layer service that manages file operations and Relationship Component implementations at run-time.

File System The File System is a collection of data objects that are persistent until explicitly destroyed. File systems support four fundamental issues, *naming structure*, *programming interface*, *physical mapping* and *integrity*.

GLOMAR A Component Based Framework for Maintaining Consistency of Data Objects within a Heterogeneous Distributed File System.

GLOMAR Middleware Layer The middleware layer used to manage the run-time implementation of Relationship Components.

GLOMAR Taxonomy The taxonomy used within GLOMAR for classifying scenarios. The current taxonomy is based on *System*, *User* and *File* profiles.

Heterogeneous Environment A system that exhibits all manner of hardware, software and users collectively and concurrently.

IL *See* Intermediate Language.

Local Operation Interface The GLOMAR middleware layer service that received file operations from the local system.

Multiple Consistency Model Approach The approach where numerous consistency models are available for implementation by a DFS.

New Instance Relationship Component A Relationship Component that is created dynamically at run-time.

One-copy equivalence Data objects must be perceived as centralised, but implemented distributed.

Operating System An Operating System is the software that an application, uses to communicate with the physical part of the computer.

Remote Operation Interface The GLOMAR middleware layer service that received file operations from a remote system.

Reflection Reflection is the mechanism of discovering class information and instantiating classes solely at run-time.

Relationship Component The component contains the concurrency control and consistency maintenance functionality. In addition, it contains the description of the scenario for which this component is valid. Contains the sub-components, Consistency Model, Relationship Scope and Clone List.

Relationship Component Processing The name of the process of selecting the appropriate Relationship Component to invoke based on the current scenario.

Relationship Component Repository The GLOMAR middleware layer service that implements and manages all Relationship Component implementations.

Relationship Scope The sub-component that defines the scenario for which it is valid.

Remoted Service A GLOMAR middleware layer or user-specific service that uses the .NET Remoting infrastructure.

Replication Replication is the placement and management of replicated file objects for improving the availability, performance and usability of data objects.

Replica Placement The process of determining the most appropriate location for a replicated data object.

Service Manager The GLOMAR middleware layer service that implements all user-specific services.

Single Consistency Model Approach The approach where a single consistency models is available for implementation by a DFS.

Singleton Relationship Component A Relationship Component that is referenced from an existing Relationship Component.

Scenario A scenario is used to describe the elements and situation that is/can be experienced by a DFS environment.

System Grader The GLOMAR middleware layer service that determines the current scenario.

TTM *See* Twin Transaction Model

Twin Transaction Model The Twin Transaction Model is a consistency model developed by (Rasheed 1999).

URI Universal Resource Identifier.

XML Web Service An XML based method of invocating functionality on remote machine.

Bibliography

Abbadi, D. Skeen, and F. Christian (1985). An efficient fault tolerant protocol for replicated data management. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 25-27, 1985, Portland, Oregon*, pp. 215–229. ACM.

Agrawal, D. and A. E. Abbadi (1990). The tree quorum protocol: An efficient approach for managing replicated data. In D. McLeod, R. Sacks-Davis, and H.-J. Schek (Eds.), *16th International Conference on Very Large Data Bases, August 13-16, Brisbane, Queensland, Australia, Proceedings*, pp. 243–254.

Alonso, R., D. Barbara, and H. Garcia-Molina (1990, Sep). Data caching issues in a information retrieval system. *ACM Transaction on Database Systems 15*(3), 359–384.

Alonso, R., D. Barbará, H. Garcia-Molina, and S. Abad (1988). Quasi-copies: Efficient data sharing for information retrieval systems. In J. W. Schmidt, S. Ceri, and M. Missikoff (Eds.), *Advances in Database Technology - EDBT'88, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988*, Volume 303 of *Lecture Notes in Computer Science*, pp. 443–468. Springer.

Alsberg, P. A. and J. D. Day (1976). A principle for resilient sharing of distributed resources. In *2nd International Conference on Software Engineering, 13-15 October , San Francisco, California*, pp. 627–644.

Apple Computer, I. (1985). *Inside Macintosh, Volume II*. Addison Wesley.

Badrinath, B. R., A. Acharya, and T. Imielinski (1993). Impact of mobility on distributed computations. *Operating Systems Review 27*, 15–20.

Baker, M., J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout (1991, October). Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 198–212. Association for Computing Machinery SIGOPS.

Barbara-Milla, D. and H. Garcia-Molina (1994, April). Replicated data management in mobile environments: Anything new under the sun? In *IFIP Working Conference on Applications in Parallel and Distributed Computing*, pp. 237–246.

Bernstein, P. and N. Goodman (1984). An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transaction on Database Systems 9*(4), 596 – 615.

Bernstein, P. and N. Goodman (1986). Serializability theory for replicated databases. *Journal of Computer and System Sciences 31*(3), 355–374.

Berstein, P., V. Hadzilacos, and N. Goodman (1987). *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

Boling, D. (1998). *Programming Microsoft Windows CE.* Microsoft Press.

Borghoff, U. M. and K. Nast-Kolb (1989). Distributed systems: A comprehensive survey. Technical Report Report No TUM-I8909, Technische Universität München.

Box, D. (1998). *Essential COM.* Object Technology Series. Addison Wesley.

Brown, N. and C. Kindel (1998). Distributed component object model protocol—dcom/1.0. http://www.microsoft.com/com/, microsoft corp, network working group internet draft.

Burns, R. C. and D. D. E. Long (1997). Efficient distributed backup with delta compression. In *I/O in Parallel and Distributed Systems*, pp. 27–36.

Burrows, M. (1988). *Efficient Data Sharing.* Ph. D. thesis, Computer Laboratory, University of Cambridge.

Byrne, R. (2001). *Building Applications with Microsoft Outlook Version 2002.* Microsoft Press.

Caron, R. (2002). Getting started with xml web services in visual studio .net, http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vbtchgettingstartedwithxmlwebservicesinvisualstudionet.asp.

Ceri, S., M. A. W. Houtsma, A. M. Keller, and P. Samarati (1991). A classification of update methods for replicated databases. Technical Report CS-TR-91-1392, Stanford University, Computer Science.

Chandra, B. R. and J. R. Larus (1999). Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering 25*(3).

Comer, D. (1995). *Internetwroking with TCP/IP. Volume I Principles, Protocols and Architecture.* Prentice Hall.

Coulouris, G., J. Dollimore, and T. Kindberg (2001). *Distributed Systems: Concepts and Design.* Addison Wesley.

Cuce, S. (1999, November). Conflict avoidance within a disconnected mobile environment. In *Proceedings of the 6th Australian Conference on Parallel and Real-Time Systems (PARTS99).*

Cuce, S. and A. Zaslavsky (1998a, December). Adaptive cache validation for mobile file systems. In *Lecture Notes in Computer Science, Springer-Verlag, LNCS 1552.*

Cuce, S. and A. Zaslavsky (1998b). Partially consistent cache management model for a mobile environment. In *1st Annual South African Telecommunications, Networks and Applications Conference (SATNAC98).*

Cuce, S. and A. Zaslavsky (2002a, January). Adaptable consistency control mechanism for mobility enabled file system. In *3rd International Conference on Mobile Data Management (MDM 2002), Singapore.*

Cuce, S. and A. Zaslavsky (2002b). Run-time file system consistency support in mobile computing systems. In *2nd Asian International Mobile Computing Conference. (AMOC02) Langkawi, Malaysia.*

Cuce, S., A. Zaslavsky, B. Hu, and J. Rambhia (2002, September). Maintaining consistency of twin transaction model using mobility-enabled distributed file system environment. In *5th International Workshop on Mobility in Databases and Distributed Systems in conjunction with the 13th International Conference on Database and Expert Systems Applications (DEXA'2002). Aix-en-Provence, France.*

Davidson, S. (1982). *An Optimistic Protocol for Partitioned Distributed Database Systems.* Ph. D. thesis, Dept. of EECS, Princeton University.

Demers, A. J., K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch (1994, December 8-9). The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, Santa Cruz, California, pp. 2–7.

DOS (1983). *Disk Operating System, Version 2.1. 1502343.* IBM Corporation.

Dwyer, D. (1998a). *Adaptive File System Consistency for Mobile Computing Environment.* Ph. D. thesis, University of Illinois at Urbana-Champaign.

Dwyer, D. (1998b, September). Adaptive file system consistency for unreliable mobile computing environments. In *IEEE International Computer Performance and Dependability Symposium*, pp. 64–173.

Dwyer, D. and V. Bharghavan (1997, Jan). A mobility-aware file system for partially connected operations. *Operating Systems Review 31*(1), 24–30.

Eager, D. and K. C. Sevcik (1983). Achieving robustness in distributed database systems. *ACM transaction on Database Systems 8*(3), 354–381.

ECMA (2001). Ecma c# and common language infrastructure standards - http://www.ecma.ch/ecma1/stand/ecma-335.htm.

Edwards, W. K., E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer (1997). Designing and implementing asynchronous collaborative applications with bayou. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Asynchronous Collaboration, pp. 119–128.

Faiz, M. (1995). Database replication strategy in mobile computing environment. Master's thesis, Computer Technology, Monash University.

Flenner, R. (2002). *Java P2P Unleashed*. Sams Publishing.

Galli, D. (2000). *Distributed Operating Systems*. Prentice Hall.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns, Elements of Reusable Object-Oriented Softwares*. Addison-Wesley.

Gifford, D. K. (1979). Weighted voting for replicated data. In *Proc 7th Symp. on Operating Systems Principles*, pp. 150–162.

Gill, D. S., S. Zhou, and H. S. Sandhu (1994). A case study of file system workload in a large-scale distributed environment. In *Measurement and Modeling of Computer Systems*, pp. 276–277.

GlowCode.Com (2002). Glowcode.net v4.0 memorydashboard - http://www.glowcode.com/.

GNUTELLA. (2002). http://gnutella.wego.com.

Goodman, N., D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries (1983). A recovery algorithm for a distributed database system. In *In Proceedings of the 2nd ACM Symposium on Principles of Database Systems*, pp. 8–15.

Gosling, J., B. Joy, G. Steele, and G. Bracha (2000). *The Java Language Specification Second Edition*. Boston, Mass.: Addison-Wesley.

Gough, J. and D. Corney (2000, September). Evaluating the java virtual machine as a target for languages other than java. In *Presented to the Joint Modula Languages Conference*, Zurich, Switzerland.

Gray, C. and D. Cheriton (1989). Leases: An efficient fault-tolerant mechanisms for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, pages 202–210.*

Guy, R. G., J. S. Heidemann, W. Mak, J. Thomas W. Page, G. J. Popek, and D. Rothmeir (1990, Summer). Implementation of the ficus replicated file system. In *Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA*, pp. 63–72.

Guy, R. G., P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek (1998). Rumor: Mobile data access through optimistic peer–to–peer replication. In *Lecture Notes in Computer Science - Advances in Database Technologies*, pp. 254–265. Springer-Verlag.

Haerder, T. and A. Reuter (1983). Principles of transaction-oriented database recovery. *ACM Computer Surveys 15*(4), 151–166.

Helal, A. Heddaya, and B. Bhargava (1996). *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers.

Honeyman, P. and L. Huston (1995). Communication and consistency in mobile file systems. *IEEE Personal Communications 6*(2).

Horton, M. and R. Adams (1995). Rfc 1036 - standard for interchange of usenet messages. http://www.faqs.org/rfcs/rfc1036.html.

Howard, J. (1988). An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference. Feb. 1988, Dallas, TX.*

Huston, L. B. and P. Honeyman (1993, 2–3). Disconnected operation for AFS. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, Cambridge, MA, pp. 1–10.

IRDA (2002). Irda guidelines. http://www.irda.org/standards/guidelines.asp.

Kistler, J. J. (1993). *Disconnected Operation in a Distributed File System*. Ph. D. thesis, School of Computer Science, Carnegie Mellon University.

Kistler, J. J. and M. Satyanarayanan (1991). Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, Volume 25, Asilomar Conference Center, Pacific Grove, U.S., pp. 213–225. ACM Press.

Kuenning, G. (1994, December). Design of the SEER predictive caching scheme. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US.

Kumar, A. (1991, May). A randomised voting algorithm. In *IEEE 10th International Conf. On Distributed Computing Systems. Arlington, TX*, pp. 412–419.

Kumar, A. and A. Segev (1988). Optimising voting-type algorithms for replicated data. In J. W. Schmidt, S. Ceri, and M. Missikoff (Eds.), *Advances in Database Technology - EDBT'88, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988*, Volume 303 of *Lecture Notes in Computer Science*. Springer.

Kumar, P. (1994). *Mitgating the Effects of Optimistic Replication in Distributed File System*. Ph. D. thesis, School of Computer Science, Carnegie Mellon University.

Kumar, P. and M. Satyanarayanan (1995, January). Flexible and safe resolution of file conflicts. In USENIX Association (Ed.), *Proceedings of the 1995 USENIX Technical Conference: January 16-20, 1995, New Orleans, Louisiana, USA*, Berkeley, CA, USA, pp. 95–106 (or 95–105). USENIX.

Kung, H. T. and J. T. Robinson (1981). On optimistic methods for concurrency control. *ACM Transaction of Database systems 6*(2).

Lampson, B. and H. Sturgis (1976). Crash recovery in a distributed system. Technical report, Xerox, Palo Alto Research Center.

Lei, H. and D. Duchamp (1997, January). An analytical approach to file prefetching. In USENIX (Ed.), *1997 Annual Technical Conference, January 6-10, 1997. Anaheim, CA*, Berkeley, CA, USA, pp. 275–288. USENIX.

Levy, E. and A. Silberschatz (1990, Dec). Distributed file systems: Concepts and examples. *ACM Computing Surveys 22*(4), 321–374.

Lippman, S. and J. Lajoie (1998). *C++ Primer*. Addison Wesley.

Liskov, B., S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams (1991). Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 226–38. Association for Computing Machinery SIGOPS.

Liu, G. and G. Q. M. Jr. (1995, November). A predictive mobility management algorithm for wireless mobile computing and communications. In

IEEE International Conference on Universal Personal Communications (ICUPC'95), Tokyo, Japan.

Liu, G. and G. Q. Maguire (1994). A survey of caching and prefetching techniques in distributed systems. Technical report, TRITA-IT R 94:40, Royal Institute of Technology (KTH), Department of Teleinformatics, Telecommunication Systems Laboratory.

Lu, S., K.-W. Lee, and V. Bharghavan (1997). Adaptive service in mobile computing environments. In *Proc. 5th IFIP Int'l Wksp on QoS*.

Macdonald, R. (2001). Understanding assemblies - http://msdn.microsoft.com/library/en-us/dnvbdev01/html/vb01g10.asp.

Mann, T., A. Birrell, A. Hisgen, C. Jerian, and G. Swart (1994, May). A coherent distributed file cache with directory write-behind. *ACM Transaction of Computer Systems 12*(2).

Marzullo, K. and F. Schmuck (1988). Supplying high availability with a standard network file system. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, Washington, DC, pp. 447–455. IEEE Computer Society.

Microsoft (1998). Programmer's guide for activesync.

Microsoft (2000). Performance monitoring, http://msdn.microsoft.com/library/en-us/exchserv/html/admnfunc_50mf.asp.

Microsoft (2002a). http://www.microsoft.com/windowsxp/default.asp.

Microsoft (2002b). .net for smart devices - http://www.gotdotnet.com/team/netcf/.

Minoura, T. and G. Wiederhold (1982, May). Resilient extended true copy token scheme for a distributed database. *IEEE Transaction of Software Engineering 5*(9), 173–189.

Mitchell, J. and J. Dion (1982). A comparison of two network-based file servers. *Communication of the ACM 25*(4), 233–245.

Morris, J., M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith (1986). Andrew: A distributed personal computing environment. *Communications of the ACM 29*(4), 184–201.

Mukherjee, A. and D. P. Siewiorek (1994, December). Mobility: A medium for computation, communication and control. In *IEEE Workshop on Mobile Computing Systems and Applications*.

Mummert, L., M. Ebling, and M. Satyanarayanan (1995). Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain Resort, CO.*

Nagar, R. (1997). *Windows NT File System Internals; A Developer's Guide.* O'Reilly.

Nelson, M. N., B. B. Welch, and J. K. Ousterhout (1988). Caching in the Sprite network file system. *ACM Transactions on Computer Systems 6*(1), 134–154.

Netscape (1998). Netscape communicator plug-in guide http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm.

Noble, B. (2000, Feb). System support for mobile, adaptive applications. *IEEE Personal Communications 7*(1).

Object Management Group, I. (1999). The common object request broker: Architecture and specification. minor revision 2.3.1.

Oki, B. M. and B. H. Liskov (1988, Aug). Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proc. 7th ACM Symp. on Principles of Distributed Computing. Toronto, Ontario*, pp. 8–17.

Oney, W. (1999). *Programming the Microsoft Windows Driver Model.* Microsoft Press.

Ousterhout, J. K., A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch (1988). The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314 21*(2), 23–36.

Ousterhout, J. K., H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson (1985). Driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th Symposium on Operating System Principles, Orcas Island, WA*, pp. 15–24.

Ozsu, T. and P. Valduriez (1991). *Principles of Distributed Database Systems.* Prentice Hall.

Page, T., R. Guy, J. Heidemann, D. Ratner, P. Reiher, A. Goel, G. Kuenning, and G. Popek (1998, February). Perspectives on optimistically replicated, peer–to–peer filing. *Software Practice and Experience 28*(2), 155–180.

Parker, S., G. Popek, G. Rudisin, B. W. Allen Stoughton, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline (1983). Detection of mutual

inconsistency in distributed systems. *Transactions on Software Engineering 9*(3), 240–246.

Popek, G. and B. Walker (1985). *The LOCUS Distributed System Architecture.* MIT Press.

Psion (2002). http://www.psion.com/.

Rasheed, A. (1999). *Twin-Transaction Model to Support Mobile Data Access.* Ph. D. thesis, School of Computer Science and Software Engineering, Monash University.

Ratner, D. (1995). Selective replication : Fine-grain control of replicated files. Master's thesis, University of California.

Ratner, D. (1998). *Roam: A scalable replication system for mobile and distributed computing.* Ph. D. thesis, University of California.

Ratner, D., G. J. Popek, and P. Reiher (1996, July). Peer replication with selective control. Technical Report CSD-960031, University of California, Los Angeles.

Ratner, D., P. Reiher, and G. Popek (1996, Oct). The ward model: A scalable replication architecture for mobility. In *OOPSLA'96 Workshop on Object Replication and Mobile Computing (ORMC'96), San Jose, California.*

Ratner, D., P. Reiher, and G. Popek (1997). Dynamic version vector maintenance. Technical Report CSD–970041, University of California.

Reiher, P., J. Heidemann, D. Ratner, G. Skinner, and G. J. Popek (1994, Summer). Resolving file conflicts in the Ficus file system. In USENIX Association (Ed.), *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, Berkeley, CA, USA, pp. 183–195. USENIX.

Rhodes, N. and J. Mckeehan (1999). *Palm Programming.* O'Reilly.

Rice, F. (2000). Building a com add-in for microsoft office xp using microsoft visual basic 6.0 - http://msdn.microsoft.com/library/en-us/dnoxpta/html/odc_comaddinvb6.asp.

Ritchie, M. and K. Thompson (1978). The unix time-sharing system. *The Bell System Technical Journal 57*(6), 1905–1929.

Roselli, D., J. Lorch, and T. Anderson (2000, June). A comparison of file system workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, San Doego, California, USA.

Rosenthal, D. S. H. (1990, Summer). Evolving the Vnode interface. In USENIX Association (Ed.), *Proceedings of the Summer 1990 USENIX Conference: June 11–15, 1990, Anaheim, California, USA*, Berkeley, CA, USA, pp. 107–118. USENIX.

Roxio (2002). http://www.roxio.com/en/products/ecdc/dcdfeatures.jhtml.

Saito, Y. and M. Shapiro (2002). Replication: Optimistic approaches. Technical report, HP Laboratories, Palo Alto. HPL-2002-33.

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon (1985). Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX Technical Conference, USENIX Assoc., Berkeley, Calif*, pp. 119–130.

Satyanarayanan, M. (1989). Coda: A highly available file system for a distributed workstation environment. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems. Pacific Grove, CA.*

Satyanarayanan, M. (1990). A survey of distributed file systems. *Annu. Rev. Computer Science 4*, 73–104.

Satyanarayanan, M. (1996, May). Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA.

Satyanarayanan, M., J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*(4), 447–459.

Satyanarayanan, M., B. Noble, P. Kumar, and M. Price (1995). Application-aware adaptation for mobile computing. *Operating Systems Review 29*, 7.

Schlichting, R. and F. Scheider (1983). Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transaction on Computer Systems 1*(3), 222–238.

Severance, D. and G. Lohman (1976). Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems 3*(1), 256–267.

Solomon, D. and M. Russinovich (2000). *Inside Windows 2000.* Microsoft Press.

Srinivasan, P. (2001). An introduction to microsoft .net remoting framework - http://msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp.

Stonebraker, M. and E. Neuholf (1979). Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transaction on Software Engineering 3*(3), 188–194.

Szyperski, C. (1997). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

Tait, C. D. (1993, August). *A File System for Mobile Computing.* Ph. D. thesis, Graduate School of Arts and Sciences. Columbia University.

Tait, C. D. and D. Duchamp (1991). Service interface and replica management algorithm for mobile file system clients. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida*, pp. 190–197.

Tait, C. D. and D. Duchamp (1992). An efficient variable-consistency replicated file service. In *In proceedings of File System Workshop, USENIX, MI, USA.*, pp. 111–126.

Tait, C. D., H. Lei, S. Acharya, and H. Chang (1995). Intelligent file hoarding for mobile computers. In *Mobile Computing and Networking*, pp. 119–125.

Terry, D., A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch (1994, September). Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, Austin, TX, US, pp. 140–149.

Thomas, R. H. (1979, June). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transaction on Database Systems 4*(2), 180–209.

UPNP (2002). Universal plug and play forum - http://www.upnp.org/.

Watkins, D., M. Hammond, and B. Abrams (2003). *Programming in the .NET Environment.* Addison Wesley.

Wendt, C. P. (2002). http://www.codeproject.com/system/sysinfo.asp.

Whittington, J. (2002). Shared source cli provides source code for a freebsd implementation of .net-http://msdn.microsoft.com/msdnmag/issues/02/07/sharedsourcecli/default.asp.

Ximian (2002). Mono project, http://www.go-mono.com/.

Yu, H. and A. Vahdat (2000a). Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA*, pp. 305–318.

Yu, H. and A. Vahdat (2000b). Efficient numerical error bounding for replicated network services. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang (Eds.), *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt*, pp. 123–133.

Zaslavsky, A. and Z. Tari (1998, May). Mobile computing: Overview and current status. *The Australian Computer Journal 30*(2), 42–52.