Distributed systems (H04I4A) - summary (version 1)

Joris Schelfaut

January 29, 2014

Foreword

This is a summary for the course Gedistribueerde systemen[1] given by Prof. Dr. Joosen[2] and Dr. Walraven[3] at the Catholic University of Louvain[4] (KUL). The contents of this text are based on the book "Distributed Systems: Concepts and Design (5th Edition)"[5] by Coulouris et al. and the lecture slides.

Note that this is not a complete summary and may become outdated as the contents of the course change over the years. I can also not guarantee that there are no errors in this text, so be aware of that. Other than that, feel free to use this text while studying the course.

This text is also available at `http://districted.wordpress.com/concepts/distributed-systems/`. The text on that website may be a bit more up-to-date, depending on how much work I'm willing to put into it. If there are any mistakes or you would like to give some feedback, feel free to let me know on that website.

Joris.

[1] `http://onderwijsaanbod.kuleuven.be/syllabi/n/H04I4AN.htm`

[2] `https://distrinet.cs.kuleuven.be/people/wouter`

[3] `https://distrinet.cs.kuleuven.be/people/showMember.do?memberID=u0059786`

[4] `http://www.kuleuven.be/english`

[5] `http://www.cdk5.net/wp/`

Contents

Chapter 1

Introduction

A distributed system consists out of components that reside on different machines and communicate through message passing. This definition introduces three notable challenges:

1. The lack of a global clock is largely resolved by coordinating actions through messages, but has its limitations.
2. The nodes within this system may also fail independently and may even go undetected.
3. Concurrency of operations brings another challenge as resources may be accessed simultanously, possibly introducing inconsistencies.

Although these constraints make distributed systems complex to design, they come with significant benefits as well. The main motivation for developing and using distributed systems is resource sharing.

Middleware is the layer between architecture and application. It introduces an abstraction layer to build distributed applications, hiding the heterogeneity of the underlying architectural elements, e.g. protocols, servers, operating systems and so on. Middleware technology may be provide certain services such as distribution, security, ... [1]

1.1 Overview

The following topics are addressed in this text:

- **Direct communication** : This article discusses protocols and paradigms for direct communication in distributed systems such as remote procedure call and remote method invocation.
- **Indirect communication** : This article discusses protocols and paradigms for indirect communication in distributed systems such as group communication, message queues and publish-subscribe systems.
- **Distributed file systems** : This article gives an introduction to distributed file systems such as Sun NFS and the Andrew File System.
- **Distributed transactions** : This article gives an overview of some concepts related to distributed transactions such as distributed deadlock and commit protocols.
- **Replication** : This article explains the basics of replication and gives an overview of the Coda file system.
- **Cloud computing** : Introduction to cloud computing.

1.2 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

1.3 Links

- http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/ : Java RMI overview by Oracle;
- http://docs.oracle.com/javaee/5/tutorial/doc/docinfo.html : JEE tutorial by Oracle;
- https://developers.google.com/appengine/docs/java/ : Google App Engine Java overview.

Chapter 2

Direct communication

2.1 Introduction

2.1.1 Data representation

Marshalling is the assembling of a collection of data items into a form suitable for transmission, whereas *unmarshalling* is the disassembling of a message on arrival to produce the equivalent collection of data items. For example Java uses serialization and deserialization of objects, or SOAP in Service Oriented Architectures et cetera [2].

2.1.2 Message passing

Table 1 lists the two basic operations in message passing.

The semantics of send and receive operations can differ. In *synchronous* send the process waits for the corresponding receive. In synchronous receive, the process waits for the message arrival. In *asynchronous* send the process does not wait for the message arrival. In aynschronous receive, the process announces its willingness to accept or check for message arrival [2].

There a various possible message destinations, e.g., *processes*, which have a single entry point per process for all messages, or *ports*, which have one receiver and possibly many senders, or *mailboxes*, which may have many receivers [2].

Reliable communication

Failures in message passing may be the result of a number of causes, for example corrupted messages, duplicate messages, omission, i.e., the loss of messages, wrong ordering of messages, or receiver process failures [2]. In order to achieve reliable communication, messages should be delivered uncorrupted, in order, without duplicates, despite a reasonable number of packets dropped or lost. Unfortunately perfectly reliable communication can not often be guaranteed [2].

Reliable communication can be implemented using a number of techniques, for example [2]:

- **Corruption** : Include checksum in message;
- **Order mistakes and duplicates** : Include a message number which identifies the message;

Table 2.1: Basic message passing API.

Operation	**Description**
send (p: PortId; m: Message)	Send a message *Message* to a process at a port with given *PortId*.
receive (p: PortId; VAR m: Message)	Receive a message *Message* while listening at a port with given *PortId*.

Table 2.2: Basic request-reply protocol operations.

Operation	Description
doOperation	The client sends request and returns answer to the application program.
getRequest	The server gets the request from the client.
sendReply	The server sends a reply to the client.

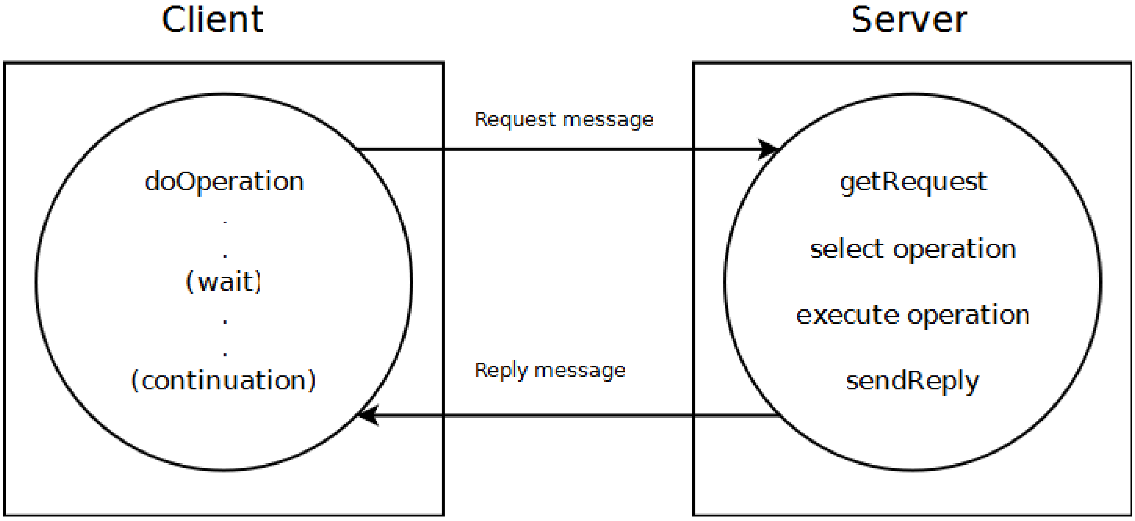

Figure 2.1: Model for request-reply communication.

- **Omission** : Sender stores message in buffer, sends it and sets a time-out and the receiver replies with acknowledgement. If no aknowledgement was received, the sender retransmits messages after timeout.

2.2 Remote invocation

2.2.1 Request-reply protocols

Request-reply protocols are designed to support roles and message exchanges in typical client-server interactions. **Table 2** lists the basic operations of request-reply protocols. **Figure 1** shows the general model for request-reply protocols.

Reliable communication

In reliability measures of TCP are an overkill as the acknowledgement from the receiver is redundant because the reply message is an acknowledgement. As a result, UDP can be used for building more efficient client server communication [2].

2.2.2 Remote procedure call

Traditional applications consist of a main program with a number of procedures (functions). In distributed systems procedures are grouped into servers, and main programs become clients. To achieve transparency the (remote) operations on a server from a client should look like conventional procedure calls [2].

To achieve this an additional message subsystem is introduced. When an application program calls client *stub procedure*, the client stub procedure marshalls parameters of call and gives it to

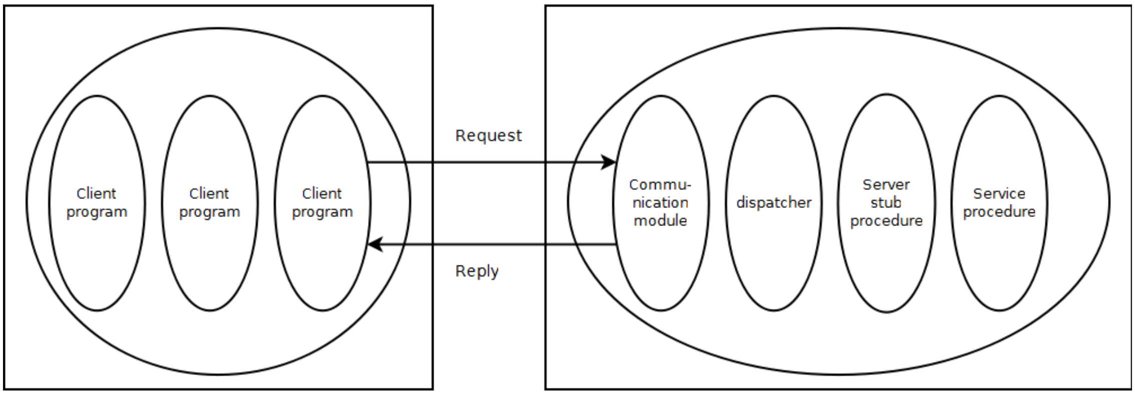

Figure 2.2: Remote procedure call.

communication module in client. The communication module then transmits a message with the marshalled RPC to the server's communication module who passes it on to the *dispatcher*. The dispatcher determines which procedure is called and calls the correct server stub procedure. Next, the server stub procedure unmarshalls data and calls the relevant server procedure. The server procedure returns the answer to the server stub procedure. The result is marshalled and sent to the client via the communication modules. The communication module at client side gives data to client stub procedure, who finally unmarshalls data and returns the answer to the calling program [2]. **Figure 2** shows the architectural elements in this scenario.

Remote procedure calls (RPC) can be integrated within a particular programming language, or based on a special *interface definition language* (IDL) [2].

Design issues

Heterogeous environment An Interface Definition Language (IDL) tries to provide abstraction for the heterogeneity of client and server implementations through programming language-independency. IDL describes operation signatures and its interface compilers are used as a base for generating client and server stubs, which can be implemented in different languages [2].

Transparancy To achieve transparency, RPC should be as much like local procedure calls as possible. However the calling instruction set is different, and there is no shared memory between caller and callee [2].

RPC requires some form of exception handling as failures cannot be hidden. Clients cannot distinguish between network failures or server failures. To deal with failures, language specific solutions may be applied, expressive return codes of functions, or extensions provided by IDL [2].

Semantics Remote procedure calls may have the following semantics [2]:

- **Maybe** : Requests are not resent, as a result no duplicate filtering is required, and it logically follows that procedure need no re-exececution, and no replies are re-transmited;
- **At-least-once** : Requests are re-sent, but as there is no duplicate filtering, procedures can be re-exececuted. To retain system consistency, operations have to be idempotent;
- **At-most-once** : Requests are re-sent, hence duplicate filtering is required and replies are re-transmitted. Procedures can not be re-exececuted;
- **Exactly-once** : Difficult or impossible given failures;

Implementation aspects

The task of the interface compiler is to generate client and server stub procedures, marshalling and unmarshalling operations for each argument type, and implement headers for server procedures.

Binding is the linking of the client to the server at execution time: the server will register a service at binder, and the client will perform a lookup for the service. Locating the binder is done through a well-known host address and is the responsibility of the operating system [2].

Asynchronous RPC

Asynchronous RPC can be used to reduce the idle time of processes that are waiting for a remote procedure call to complete [2].

Conclusions

RPC is a familiar paradigm which has been a basic primitive for distributed programming for many applications and systems [2].

RPC has some limitations with respect to failure handling, no transaction support, and the fact that RPC only supports one-to-one communication [2].

2.2.3 Remote method invocation

Remote method invocation (RMI) is a technology that similar to RPC allows clients to invoke methods on remote objects. It also uses programming with interfaces, called *remote interfaces*, can offer a number of call semantics and provides a similar level of transparancy as RPC, i.e., local and remote calls have the same syntax but the distributed nature of calls can be exposed, e.g., through remote exceptions. As a result, they share many of the design issues explained earlier, with the additional design issue of dealing with (distributed) objects for RMI [1].

RMI allows parameter passing by value, as input or output parameters, and also as object references. Remote invocation can then be used on these object references, instead of transmitting the complete object value accross the network [1]. The possible remote invocations are listed in the remote interface of that object.

Distributed objects

Classic object model Objects are accessed via their reference. An object is associated with an interface, which separates the method signatures from the actual implementation. Actions in object-oriented programs are initiated by invoking a method in another object [1]. Three possible effects are associated with method invocation [1]:

1. Change of the target object state, which consists of the values of its instance variables;

2. Creation of a new object;

3. Resulting in a new invocation on methods in other objects.

To deal with errors during execution, exceptions are used to create clearer error handling in complex code. Exceptions are a way to alter the control flow of a program [1].

A garbage collector detects when objects are no longer used and frees the memory occupied by these instances.

Distributed object model *Distributed objects* are objects that are managed by a server and implement a *remote interface* by which clients can invoke on the distributed object via its *remote object reference*. A remote object reference is an identifier that can be used throughout the distributed system to refer to a unique remote object [1].

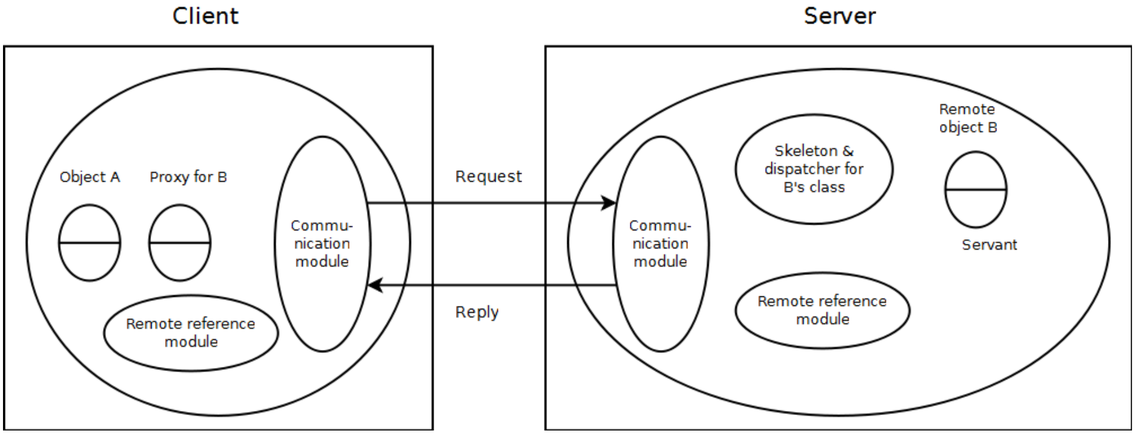

Figure 2.3: Remote method invocation.

Remote exceptions reveal some of distributed nature of RMI. Aside from errors in the program, exceptions in remote objects may be due to crashes in the server process, timeouts due to network failure et cetera [1].

In a distributed context, distributed garbage collection is slightly more complicated as object references may be spread accross different machines. It is achieved by extending the local garbage collection with a distributed module, often based on reference counting [1].

RMI implementation

Figure 3 shows the remote method invocation architectural components. In what follows we describe some of these components in more detail to finally form the complete picture of the RMI technology.

Communication module Communication modules operate using a request-reply protocol between client and server. They are responsible for enforcing specific invocation semantics, e.g., *at-most-once* [1].

When a request is received by the server's communication module, the communication module passes on the remote reference in the request to the remote reference module which returns the local reference. Next the server's communication module selects the dispatcher for the class of the object to be invoked (cf. RPC), passing on the local reference [1].

Remote reference module The task of the *remote reference module* is to translate between remote and local references by looking them up in a *remote object table.* The remote object table holds the following information [1]:

- Each remote object held by the process at the server's remote reference module;
- Each local proxy at the client's remote reference module.

RMI software layer The *RMI software layer* is a layer between the application-level objects and the communication and remote reference modules. The following middleware components are part of this layer [1]:

- **Proxy** : The proxy appears as a normal object in the client process, achieving some transparancy. However, instead of executing invocations, it forwards them in a message to the corresponding remote object after marshalling arguments. Results from the invocation are then unmarshalled and passed on to the client process;

- **Dispatcher** : For each class representing a remote object there is one skeleton and one dispatcher at the server. The dispatcher receives requests from the communication module;
- **Skeleton** : The class of a remote object has a skeleton, which implements the methods in the remote interface. It is responsible for marshalling and unmarshalling arguments and results respectively of requests before passing them on to the servant. A *servant* is an instance of a class that provides the body to a remote object. Servants live within a server process and handles the remote requests passed on by the corresponding skeleton;

2.3 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

[2] W. Joosen, 2013, "Distributed Systems Direct Communication PART I", iMinds-DistriNet, KULeuven

[3] W. Joosen, 2013, "Distributed Systems Direct Communication PART II", iMinds-DistriNet, KULeuven

Chapter 3

Indirect communication

3.1 Introduction

In [1] *indirect communication* is defined as "communication between entities in a distributed system through an intermediary with no direct coupling between sender and receiver". Uncoupling may be established in two ways:

- **Space** : The sender does not (need to) know the identity of the receiver.

- **Time** : The sender and receiver does not (need to) exist at the same time as the receiver.

Table 1 categorizes a number of technologies according to their support for time and/or space uncoupling. Note the relationship between time uncoupling and asynchronous communication. However, in the case of strict uncoupling in time, the receiving end does not necessarily exist at the time of sending, as mentioned earlier [1].

Typical applications of indirect communication is in mobile environments, cloud computing, and event dissemination where receivers are unknown or change rapidly [1].

3.2 Paradigms

3.2.1 Group communication

In group communication the messages within a distributed system are sent to a group, and from there sent to all other members of the group. The sender has no knowledge of the identities of the receivers, hence the indirection [1]. This kind of communication is called broadcasting where the sender forms a one-to-many relationship with the other members of the group.

Groups may be open or closed. In closed groups only members can multicast to it, opposed to open groups where processes outside the group can multicast to it as well.

Figure 1 gives an overview of the basic operations of group management. The operation set for group communication is shown in **table 2**.

Table 3.1: Overview of space and time coupling for distributed communication paradigms.

	Time-coupled	**Time-uncoupled**
Space-coupled	Message-passing, RMI	Message queues
Space-uncoupled	IP multicast	Publish-subscribe, tuple spaces

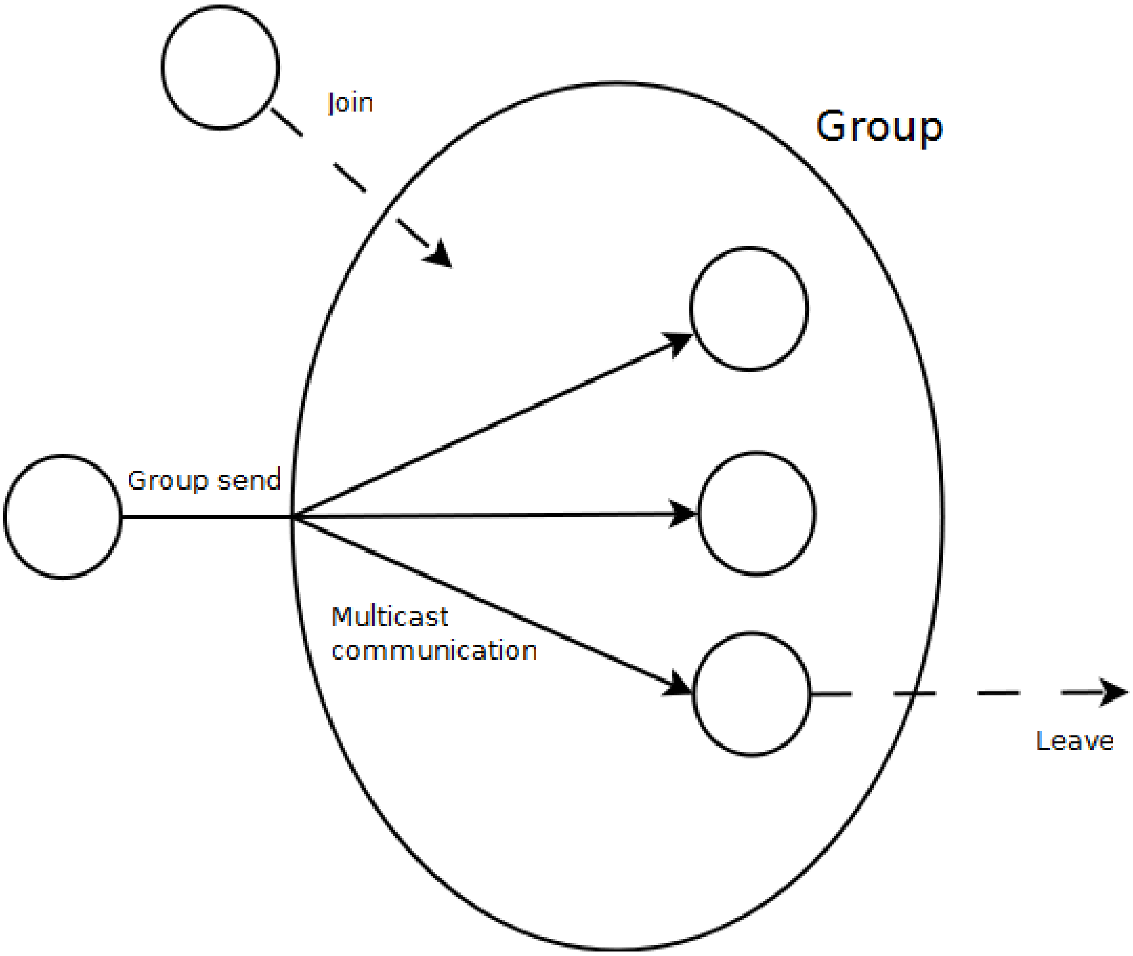

Figure 3.1: Group membership management in group communication.

Table 3.2: Group communication API.

Operation	**Description**
join (group)	*A process joins the group.*
leave (group)	*A process leaves the group.*
send (group, message)	*Send a message to the group. The group indirection layer propagates the message to all other members.*

Table 3.3: Publish-subscribe system API.

Operation	**Description**
publish (event)	*A publisher publishes an event.*
subscribe (filter)	*A subscriber subscribes to a set of events through a filter.*
unsubscribe (filter)	*A subscriber unsubscribes from a set of events.*
notify (event)	*Deliver events to its subscribers.*
advertise (filter)	*A publisher declare the nature of the events they will produce.*
unadvertise (filter)	*A publisher revokes the advertisement.*

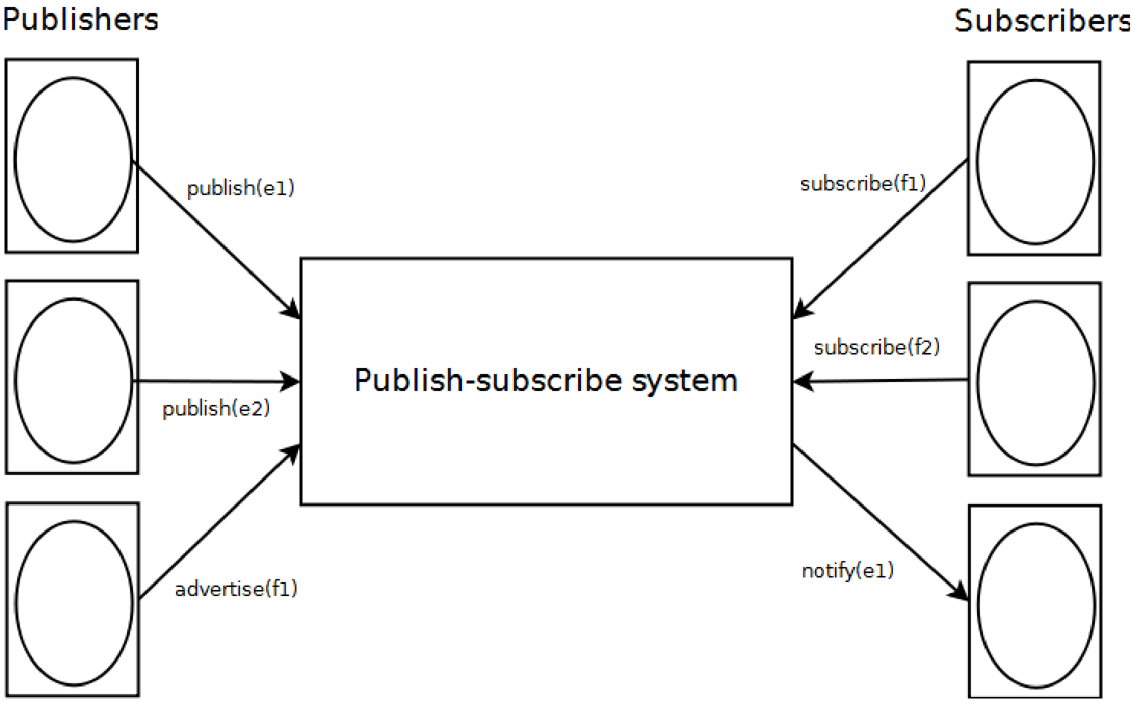

Figure 3.2: Publish-subscribe system architecture.

3.2.2 Publish-subscribe systems

A publish-subscribe system is a platform where *subscribers* can subscribe to certain events provides by *publishers*. The system then matches published events against subscriptions. Subscribers then receive an update if successful matches are found.

Publishers form a one-to-many relationship with their subscribers, but the publishers do not know who is subscribed. Subscribers also do not need to know the publisher, as long as they can specify which kind of messages they would like to receive . Publish-subscribe systems are uncoupled in time as they provide asynchronous communication between senders and receivers [1].

The operations of a publish-subscribe system are listed in **Table 3**.

3.2.3 Message queues

Message queues are a form of message-oriented middleware. A message queue introduces a layer of indirection between *producers* and *consumers*. A producers sends messages to a queue, next consumers receive messages from these queues. Message queues are uncoupled in time, but not in space. The relation between a consumer and a producer through a message is one-to-one [1].

Table 3.4: Message queue API.

Operation	**Description**
send (message)	*A producer sends a message to the queue.*
receive (message)	*A blocking receive operation. The consumer will block until an appropriate message is available.*
poll (message)	*The consumer checks the status of the queue. A message is returned if available, else a negative signal.*
notify (message)	*Start listening for event notications if a message is available.*

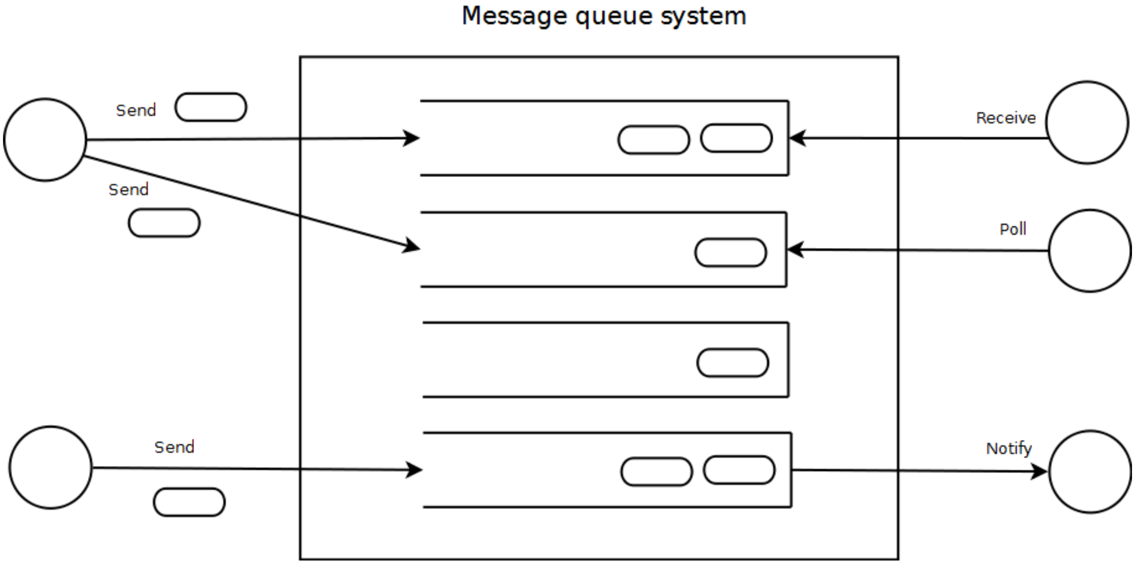

Figure 3.3: The message queue paradigm.

The operations that can be invoked on a message queue are listed in **table 4**.

Messages are usually added to the queue based on the first-in-first-out (FIFO) policy, but priorities may be used as well. Message queues try to ensure reliable delivery by persisting messages: messages are eventually delivered (time uncoupling). Messages are also only sent once and as received to provide integrity [1].

The consumers may receive messages by actively checking (polling) if messages are available, or by receiving notifications that messages have become available. Messages may be filtered based on certain properties [1].

3.2.4 Distributed shared memory

The objective of distributed shared memory (DSM) is to share data between computers. Each computer has a local copy of the data. This data is kept up to date by passing messages between each node over the DSM middleware. **Table 5** shows the operations for DSM.

3.2.5 Tuple spaces

A tuple space is a form of distributed memory where "processes communicate indirectly by placing tuples in a tuple space from which other processes can read and remove them" [1]. Space uncoupling is achieved as the sending and receiving processes may come from anywhere. Tuples may be taken from the space at any time and may even reside indefinately in the tuple space, thus achieving time uncoupling.

Table 3.5: Distributed shared memory API.

Operation	**Description**
read (data)	*Read from the shared memory.*
write (data)	*Write to the shared memory*
update (message)	*Send an update message to the other members of the distributed shared memory.*

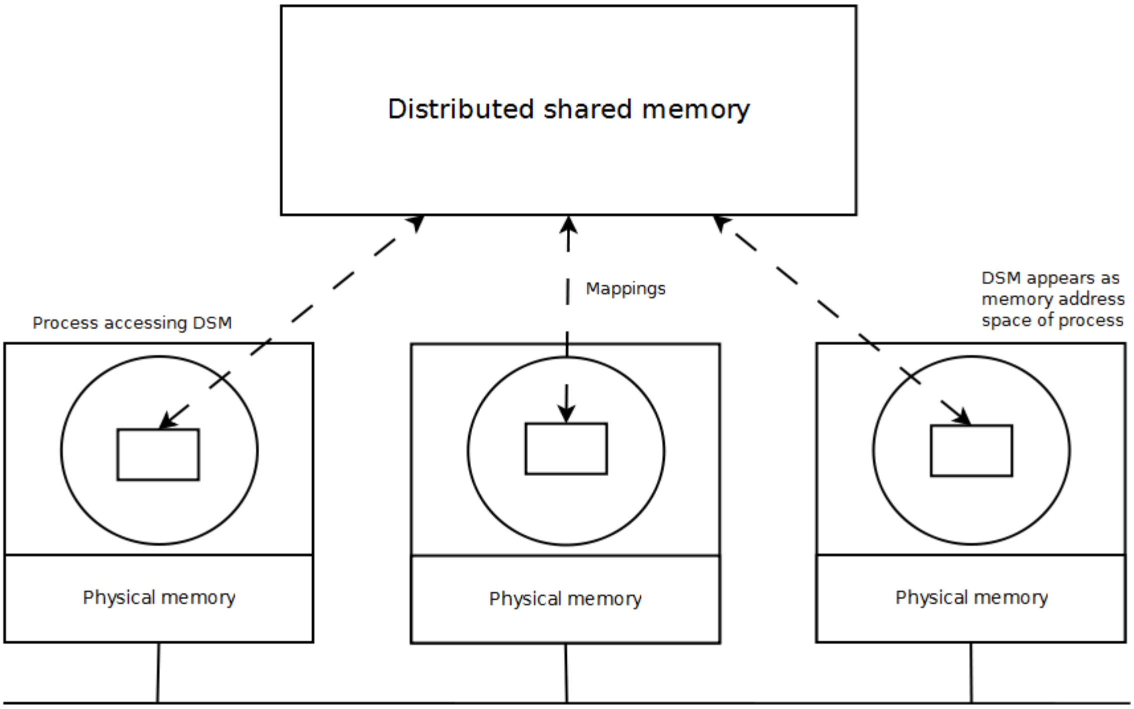

Figure 3.4: Distributed shared memory architecture.

Table 3.6: Tuple space API.

Operation	**Description**
read (tuple)	*Reads a tuple from the tuple space.*
take (tuple)	*Extract a tuple from the tuple space.*
write (tuple)	*Write a new tuple to the tuple space.*

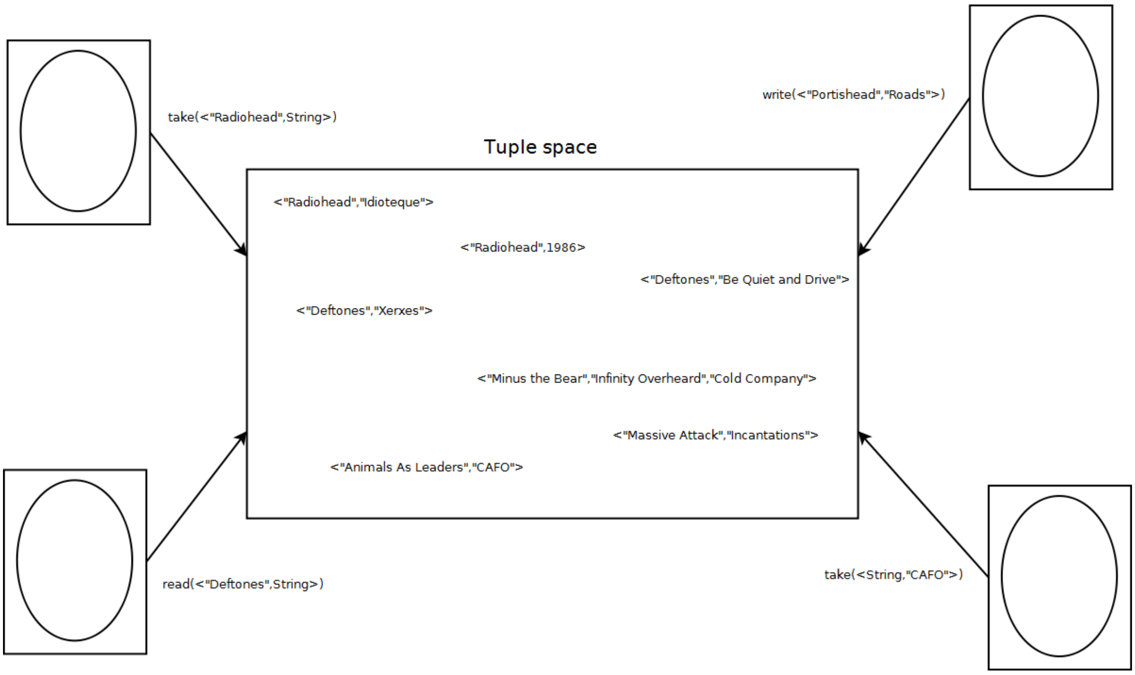
take(<"Radiohead",String>)
write(<"Portishead","Roads">)
Tuple space
<"Radiohead","Idioteque">
<"Radiohead",1986>
<"Deftones","Be Quiet and Drive">
<"Deftones","Xerxes">
<"Minus the Bear","Infinity Overheard","Cold Company">
<"Massive Attack","Incantations">
<"Animals As Leaders","CAFO">
read(<"Deftones",String>)
take(<String,"CAFO">)

Figure 3.5: Tuple space abstract example.

A tuple is typically of the form <var1, var2>, e.g. <*"hugo",1.19*>. A number of operations can be executed on a tuple space as listed in **table 6**.

3.3 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

Chapter 4

Distributed file systems

4.1 Definitions, architectural model and requirements

A distributed file system enables persistent data storage throughout an intranet. The objective of a distributed file service is resource sharing in an effort to reduce costs of storage and data management. Examples of these file systems are *Sun NFS* and the *Andrew File System (AFS)*, which will be discussed later.

Files form an elementary unit of a file system. A file consists of the actual data and a number of attributes, e.g. file length and timestamp. File naming is determined in *directories* in which text names are mapped to file identifiers. Files can be managed and manipulated through a series of operations such as create, read and write.

4.1.1 Design requirements

When designing a distributed file system, a number of transparency requirements should be taken into account. For example: *access transparency*, i.e., the set of operations for both local and remote files is the same, *location transparency*, i.e., the representation of the file system does not reveal the physical location of the files, or *mobility transparency*, i.e., when files are moved, no client code or administration tables should be altered.

Other requirements are for example maintaining file consistency, security (e.g. through Access Control Lists), concurrent file updates, replication and fault tolerance.

4.1.2 Architectural model design

The following architectural model is based on the implementation of both NFS and AFS. It considers three separate modules dividing responsibilities between them [1]:

- **Flat file service** : Provides an implementation for operations on the file contents. *Unique file identifiers (UFID)* are used to refer to files in all flat file service operations.

- **Directory service** : Provides a mapping between text names for files and their UFIDs and operations on directories.

- **Client module** : Provides a single programming interface for the flat file service and directory service and runs on a each client machine.

Figure 1 gives a schematic overview of this model.

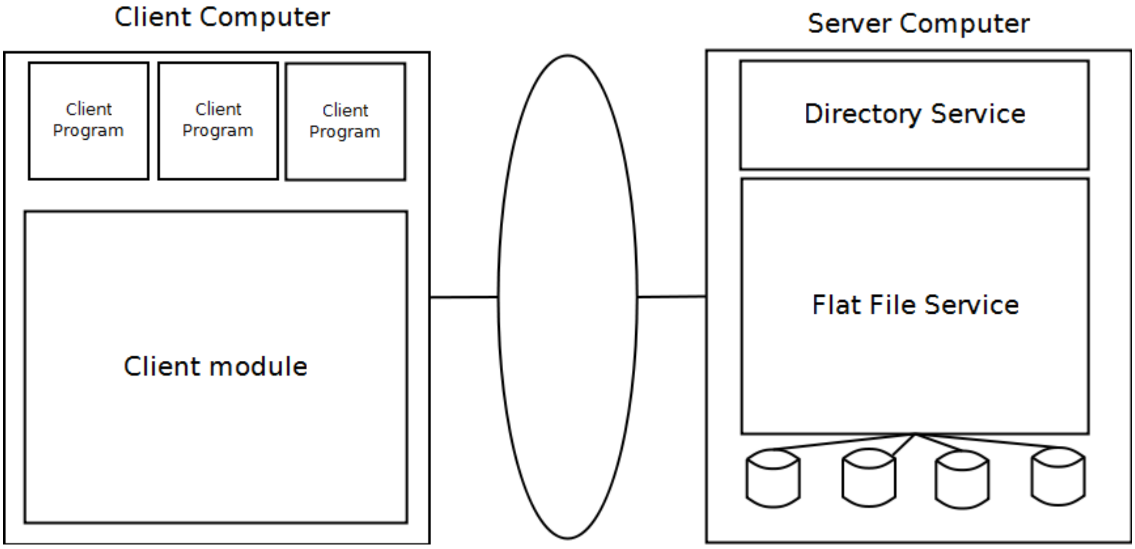

Figure 4.1: File service architecture.

Table 4.1: Flat file system API, adapted from [1].

Operation	**Description**
Read(FileId, i, n) → Data - throws BadPosition	*If* $1 \le i \le Length(File)$ *: Reads a sequence of up to n items from a file starting at item i and returns it in Data.*
Write(FileId, i, Data) - throws BadPosition	*If* $1 \le i \le Length(File) + 1$ *: Writes a sequence of Data to a file, starting at item i, extending the file if necessary.*
Create() → FileId	*Creates a new file of length 0 and delivers a UFID for it.*
Delete(FileId)	*Removes the file from the file store.*
GetAttributes(FileId) → Attr	*Returns the file attributes for the file.*
SetAttributes(FileId, Attr	*Sets the file attributes.*

Table 4.2: Directory service API, adapted from [1].

Operation	**Description**
Lookup(Dir, Name) → FileId - throws NotFound	Locates the text name in the directory and returns the relevant UFID. If Name is not in the directory, throws an exception.
AddName(Dir, Name, FileId) - throws NameDuplicate	If Name is not in the directory, adds (Name, File) to the directory and updates the file's attribute record. If Name is already in the directory, throws an exception.
UnName(Dir, Name) - throws NotFound	If Name is in the directory, removes the entry containing Name from the directory. If Name is not in the directory, throws an exception.
GetNames(Dir, Pattern) → NameSeq	Returns all the text names in the directory that match the regular expression Pattern.

4.1.3 Implementation techniques

File groups

A file group is a collection of files mounted on a given server. A server can contain multiple file groups. File groups can be transferred between servers and forms a unit of distribution over servers allowing transparent migration of file groups.

Files are locked in a file group on creation and are assigned a UFID including a file group identifier component [1,2]. File group identifiers are unique throughout the distributed system.

Examples of file groups are *filesystems* (as opposed to *file systems*) on NFS and *volumes* on AFS.

Space leak prevention

The creation of a file is a two-step process [2]:

1. Creation of an (empty) file with a new UFID;

2. Naming of the file and adding the UFID to the corresponding directory;

A failure after the first step causes the file to exist in the file server, but makes it unreachable, as the UFID is not in any directory. This lost space on disk is called a space leak. Detection of space leaks requires co-operation between the file server and the directory server [2].

Access control

Access control is required for security reasons. Access control models usually follow the model outlined by Lampson. Different abstraction models can be used to implement access control, e.g. proection domains, capabilities, access control lists, and so on [1].

Capabilities are binary values that act as digital keys. Ownership of a capability grants access to certain resources [1]. Capabilities are prone to two issues [1]:

1. **Key theft** : Stolen keys can be abused regardless who its current owner is;

2. **Revocation** : Users that are no longer authorized may still keep the key and use it maliciously;

Replication

A file may be represented by a list of copies at different locations. This way servers can share the load, improving scalability and fault tolerance [1].

Caching

Server caching is used to reduce delay for disk I/O. Client caching reduces network delay [2].

4.2 Example systems

4.2.1 Sun NFS

Figure 2 gives a schematic overview of the Sun NFS architecture. Note the similarity with the model depicted in **figure 1**.

In Sun NFS client and server modules can be in any node. Sun NFS attempts to emulate a standard file system by integrating file and directory services and integrating remote file systems in a local one through mounting [2]. This way Sun NFS tries to achieve access transparency.

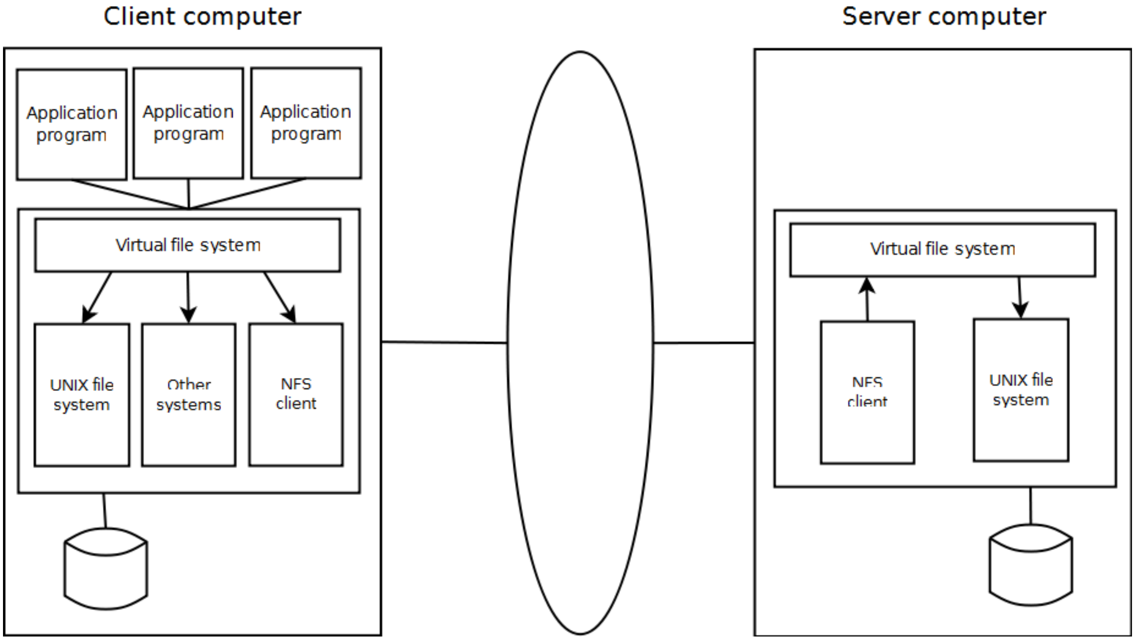

Figure 4.2: Sun NFS architecture.

Virtual file system

Sun NFS tries to achieve access transparency through a *virtual file system (VFS)* that provides an abstraction layer on top of local and remote files. **Figure 2** illustrates the VFS layer providing access transparency for client applications [1].

The VFS is part of the UNIX kernel to manage local file identifiers and remote file identifiers, called *file handles*. A file handle is a combination of the filesystem identifier, the i-node number and the i-node generation number. The i-node number of a UNIX file is an identification within the system that the file is stored. The i-node generation number reflects the number of times the i-node number has been reused [1].

Integration through interfaces

The NFS client module is integrated in kernel and offers a standard UNIX interface. This has several advantages [2]:

- No client recompilation/reloading;
- Single client module for all user level processes;
- Encryption at kernel level.

Server integration is mainly implemented for performance reasons.

In the model depicted in **figure 2** is connected to the VFS in both client and server. Communication between client and server occurs through the interface and NFS protocol.

Mounting service

The mount service is a process that runs on the NFS server. The file */etc/exports* contains the names of the local filesystems that are avilable for remote mounting. Access lists determine which hosts are permitted to mount which filesystems. Users can mount any subtree of the filesystems they have access to, based on a chosen directory [1].

Remote filesystems may either be hard-mounted or soft-mounted in a client computer [1,2]:

- **Hard-mounting** : A client waits until a request for a remote file succeeds;
- **Soft-mounting** : Mounting failure is returned if the request does not succeed after *n* retries;

At the client, multi-part file pathnames are translated to i-node references. Each i-node referencing a remote mounted directory is translated into a file handle using a separate *lookup()* request to the remote server. The VFS resolves file handles to local or remote directories. Mounting performance is improved through caching [1].

An addition to this system is the automounter. The automounter dynamically mounts remote directories when an empty mount point is referenced by a client. The automounter behaves as a local NFS server for the client machine. It holds a mapping of pathnames, or *mount points* against corresponding servers. As the client accesses mount points when resolving a path name, the client module invokes a *lookup()* request on the local automounter. The automounter triggers a number of probe requests to the corresponding NFS servers. Finally the referenced file systems are mounted onto the mount points via a symbolic link to avoid redundant requests to automounter [2].

Caching

Caching is done both at client and server side. Caching improves the performance of NFS significantly.

Server caching The server caching system is based on standard UNIX caching. File pages, directories, and file attributes read from disk are maintained in a memory buffer cache until the buffer space is required for other pages. Requests for files that are already in the cache then don't require expensive disk access. *Read ahead* anticipates future read requests by fetching nearby file pages from memory [1].

When writes are performed in this system, additional measures are needed. The system supports two modes of writing [1]:

- **Write-through** : The server writes updated file pages in the server's cache to disk before sending a reply to the client. When the reply is received, the client knows the data has been written to disk;
- **Delayed write** : Data stored in the cache is only written to memory when a commit operation is received for the relevant file. The client knows the file is written to disk as soon as a reply is received to a commit operation request;

Client caching Caching is used to reduce the number of requests to the server. The results of the following operations are cached: *read()*, *write()*, *getattr()*, *lookup()*, and *readdir()*. As a result of the delayed updates introduced by caching, different versions of the same file pages may exist simultanously at different client nodes. To solve this problem, clients use polling to check the currency of their cached data, using a timestamp based method [1].

Let *Tc* be the time when the cache entry was last validated, and *Tm* the time when the block was last modified at the server. A cache entry is valid at time *T* if *T* - *Tc* is less than a freshness interval *t*, or if the value for *Tm* recorded at the client is equal to this value recorded at the server. The value of the freshness interval *t* is a tradeoff between consistency and efficiency - on a Sun Solaris client, *t* lies somewhere between 3 and 30 seconds. Formally this yields to following formula:
$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$

The cause for recent updates not to be visible immediately at the client has two sources of delay:

- The delay after the write before the updated data leaves the cache in the updating client's kernel;

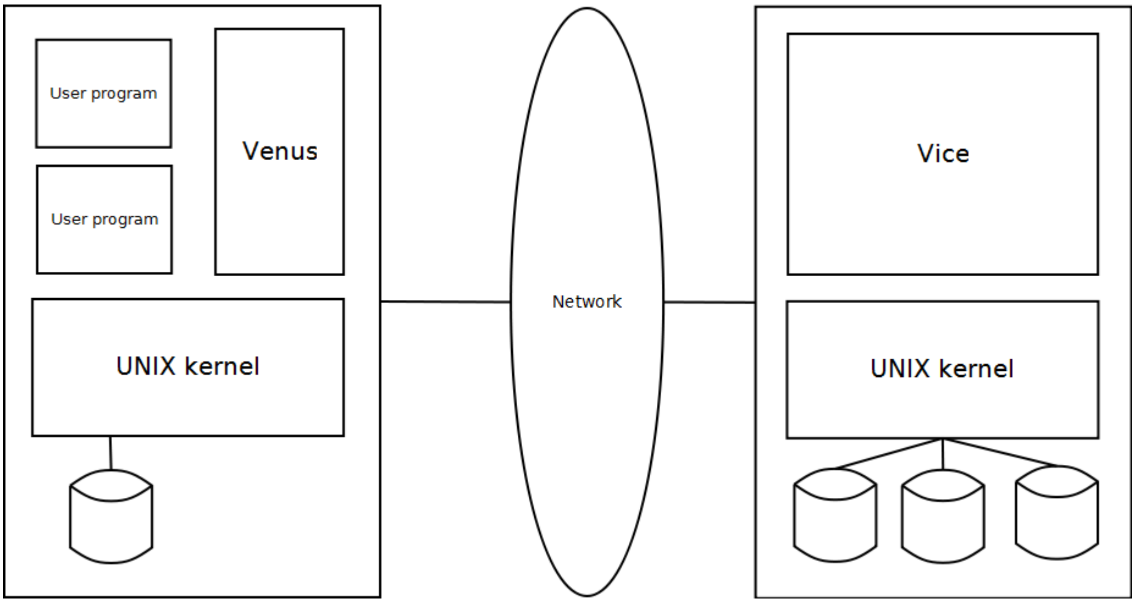

Figure 4.3: Andrew File System architecture.

- The window for cache validation.

When a cached page is modified, it is marked as dirty and scheduled to be flushed to the server asynchronously. Pages are flushed when a file is closed or a *sync* operation occurs at the client. A *bio-daemon* is used to facilitate read-ahead and delayed-write operations. A bio-daemon is notified after each read request, and it requests the transfer of the following file block from the server to the client cache. In the case of writing, the bio-daemon will send a block to the server whenever a block has been filled by a client operation [1].

Access control

The NFS server is stateless. As a result, the user's identity has to be verified against the file's access permission attributes with each request. Encryption and integration with Kerberos tries to prevent impersonation [1].

4.2.2 Andrew File System

A schematic overview of the Andrew File System is given in **figure 3**.

The Andrew File System is characterized by two design decisions [1]:

1. **Whole-file serving** : The entire file contents are transmitted to client computers by AFS servers;
2. **Whole-file caching** : File copies at the client are stored in a cache on the local disk, i.e., the cache is permanent.

The design strategy is based on a number of assumptions about the average and maximum file size and locality of reference to files in UNIX systems [1]:

- Files are small;
- Read operations are much more common than write operations;
- Sequential access is common, random writes are rare;

- Most files are read and written only by one user;
- Files are referenced in bursts, i.e., files referenced recently are likely to be referenced again.

A scenario that illustrates the operation of AFS is as follows [1]:

1. The user process in a client computer issues an *open* call for a file the shared file space. There is no current copy of the file in the cache. The client sends a request to the server containing the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is opnened and the resulting UNIX file descriptor is returned to the client.
3. Subsequent *read*, *write*, etc. operations on the file by processes in the client computer are applied on the local copy.
4. The client process issues a *close* system call. If the local copy has been updated, its contents are sent back to the server. The server performs relevant updates. The client keeps its copy of the file in the cache.

The design affects performance and the semantics of the system. Based on the previously described design characteristics, we can make the following predictions about AFS performance:

- Locally cached copies remain valid for a long time if the files are updated infrequently and/or files that are updated by just a single user;
- The provision of sufficient cache space on a client machine ensures that files in regular use are normally retained in the cache until they are needed again;
- Databases do not scale well with AFS as they are updated frequently and shared by many users.

Implementation

The *Vice* is server software that runs as a user-level UNIX process in each server computer. The *Venus* is a user-level process that runs in the client computer. Referring to the abstract model in **figure 1**, the Venus process corresponds to the client module [1].

File in the workstations are either local or shared. Shared files are stored on servers and copies are cached in client computers. A specific subtree *cmu* contains all the shared files. User directories are in the shared space, enabling file access from any workstation. One of the partitions on the local disk of each workstation is used as a cache. It is managed by the Venus component of the client [1].

Files are grouped into volumes. *Fids* include the volume number of the volume containing the file, an NFS file handle identifying the file within the volume, and a *uniquifier* to avoid fid reuse. The Vice servers only accept requests by the Venus in terms of fids. On the client computer pathnames are used to access files, which are then translated by the Venus component into fids [1].

Caching

When Vice supplies a copy of a file to a Venus process it also provides a callback promise. A *callback promise* is a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file [1].

Whenever the client's Venus handles an *open* operation, it checks the cache. If the required file is found in the cache, then its token is checked. The token can have two states [1]:

- **Valid** : The cached copy can be opened and used without reference to Vice;
- **Cancelled** : A fresh copy of the file must be fetched from the Vice server. When the Venus process receives a callback, it sets the callback promise token for the relevant file to cancelled.

Maintaining consistency When a workstation is restarted after a failure or a shutdown, its Venus component cannot assume that the callback promise tokens are correct, as some callbacks may have been missed. So before a file is accessed, the Venus has to send a validation request containing the file modification timestamp to the server that is the custodian of the file. If the timestamp is current, the server responds with valid and the token is reinstated. If the timestamp shows that the file is out of date, then the server responds with cancelled and the token is set to cancelled [1].

To deal with possible communication failures, e.g., loss of callback messages, callbacks must be renewed before an *open* operation if a certain amount of time has passed since the file was cached without communication from the server.

Since the majority of files are not accessed concurrently, and read operations predominate over writes in most applications, the callback mechanism results in a dramatic reduction in the number of client-server interactions [1].

The callback mechanism used in AFS requires Vice servers to maintain some state on behalf of their Venus clients, unlike NFS. To retain callback lists must be retained over server failures, they are held on the server disks and are updated using atomic operations. This design decision introduces some overhead: dealing with failures, maintaining state.

Update semantics One-copy file semantics are not practicable in large-scale systems. A strict implementation of one-copy semantics would require that the results of each write to a file are distributed to all cached copies before any further accesses can occur. The goal of the cache-consistency mechanism is to achieve an approximation of these semantics [1].

A client may open an old copy of a file after it has been updated by another client. This occurs if a callback message is lost, for example as a result of a network failure. But there is a maximum time, *T*, for which a client can remain unaware of a newer version of a file. For a client C, a file F and corresponding custodian server S we have the following guarantee after a successful *open*:
$latest(F, S, 0) \vee (lostCallback(S, T) \wedge inCache(F) \wedge latest(F, S, T))$

Here *latest*(F, S, T) denotes that the copy of F seen by the client is no more than T seconds out of date, *lostCallback*(S, T) denotes that a callback message from S to C has been lost at some time during the last T seconds, and *inCache*(F) indicates that the file F was in the cache at C before the open operation was attempted [1].

4.3 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

[2] W. Joosen, 2013, "Distributed File Systems", iMinds-DistriNet KULeuven

Chapter 5

Distributed transactions

5.1 Introduction

Transactions are a way to describe sequences of operations by a client on a system. When these operations are invoked on different servers, the transaction becomes distributed [1]. The goal of distributed transactions is to achieve consistency of data in a distributed environment. To achieve this usually one of the servers acts as *coordinator* and manages the *participants* in the transaction. The coordinator keeps track of other servers, called *workers*, and is responsible for final decision: when a transaction is to be finalized, agreement is needed between all servers involved to either commit or abort [2].

Transactions are said to have ACID properties [2]:

- **Atomicity** : A transaction either completely succeeds or fails. In the first case the transaction will be committed and its results persisted; in the second case the transaction will have been aborted and won't have any effects.
- **Consistency** : A transaction moves data from one consistent state to another.
- **Isolation** : There is no interference from other transactions and intermediate effects are not visible to other transactions.
- **Durability** : Once a transaction commits, the effects of the transaction are preserved despite subsequent failures.

There are two types of transactions, based on their structure: *flat transactions* and *nested transactions*. In flat transactions all work is done at the same level between the start of the transaction and the commit or abort message. It is also not possible to commit or abort parts of a flat transaction.

Table 1 gives an overview of the operations associated with flat transactions. **Figure 1** shows a general model for flat distributed transactions.

Nested transactions have finer grained recovery from failures as sub-transactions fail independent. Sub-transactions commit or abort independently, without effect on the outcome of other sub-transactions or enclosing transactions. The effects of sub-transactions becomes durable only when top-level transaction commits [2].

Table 2 gives an overview of the operations associated with nested transactions. **Figure 2** depicts a general model for nested distributed transactions.

5.2 Two-phase commit protocols

The goal of an atomic commit protocol is to ensure that the requirements for transactions are met [1]. The protocol must work correctly, even when some servers fail messages are lost servers are temporarily unable to communicate [2].

Table 5.1: Operations in coordinator for flat transactions.

Operation	**Description**
openTransaction () → trans;	*Starts a new transaction and delivers a unique transaction identifier (TID) trans. This identifier will be used in other operations in the transaction.*
closeTransaction (trans) → commit, abort;	*Ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.*
abortTransaction (trans);	*Aborts the transaction.*
join (trans, participant);	*Informs a coordinator that a new participant has joined the transaction trans.*

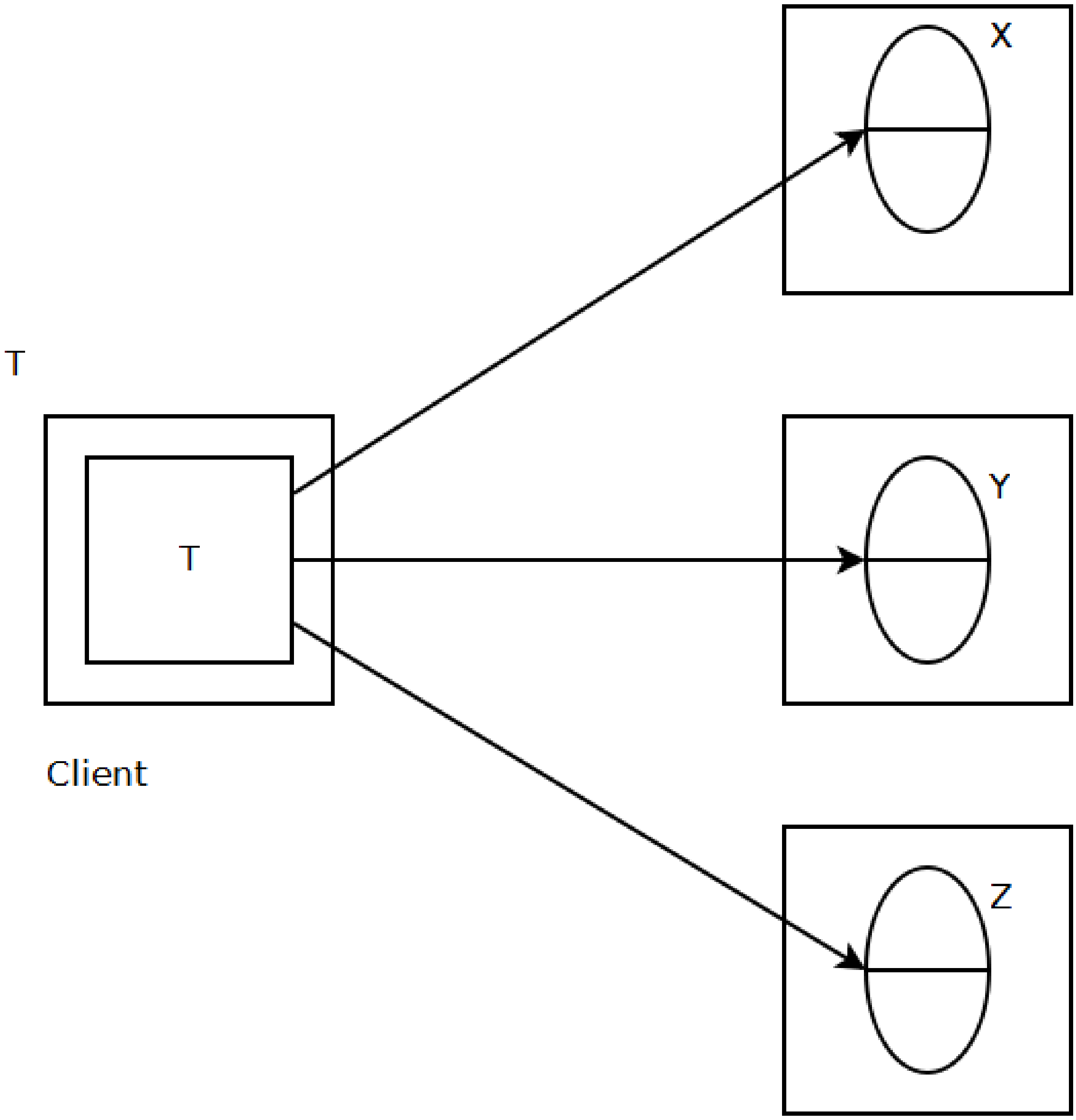

Figure 5.1: Flat transaction.

Table 5.2: Operations in coordinator for nested transactions.

Operation	**Description**
openSubTransaction (trans) → subTrans;	*Opens a new subtransaction whose parent in trans and returns a unique subtransaction identifier.*
getStatus (trans) → committed, aborted, provisional;	*Asks the coordinator to report on the status of the transaction trans. Returns values representing one of the following: committed, aborted or provisional.*

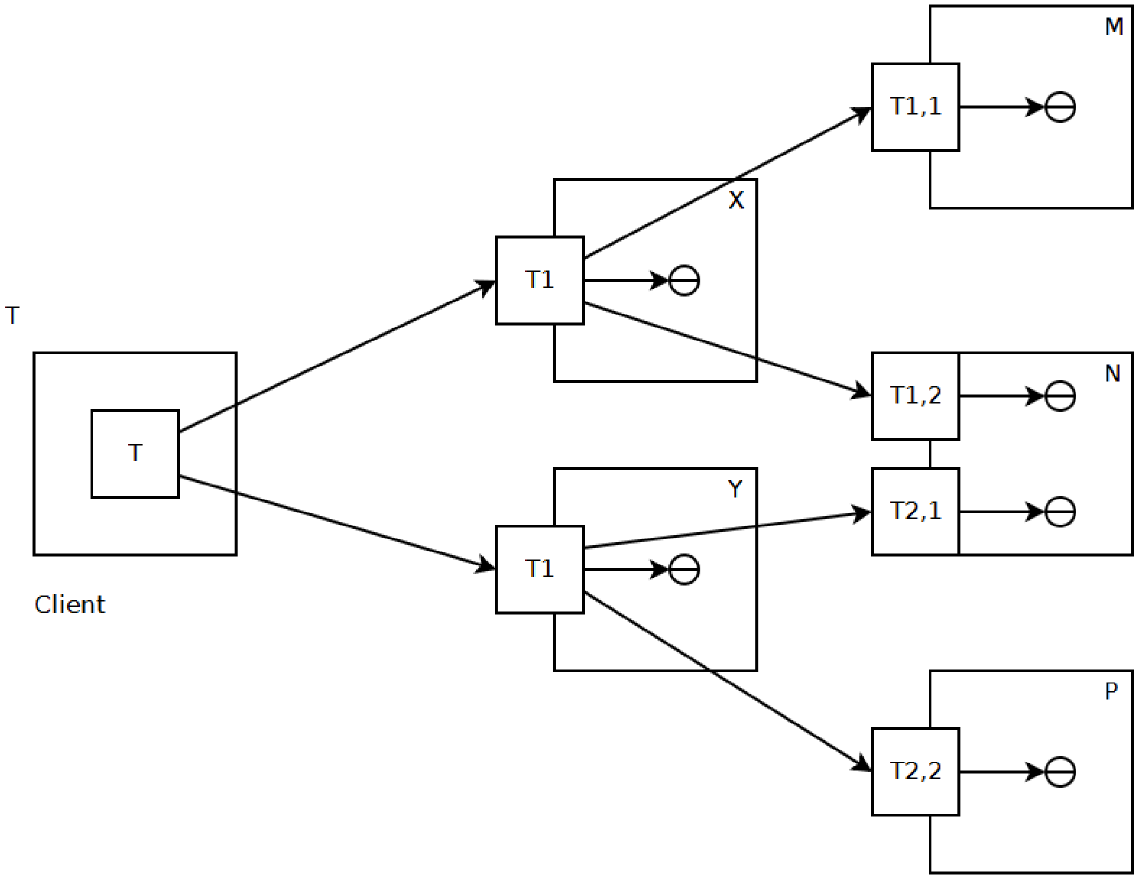

Figure 5.2: Nested transaction.

Table 5.3: Operations for the two-phase commit protocol.

Operation	**Description**
canCommit (trans) → Yes / No;	*Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.*
doCommit (trans);	*Call from the coordinator to participant to tell participant to commit its part of a transaction.*
doAbort (trans);	*Call from coordinator to participant to tell participant to abort its part of the transaction.*
haveCommitted (trans, participant);	*Call from participant to coordinator to confirm that it has committed the transaction.*
getDecision (trans) → Yes / No;	*Call from participant to coordinator to ask for the decision on a transaction when it has voted Yes but still had no reply after some delay. Used to recover from server crash or delayed messages.*

The two-phase commit protocol consists out of a votong phase and a completion phase. The completion depends on the outcome of the voting. In the voting phase participants vote for the transaction to be committed or aborted. Once voted for commit, a participant must ensure it can actually carry out the commit. In that case the participant will remain in a *prepared* state. As soon as the coordinator gives a signal that the votes where all in favour of a commit, the result transaction can be committed. In the other case, the transaction is aborted.

The protocol as given in [1] is as follows and is based on the operations listed in **table 1**; **figure 3** gives an example scenario of how the protocol is used:

- **Phase 1 (voting phase) :**
  1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
  2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No*, the participant aborts immediately.
- **Phase 2 (completion according to outcome of votes) :**
  3. The coordinator collects the votes (including its own).
     a. If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
     b. Otherwise, the coordinator decides to abort the transaction and sends doAbort requests to all participants that voted *Yes*.
  4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

For nested transactions, when a subtransaction completes, it either to commits provisionally or aborts. The decision is made indepently. A provisional commit differs from the prepared state in that nothing is backed up in permanent storage. Once all subtransactions have completed, the ones that have been committed provisionally will try to commit their results using the two-phase commit protocol with an additional constraint: if a parent transaction has aborted, the subtransactions will abort as well. A parent transaction however, can commit even if one of its subtransactions has aborted, depending on the implementation [1]. An example of how these transactions are structured is shown in **figure 4**.

There are two datastructures that are held at the coordinator: a commit list, i.e., a list of all committed (sub)transactions, and an abort list, i.e., a list of all aborted (sub)transactions [2].

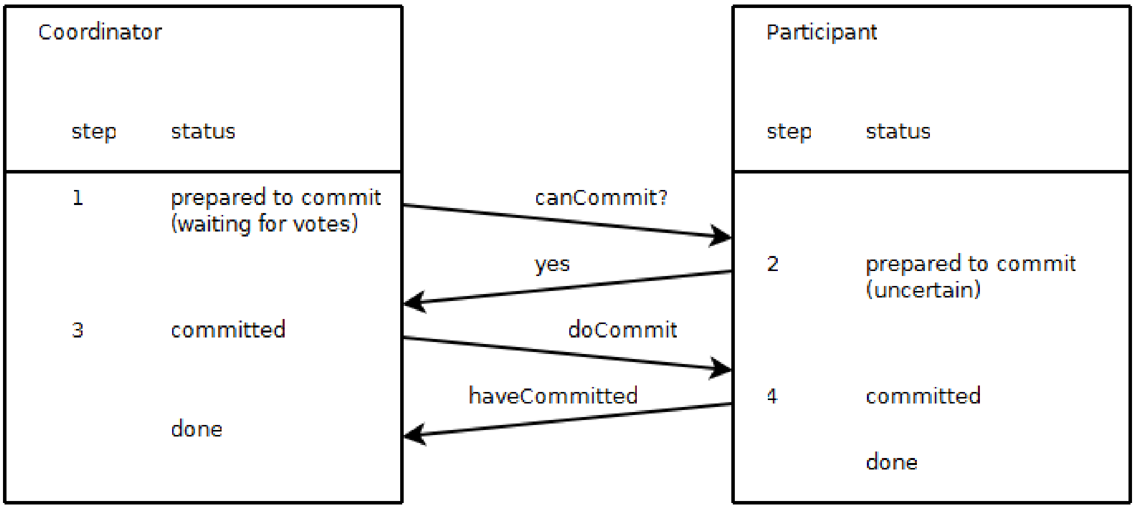

Figure 5.3: Communication in the two-phase commit protocol.

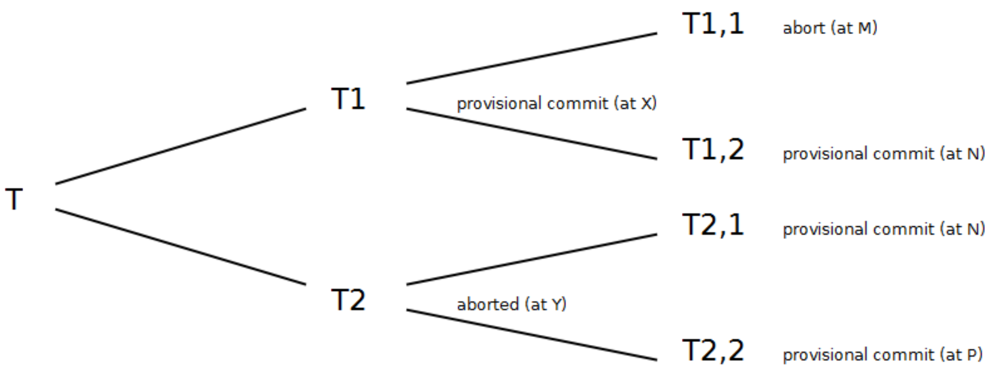
T
T1
T2
provisional commit (at X)
aborted (at Y)
T1,1
abort (at M)
T1,2
provisional commit (at N)
T2,1
provisional commit (at N)
T2,2
provisional commit (at P)

Figure 5.4: Example scenario for two-phase commit in nested transactions (based on the system depicted in figure 2).

The complete datastructure then consists of a list of all transactions, each with a list of their corresponding direct child transactions and commit and abort list [2].

5.3 Concurrency

Servers manage their objects and are responsible for their consistency. Concurrency control is an important aspect of transactions: transactions that access objects in a conflicting way must be handled in the same order by all servers [1].

There are a number of protocols to achieve concurrency control:

- **Locking** : *Locks* are used to facilitate concurrency control. They are held locally at each server. Locks are only released when a transaction has either been committed or aborted at all servers [3].
- **Optimistic concurrency control**
- **Timestamp ordering** : The ordering of transactions can be done through timestamps.

5.3.1 Locking

In nested transactions child transactions inherit locks from their parent. When a nested transaction commits, its locks are inherited by its parents, when it aborts, its locks are removed [3].

A transaction is not allowed any new locks after it has released a lock. This results in serial equivalence and requires all of a transaction's accesses to a particular data item to be serialized with respect to accesses by other transactions, and all pairs of conflicting operations of two transactions to be executed in the same order [2]. This results in the *two-phase locking* protocol, consisting of the following two steps:

1. **Growing phase** : New locks can be acquired;
2. **Shrinking phase** : No new locks and release of locks.

By releasing locks only at commit or abort, intermediate results can be hidden [2].

5.3.2 Optimistic concurrency control

5.3.3 Timestamp ordering

In the case of distributed transactions, the coordinators must issue globally unique timestamps, which they subsequently correspond to each other. To achieve the same ordering at all severs, the coordinators agree to the ordering of their timestamps [1].

5.4 Distributed deadlocks

To detect deadlocks a wait-for graph can be created. If there exists a cycle in the wait-for graph, a deadlock has occurred. A wait-for graph a directed graph G(V, E), where the vertices V represent transactions and objects, and the edges E represent either an object held by a transaction or a transaction waiting for an object [1]. An example is shown in **figure**. A *distributed deadlock* occurs when there is a cycle in the global *wait-for graph*, as opposed to the local wait-for graph at the server/clients themselves.

A particular problem that occurs in distributed deadlock detection are *phantom deadlocks*. This is a scenario where a deadlock is detected in an outdated wait-for graph. As it may take some time to construct the full graph, a waiting transaction may have already been aborted causing a resource to be no longer required by the corresponding process. Hence, other transactions may be aborted unnecessarily to resolve the phantom deadlock [1]. If transactions are using two-phase locks, they cannot release objects and then obtain more objects, and phantom deadlock cycles cannot occur in the way suggested here [1].

There exist centralized and decentralized approaches to construct a global wait-for graph. In the following paragraphs we will discuss an example of both.

5.4.1 Centralized deadlock detection

In centralized deadlock detection one server has the responsibility for detecting deadlocks. Each time after a certain time interval, each server sends the lastest copy of its local wait-for graph to this global deadlock detector. When a deadlock is detected, it makes a decision on how to resolve it and notifies the servers which transactions to abort [1].

This kind of deadlock detection has some obvious disadvantages [1]:

- Poor availability;
- Lack of fault tolerance;
- Poor scalability.

5.4.2 Distributed deadlock detection: the edge-chasing algorithm

A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*. Here, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph [1].

Edge-chasing algorithms have three steps [1]:

1. **Initiation** : When a server notes that a transaction T starts waiting for another transaction U, where U is waiting to access an object at another server, it initiates detection by sending a probe containing the edge $<$ T → U $>$ to the server of the object at which transaction U is blocked. If U is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

2. **Detection** : Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes. The global wait-for graph is built one edge at the time. As soon as cycle is detected, a deadlock has occurred.

3. **Resolution** : When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

5.5 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

[2] W. Joosen, 2013, "Distributed Systems - Transactions - I", IBBT-DistriNet, KULeuven

[3] W. Joosen, 2013, "Distributed Systems : Transactions - Part 2", IBBT-DistriNet, KULeuven

Chapter 6

Replication

6.1 Definitions and models

Replication is "the maintenance of copies of data at multiple computers" [1]. In the context of replication data can be thought of as *objects*, where each *logical* object is in fact a collection of physical copies called *replicas*. *Replication transparency* may be included as another requirement for distributed system design. Replication helps making distributed systems more effective in three ways [1]:

1. **Performance enhancement** : caching at server and client side helps resolving latency problems. Replication of immutable data is trivial, whereas data that can change over time must be kept up to date. In the latter case, the generated overhead may put a limit to the performance increase.

2. **Increased availability** : Server failures and communication disconnections may decrease availability of resources. By keeping local copies of data, the availability of this data naturally increases.

3. **Fault tolerance** : Maintaining correctness of replicated data is imperative for the effectiveness of the replication model.

The model sketched in **figure 1** consists out of a number of components called *replica managers* and a number of clients with a component called a *front end* associated with it. Replica managers apply operations directly on the replicas, often using atomic operations. In this case the state of the replicas is a deterministic function of the initial state the collection of applied operations [1].

Replica managers provide a service, i.e. access to objects, to the clients. Clients indirectly access object through the methods listed in **table 1**. The front end handles the client requests and uses messages to communicate with the replica managers. This abstraction layer ensures replication transparency at the client side [1].

Coulouris et al. [1] list five phases in which the request is handles by the system:

1. **Request** : The front end issues a request to one or more replica managers, either through unicast or through multicast. In the first case the replica manager will propagate the message to other replica managers.

Table 6.1: Replica manager service API.

Operation	Description
readOnlyRequest (object)	*Template method for an invocation by the client on an object with no updates on the object itself.*
updateRequest (object)	*Template method for an invocation by the client on an object, which alteres the state of the object.*

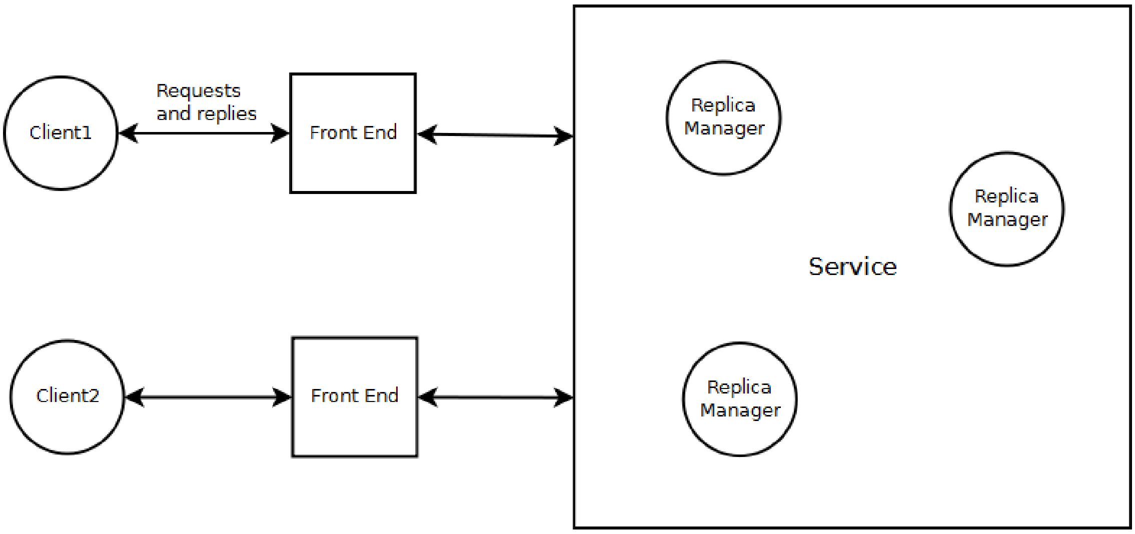

Figure 6.1: Basic architectural model for the management of replicated data.

2. **Coordination** : Replica managers decide whether or not the request can be applied, i.e. the request will not introduce inconsistencies, and how the requests will be ordered. Possible ordering policies are for example FIFO, causal or total ordering.

3. **Execution** : The request is executed by the replica managers.

4. **Agreement** : The replica managers decide whether or not the commit the results of the request.

5. **Response** : One or more replica managers communicate the result to the front end.

6.1.1 Group views

The size of the set of replica managers may be be constant, i.e. membership is static, or vary, i.e. membership is dynamic. To manage this kind of groups, the group communication paradigm is often applied; particularly in the case of dynamic membership where the join and leave operations are concerned [1].

To manage groups, *group views* are used. These are ordered lists of the current group members. Each group member has a unique process identifier. Group views are generated as members join or leave, after processes are notified of changes through view delivery. Correct view delivery requires a number of guarantees to be met [1]:

- **Order** : If a processe delivers views v(g) and v'(g), then no other process will deliver v'(g) before v(g).

- **Integrity** : If a process delivers a view, then that process is part of the view.

- **Non-triviality** : A process q that is indefinitely reachable from a process p will always be in the views that p delivers.

In the case of *view-synchronous group communication* additional constraints are to be met. **Figure 2** gives an overview of allowed and disallowed scenarios based on the following requirements [1]:

- **Agreement** : Correct processes deliver the same sequence of views and set of messages within any given view.

- **Integrity** : If a correct process delivers a message m, this process will not deliver m again.

- **Validity** : Correct processes always deliver the messages that they send.

[gallery columns="2" ids="112,113,114,115"]
View-synchronous group communication can be used to perform state transfer from the current group to new members. To ensure that the transferred state is not corrupt, the execution is usually temporary suspended. When the transfer is complete, the coordinator sends a message to the group members to continue.

The goal of group views is to increase fault-tolerance and transparency. As members crash or become unreachable, they are marked "suspicious" and may be excluded from the group by the membership service. This introduces a design challenge, as when excluding processes that are falsely excluded, resources and processing power may be (temporary) lost [1].

Another design decision has to be made in how to handle network partitions. Two general approaches exist: either the group is reduced, keeping only the primary-partition, or the group is partitionable into subgroups that can continue working independently [1].

6.1.2 Fault tolerance

The goal of fault tolerant systems is to "provide a service that is correct despite up to f process failures" [1]. This can be achieved by replicating data and functionality at replica managers. Correctness of replicated objects is subject to a number of criteria, which can vary in strictness.

Linearizability is a strong correctness requirement. Consider a sequence of operation invocations and responses, called a *history*, o<sub>2,0</sub>, o<sub>2,1</sub>, o<sub>1,0</sub>, o<sub>2,2</sub>, o<sub>1,1</sub>, o<sub>1,2</sub>, ..., where *i* represents a client performing an operation *j* for an operation o<sub>i,j</sub>. **Figure 3 (a)** shows an overview of this setup. "A replicated shared object is linearizable if for any execution there is some interleaving of the series of operations issued by all clients that satisfies the following criteria" [1] :

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

Figure 3 (b) shows how the operations in the history can be re-ordered. If there exists an ordering for which the previous conditions are met, the system is linearizable. In **figure 3 (c)** the new ordering did not meet the requirement.
[gallery columns="3" ids="149,150,151"]
Sequential consistency is an example of a weak correctness critrium. The requirements for sequential consistency are the following:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

As a result, the situation in **figure 3 (c)** is valid under a sequential consistency requirement. The absolute times are not important top obtain sequential consistency, just the order of events corresponding the clients seperately. It should be clear that sequential consistency is a much weaker constraint than linearizability. There may still be inconsistencies in the overal history, despite equential consistency. For example [2] when two clients try to lock an object, one lock will be successful and the other won't. In the case of sequential consistency, it is possible that the negative response is ordered before the successful response, which is not consistent with the sequential definition of the object, i.e. the first one to lock should have gotten a response of success.

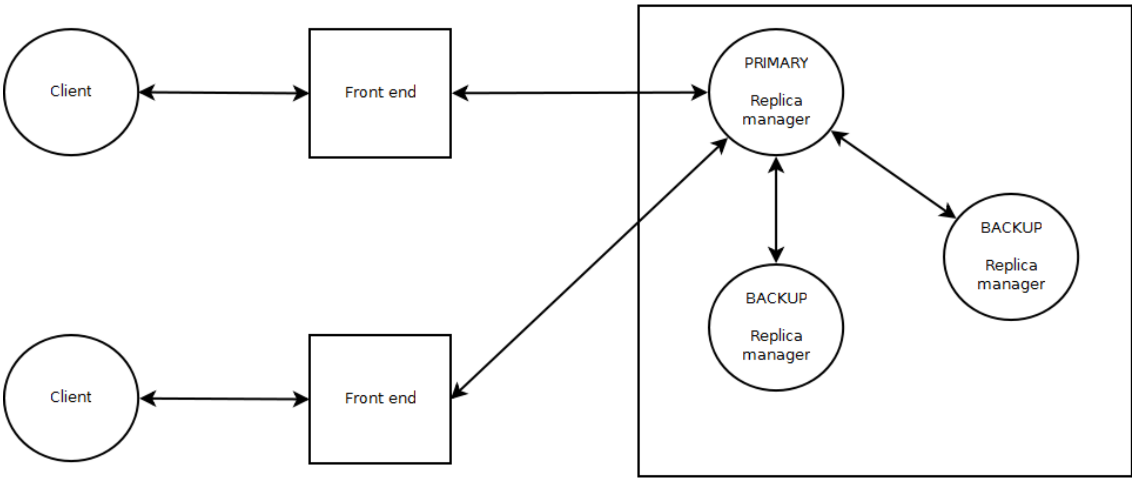

Figure 6.2: Passive replication model for fault tolerance.

Passive replication

The model for *passive replication*, the so-called *primary backup model of replication for fault tolerance*, consists out of primary replica manager and a collection of secondary replica managers or "backups". All operations are processed by the primary replica manager and afterwards changes are propagated to the backups. If the primary replica manager should fail, one of the backups is promoted to primary replica manager [1]. **Figure 4** shows an overview of this model.

The following steps occur when an operation is performed by a client [1]:

1. **Request** : The front end issues the request, containing a unique identifier, to the primary replica manager.
2. **Coordination** : The primary takes each request anatomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so, it simply resends the response.
3. **Execution** : The primary executes the request and stores the response.
4. **Agreement** : If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. **Response** : The primary responds to the front end, which hands the response back to the client.

The primary sequences all the operations upon the shared objects, so as long as the primary is correct, the system is linearizable. In the case that the primary fails, however, to ensure linearizability, the primary replica manager has to be replaced by a unique backup, and the remaining replica managers agree which operations had been performed when the replacement primary takes over. This will be the case if the replica managers apply view-synchronous group communication to send updates to the backups [1].

Active replication

In *active replication* the replica managers are state machines with equivalent status and organized as a group. The front end multicasts requests to the group. Within the group each manager processes the requests independently in the same manner and reply individually to the front end [1]. **Figure 5** shows the model for active replication.

The following steps occur when an operation is performed by a client [1]:

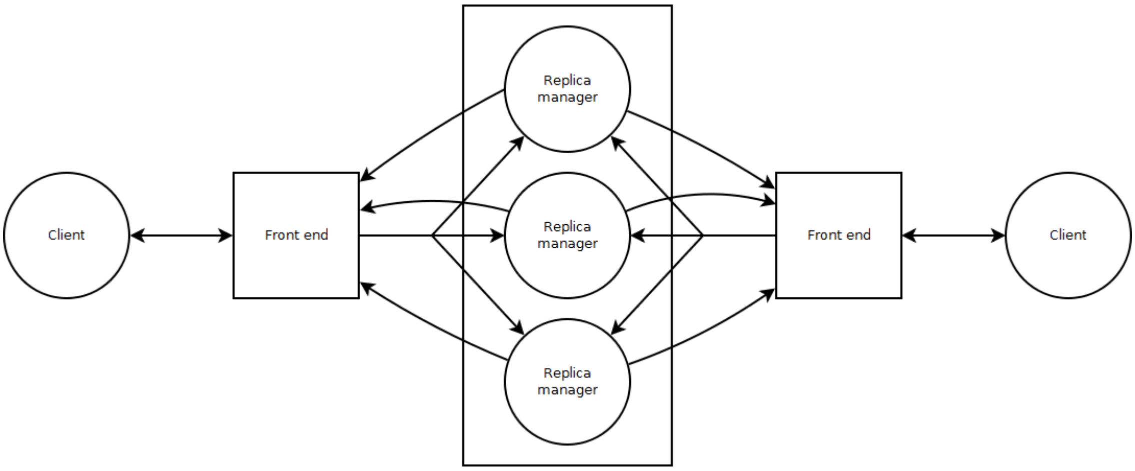

Figure 6.3: Active replication model for fault tolerance.

1. **Request** : The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst.It does not issues the next request until it has received a response.

2. **Coordination** : The group communication system delivers the request to every correct replica manager in the same (total) order.

3. **Execution** : Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

4. **Agreement** : No agreement phase is needed, because of multicast delivery semantics.

5. **Response** : Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm.

The system is sequentially consistent, but does not achieve linearizability since the total ordering is not necessarily the same as the real-time order in which the clients made their requests [1].

Crashes of replica managers have little impact on the performance in active replication, as the remaining replica managers continue to work as usual. Because the front end can compare the replies it receives, the system is less prone to Bynzantine failures [1].

Comparison: active and passive replication

6.2 Highly available services: examples of systems using replication

6.2.1 The Coda file system

The Coda file system is based on the Andrew File System (AFS). It was developed to address additional requirements for file systems, in particular to provide high availability in presence of disconnected operations. Of course, Coda retains the original goals of AFS with regard to scalability and the emulation of UNIX file semantics. Coda tries to meet the following three additional requirements under the general heading of constant data availability [1,2]:

Table 6.2: Comparison between passive and active replication.

	Passive	**Active**
Correctness	Support for linearizability.	Support up to sequential consistency.
Fault tolerance	The system requires f+1 replica managers to survive up to f process crashes.	For f Byzantine failures, the system requires 2f+1 replica managers to ensure that the system continues to function correctly. This is because the front collects f+1 replies before passing on the results to the client.
Efficiency	View-synchronous group communication is required to support linearizability, but introduces a significant overhead. In a variation where read requests are handled by backups to improve performance, the system looses its linearizability property, but maintains sequential consistency.	As managers work independently within te group, crashes have no impact on efficiency. The group communication is relatively cheap as no view-synchronous communication is required.

1. **Performance** : In large-scale distributed systems this availability requirement becomes more important, as the limited form of replication offered by AFS on read-only volumes doesn't scale well for accessing widely shared files.
2. **Fault tolerance** : The availability of AFS services was to be improved as failures (or scheduled interruptions) of servers and network components could make these services inaccessible for significant periods of time.
3. **Disconnect operations due to mobility** : Finally, mobile computers disconnect and reconnect frequently leading to an availability requirement of files the user may need despite being disconnected [1].

The design of Coda relies on the replication of file volumes to achieve a higher throughput of file access operations and a greater degree of fault tolerance. Coda also makes use of AFS's client caching extension [1]. Coda enhances availability both by [1]:

- Replication of files across servers. The advantages of replicating file volumes on multiple servers are [1]:
  - As long as at least one replica is accessible the client can access the files in that replicated volume.
  - System performance can be improved by sharing some of the load, i.e. client requests on replicated volumes, between the servers holding replicas.
- The ability of clients to operate entirely out of their caches. Coda will try to predict which files will be needed by a user and cache them in case of disconnection with the network.

The Coda architecture

Vice (server) and Venus (client) processes **Figure 6** shows an overview of the Coda file system architecture. Similar to AFS, Coda runs *Venus* processes at the client computers and *Vice* processes at file server computers. The Vices are the replica managers, and the Venuses are a hybrid of front ends and replica managers. A Venus plays the front end's role of hiding the service implementation from local client processes, but since they manage a local cache of files they are also replica managers.

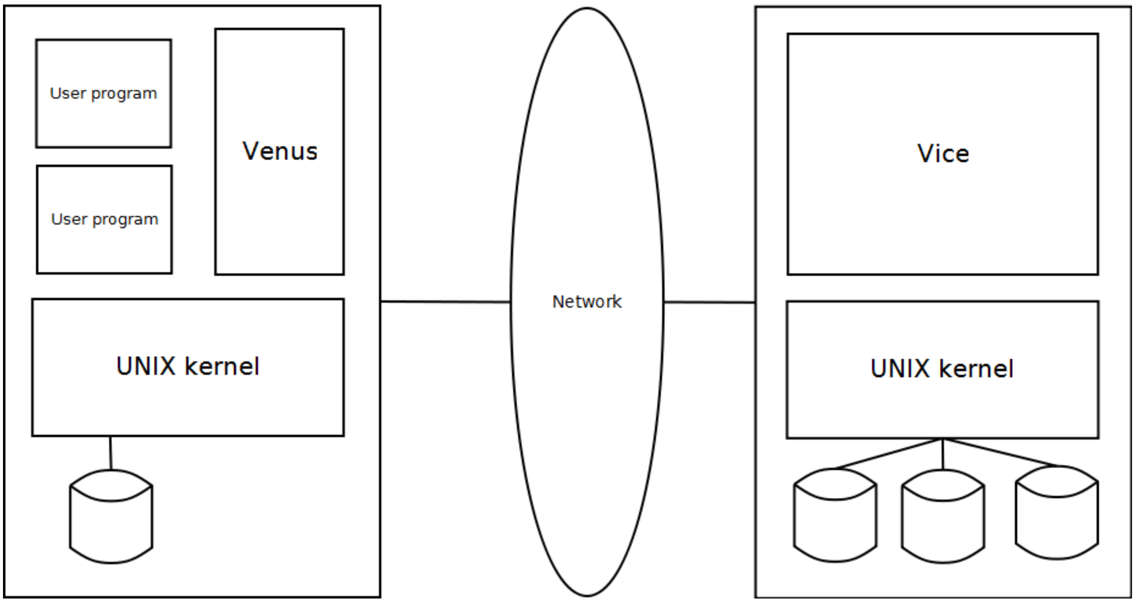

Figure 6.4: Coda file system architecture (AFS-based).

The volume storage group (VSG) A volume storage group (VSG) is the set of servers that holds replicas of a file volume. As servers become inaccessible as a result of network or server failures, the client can usually only access a subset of the VSG, known as the available volume storage group (AVSG). A disconnected operation occurs when the AVSG is empty [1].

Replication and consistency

The Coda servers are the focal point to provide quality of service, as cached file copies residing on client computers are regarded as less reliable than those at the servers. These files are periodically revalidated against the version at the servers. In presence of network partitions updated files may cause conflicts with other replicas when the network is restored [1].

Coda uses an optimistic replication strategy, i.e. files can be modified in presence of network partitions or during disconnected operations.

Coda version vector Each version of a file has a *Coda version vector* (CVV) containing a timestamp with one element for each server in the corresponding VSG [1]. When a modified file is closed, the Venus process sends an update message with the current CVV and the new contents for the file to the AVSG. If the CVV is greater than the one currently held at the AVSG, the new contents for the file are stored and a positive acknowledgement is returned. The Venus process then computes a new CVV with modification counts increased for the servers that responded positively to the update message and distributes the new CVV to the members of the AVSG. Note that AVSG is only a subset of the VSG, so possibly not all members will receive the new CVV [1].

The CVV can be used for resolving file conflicts as follows. For timestamps of two CVVs v_1 and v_2, there are two general cases [1]:

- If the CVV at one of the sites is greater than or equal to all the corresponding CVVs at the other sites then there is no conflict.
- When neither $v_1 \leq v_2$ nor $v1 \geq v_2$ holds for two CVVs then there is a conflict: each replica reflects at least one update that the other does not reflect. Coda does not, in general, resolve conflicts automatically. The file is marked as *inoperable* and the owner of the file is informed of the conflict.

Table 6.3: File open and close operation in Coda.

Operation	**Description**
open	This operation consists of the following steps. If a copy of the file is not present in the local cache: 1. The client choses a preferred server from the AVSG for the file. 2. The client requests a copy of the file attributes and contents from the preferred server. 3. The client checks with all the other members of the AVSG to verify that the copy is the latest available version. If not, a member of the AVSG with the latest version is made the preferred site, the file contents are refetched and the members of the AVSG are notified that some members have stale replicas. 4. When the fetch has been completed, a callback promise is established at the preferred server.
close	On close, copies of modified files are broadcast in parallel to all of the servers in the AVSG using *multicast remote procedure calling protocol.* This has two notable effects: 1. It maximizes the *probability* that every replication site for a file has the current version at all times. It doesn't guarantee it, because the AVSG does not necessarily include all the members of the VSG. 2. It minimizes the server load by giving clients the responsibility for propagating changes to the replication sites in the normal case.

Accessing replicas and update semantics AFS's original *callback promise* mechanism, is extended and depends on an additional mechanism for the distribution of updates to each replica. The strategy used on open and close on the replicas is a variant of the *read-one/write-all* approach [1]. **Table 3** lists the open and close operation specifics.

The update semantics for Coda are a little different to those in AFS. The single server S referred to in the currency guarantees for AFS is replaced by a set of servers S, i.e. the VSG for a file F and the client C can access a subset of servers s, i.e., the AVSG for that file seen by C. T is again the maximum time that for which a client can remain unaware of an update elsewhere to the cached file [1]. In addition the following predicates are used:

- $latest(F, s, T)$: The current value of F at C was the latest across all the servers in s at some instant in the last T seconds and that there were no conflicts among the copies of F at that instant;
- $lostCallback(s, T)$: A callback was sent by some member of s in the last T seconds and was not received at C;
- $conflict(F, s)$: The values of F at some servers in s are currently in conflict.

The currency guarantees are then summarized as follows [1]. In each definition except the last there are two cases [1]:

1. $s \neq \emptyset$: The AVSG is not empty, i.e., the client is not disconnected.

2. $s = \emptyset$: The disconnected operation.

After succesfull open : The guarantee offered by a successful open is that either that the most recent copy of F is provided from the current AVSG, or a locally cached copy of F is used if one is available, if no server is accessible:
$(s \neq \emptyset \wedge (latest(F, s, 0) \vee (latest(F, s, T) \wedge lostCallback(s, T) \vee inCache(F))))$
$\vee$
$(s = \emptyset \wedge inCache(F))$

After failed open :
$(s \neq \emptyset \wedge conflict(F, s)) \vee (s = \emptyset \wedge \neg inCache(F))$

After succesfull close : A successful close guarantees that the file has been propagated to the currently accessible set of servers, or, if no server is available, that the file has been marked for propagation at the earliest opportunity.
$(s \neq \emptyset \wedge updated(F, s)) \vee (s = \emptyset)$

After failed close :
$(s \neq \emptyset \wedge conflict(F, s))$

Caching

Cache coherence The Coda currency guarantees stated earlier mean that the Venus process at each client must detect the following events within T seconds of their occurrence [1]:

- **AVSG size changes** : Enlargement of an AVSG (due to the accessibility of a previously inaccessible server), or shrinking of an AVSG (due to a server becoming inaccessible);

- **Callback event loss** : Since maintaining callback state in all the members of an AVSG would be expensive, the callback promise is maintained only at the preferred server. However, the preferred server for one client need not be in the AVSG of another client. If this is the case, an update by the second client will not cause a callback to the first client.

AVSG size changes Venus (client) sends a probe message to all the servers in VSGs of the files that it has in its cache every T seconds. Responses will be received only from accessible servers. The following cases can be distinguished with regard to callback reponses:

- **Venus receives a response from a server that was previously inaccessible** : Venus enlarges the corresponding AVSG. As the cached copy may no longer be the latest version available, Venus will also drop the callback promises on any files that it holds from the relevant volume.

- **Venus fails to receive a response from a previously accessible server** : Venus shrinks the corresponding AVSG. Unless the preferred server is lost, no callback changes are required.

- **The preferred server is lost** : All callback promises from that server must be dropped.

- **A response indicates that a callback message was sent but not received** : The callback promise on the corresponding file is dropped.

Disjunct AVSGs Venus is sent a *volume version vector* (volume CVV) in response to each probe message, containing a summary of the CVVs for all of the files in the volume. If Venus detects any mismatch between the volume CVVs, then some members of the AVSG must have some file versions that are not up-to-date. Although the outdated files may not be the ones that are in its local cache, Venus makes a pessimistic assumption and drops the callback promises on all of the files that it holds from the relevant volume [1].

Cache miss and disconnected operation In disconnected operation (when none of the servers for a volume can be accessed by the client) a cache miss prevents further progress and the computation is suspended until the connection is resumed or the user aborts the process. It is therefore important to load the cache before disconnected operation commences so that cache misses can be avoided [1].

Selecting files to retain in the cache If a client is disconnected for an extended period of time, it is likely that files or directories will be referenced that are not in the cache. To alleviate this problem Coda allows users to specify a prioritized list of files and directories that Venus should strive to retain in the cache. Objects at the highest level are identified as *sticky* and these must be retained in the cache at all times. If the local disk is large enough to accommodate all of them, the user is assured that they will remain accessible [1].

Reintegration after disconnect operation When disconnected operation ends, a process of reintegration begins. For each cached file or directory that has been modified, created or deleted during disconnected operation, Venus executes a sequence of update operations to make the AVSG replicas identical to the cached copy. Reintegration proceeds top-down from the root of each cached volume [1].

6.2.2 Gossip framework

The goal of the Gossip framework is implementing highly available services by replicate data close to points where groups of clients need it [3]. Gossip is not fault tolerant. A framework implies that it is configurable with multiple degrees of freedom. The consequence of this is that it can be used for a variety of applications [3].

The front end sends operations to any RM that is available and has a reasonable response time. There are two types of operations [3]:

- **Queries** : Read-only operations;
- **Updates** : Change state

6.2.3 Bayou

The goals of the Bayou model are data replication for high availability, but with weaker guarantees than sequential consistency, and coping with variable connectivity [3].

6.3 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

[2] Wikipedia, 2013, "Linearizability | Wikipedia", online, available at: http://en.wikipedia.org/wiki/Linearizability

[3] W. Joosen, 2013, "Distributed Systems: Replicated Data - Part 1", iMinds-DistriNet KULeuven

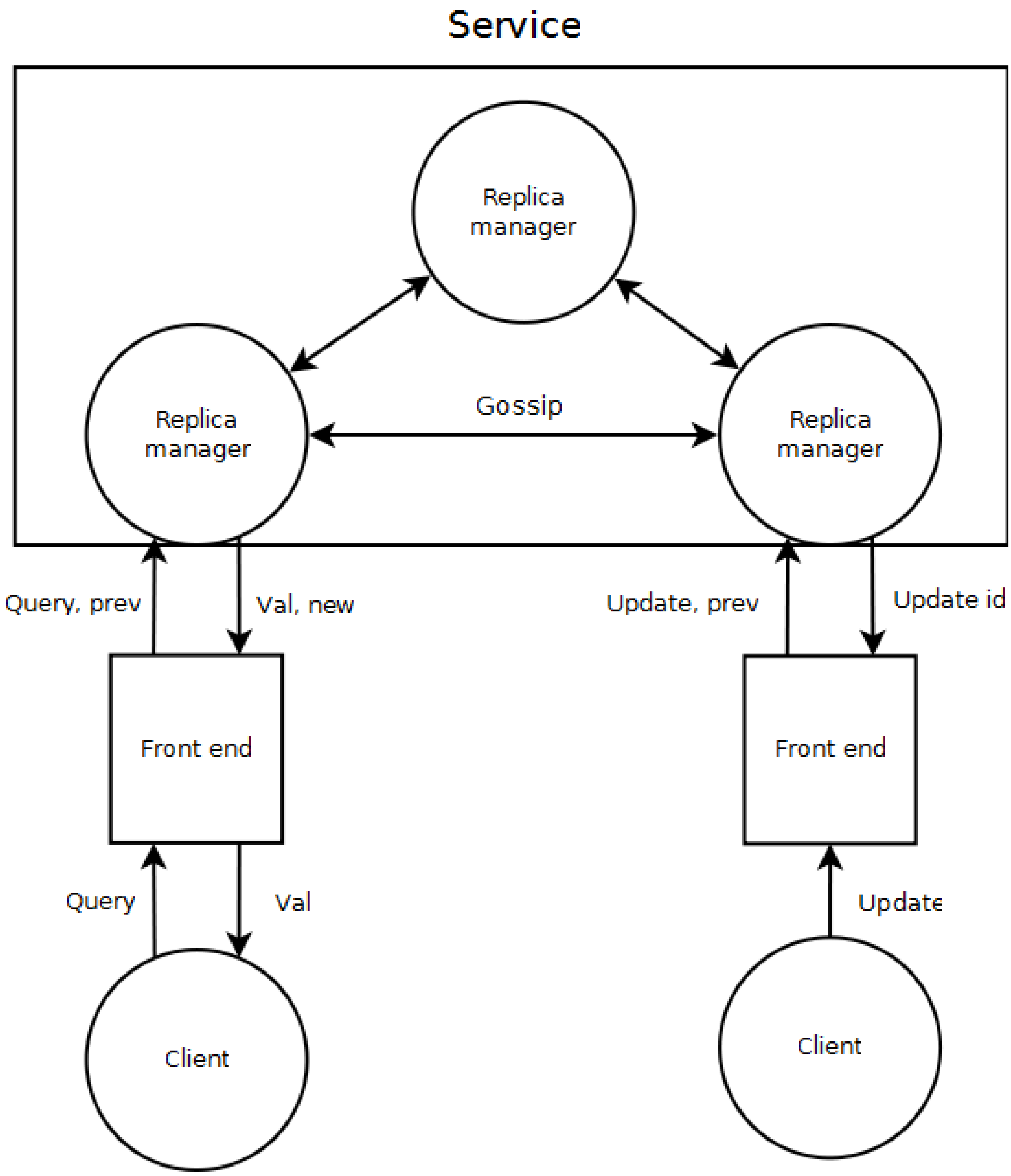

Figure 6.5: The Gossip framework architecture.

Chapter 7

Cloud computing

7.1 What is cloud computing?

7.1.1 Characteristics

Characteristics [2]:

- **On-demand self-service** : A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider;
- **Broad network access** : Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms, e.g., mobile phones, tablets, laptops, and workstations;
- **Resource pooling** : The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction, e.g., country, state, or datacenter. Examples of resources include storage, processing, memory, and network bandwidth;
- **Rapid elasticity** : Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time;
- **Measured service** : Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service;

Outsourcing requires trust from clients.
A number of advantages are associated with cloud computing [2]:

- Accelerated deployment of new applications without consuming enterprise's existing IT resources;
- Reduced capital requirements for up-front IT investments;
- Flexibility to meet sudden changes in demand peaks and troughs;

- Capability to match current and future demand;
- Significant cost savings through centralization when scale of enterprise IT resources « cloud provider;
- Data sharing and collaboration for multi-party processes. More economical and faster to deploy centrally;

7.1.2 Business models

Cloudcomputing is typically a pay per use model for on-demand, convenient access to shared pool of computing resources, e.g., storage, CPU, network, and applications. There are three basic types of services in the cloud computing model [2]:

1. **Infrastructure as a service (IaaS)** : virtual machine with processing, storage and networking;
2. **Platform as a service (PaaS)** : development platform and associated tools, e.g., PHP, .NET, Java;
3. **Software as a service (SaaS)** : Zero-install, online applications, e.g., CRM, document processing platforms, application specific record management etc.;

Infrastructure as a Service (IaaS)

"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)." [3]

Platform as a Service PaaS

"A cloud service model that provides the consumer the capability to deployonto the cloud infrastructure consumer-created or acquired applicationscreated using programming languages, libraries, services, and tools supported by the provider.1The consumer does not manage or control the underlying cloud infrastructureincluding network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment." [3]

Software as a Service SaaS

"The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure(*). The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings." [3]

7.1.3 Value levels of cloud computing

In [4] three value levels of cloud computing are discussed:

- **Utility level** : Enterprises can benefit from lower costs and higher service levels through the availability of elastic computing resources and pay-per-use models;

- **Process transformation level** : Enterprises can introduce new and improved business processes by leveraging the common and scalable assets and collaborative potential of cloud computing;
- **Business model innovation level** : New business models can be created by linking, sharing and combining resources using cloud computing in an entire business ecosystem;

7.2 References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems: Concepts and Design (5th Edition)", M. Horton, Red., Addison-Wesley, 2011, p. 1063.

[2] Wouter Joosen, 2013, "Perspectives on Cloud Computing", iMinds-DistriNet, KU Leuven

[3] P. Mell and T. Grance, 2011, "The NIST Definition of Cloud Computing", online, available at: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[4] D. Dean and T. Saleh, 2009: "Capturing the Value of Cloud Computing: How Enterprises Can Chart Their Course to the Next Level", BCG -http://www.bcg.be/documents/file34246.pdf

Bibliography

[1] Distributed Systems: Concepts and Design (5th Edition). *G. Coulouris and J. Dollimore and T. Kindberg and G. Blair.* Addison-Wesley, 2011.

[2] W. Joosen. Distributed file systems, 2013.

[3] W. Joosen. Distributed systems - transactions - i, 2013.

[4] W. Joosen. Distributed systems : Transactions - part 2, 2013.

[5] W. Joosen. Distributed systems direct communication part i, 2013.

[6] W. Joosen. Distributed systems direct communication part ii, 2013.

[7] W. Joosen. Distributed systems: Replicated data - part 1, 2013.

[8] Wikipedia. Linearizability, 2013.

List of Figures

List of Tables