



Master of engineering thesis in computer science

---

Asynchronous Lock Distribution  
in the New File Repository

---

André Larsen Risnes  
Department of Computer Science  
The University of Tromsø  
andrer@stud.cs.uit.no

DECEMBER 14, 2000



## Abstract

When users work on data replicated on several machines, the data may be inconsistent if updates are not controlled by some protocol. One class of protocols that handles this is called *atomic commit protocols* (ACP). The *two-phase commit* (2PC) protocol is a well-known, simple and elegant ACP.

In the future it is expected that the machines containing replicated data may be portable, and therefore often disconnected from the network for arbitrary amounts of time. Then we will need an *asynchronous* 2PC protocol that does not rely on any assumptions about message delays.

This report describes the theory behind, the requirements to, the design and implementation of a proof-of-concept version of an asynchronous message delivery system and 2PC protocol for the *New File Repository*, a distributed storage architecture.

## Background

This semester assignment report is the final part of my masters degree<sup>1</sup> study in computer science at the University of Tromsø. It is a part of the Pasta project [1], which itself is a part of the Global Distributed Diary (GDD) project. My supervisor has been Tage Stabell-Kulø.

The Pasta project addresses the consistency problems that occur when data is replicated in a distributed system with mobile machines characterized by frequent disconnection and varying communication capability. The research vehicle used is a distributed storage architecture called the “New File Repository” (NFR).

The GDD projects goal is to study the consistency problems related to integration of mobile and fixed computers in computer networks.

The form “we” is used throughout the report to avoid a too personal style, which means that “we” may in most cases be replaced with “I” since this report has no co-authors.

## Acknowledgments

I would like to thank my supervisor Tage Stabell-Kulø for an interesting project and helpful hints and advice, the people in the Pasta lab for making room for yet another student and helping with everything from networking to L<sup>A</sup>T<sub>E</sub>X, and in particular Feico Dillema for working so hard to finish the NFR modules my system is built on. I would also like to thank my employer Kolbjørn Engeseth for allowing me a very flexible work schedule.

---

<sup>1</sup>The international equivalent of the Norwegian title “sivilingeniør”.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background . . . . .	4
1.2	The problem . . . . .	4
1.3	Method . . . . .	5
1.4	Conventions and definitions . . . . .	5
1.5	Organization of the rest of the report . . . . .	6
<b>2</b>	<b>Theoretical framework</b>	<b>7</b>
2.1	Network and failure models . . . . .	7
2.2	Electronic mail . . . . .	8
2.3	Replicated data . . . . .	10
2.4	The Open-End Argument . . . . .	10
2.5	The New File Repository . . . . .	11
2.6	Transaction protocols . . . . .	14
2.6.1	The Two-Phase Commit Protocol . . . . .	15
2.6.2	Failure scenarios and a recovery protocol for 2PC . . .	17
2.7	Concurrency control mechanism . . . . .	18
2.8	Conclusion . . . . .	20
<b>3</b>	<b>Requirements specification</b>	<b>21</b>
3.1	Non-functional requirements . . . . .	21
3.2	Overall system description . . . . .	22
3.3	Communication subsystem . . . . .	23
3.4	NFR interface . . . . .	24
3.5	Protocol subsystem . . . . .	24
3.6	Conclusion . . . . .	24
<b>4</b>	<b>Design</b>	<b>25</b>
4.1	Protocol subsystem . . . . .	25
4.2	Communication subsystem . . . . .	27
4.2.1	The NFR command set . . . . .	29
4.3	NFR Interface . . . . .	30
4.4	Conclusion . . . . .	31
<b>5</b>	<b>Implementation</b>	<b>32</b>
5.1	Environment . . . . .	32
5.1.1	Existing NFR modules . . . . .	33
5.2	Structure . . . . .	33
5.2.1	Full class hierarchy . . . . .	34
5.3	CommInfo module . . . . .	34
5.4	Util module . . . . .	36
5.5	Termoutput module . . . . .	36

5.6	Command module . . . . .	36
5.6.1	Uniform Resource Locators . . . . .	38
5.7	Server module . . . . .	39
5.7.1	The MailServer class . . . . .	41
5.8	Port module . . . . .	42
5.8.1	Recovery . . . . .	43
5.8.2	Command handling . . . . .	43
5.9	Protocol module . . . . .	44
5.9.1	Protocol initiation . . . . .	45
5.9.2	Logs . . . . .	46
5.9.3	Protocol execution . . . . .	47
5.9.4	Duplicate Commands . . . . .	49
5.9.5	Recovery . . . . .	49
5.10	Conclusion . . . . .	50
<b>6</b>	<b>Testing</b>	<b>51</b>
6.1	Test configuration . . . . .	51
6.2	Demonstration . . . . .	52
6.3	Performance . . . . .	54
6.4	Failure and recovery . . . . .	55
6.4.1	Participant failure scenario 1 . . . . .	55
6.4.2	Participant failure scenario 2 . . . . .	56
6.4.3	Participant failure scenario 3 . . . . .	56
6.4.4	Participant failure scenario 4 . . . . .	57
6.4.5	Participant failure scenario 5-1 . . . . .	57
6.4.6	Participant failure scenario 5-2 . . . . .	57
6.4.7	Coordinator failure scenario 2 . . . . .	57
6.4.8	Coordinator failure scenario 3 . . . . .	58
6.4.9	Coordinator failure scenario 4 . . . . .	58
6.5	Conclusion . . . . .	59
<b>7</b>	<b>Conclusion</b>	<b>60</b>
7.1	Accomplishments . . . . .	60
7.2	Future work . . . . .	60
<b>A</b>	<b>The CD-ROM</b>	<b>64</b>

## List of Figures

1	Process failure models . . . . .	8
2	The Two-Phase Commit protocol . . . . .	16
3	Classifications of concurrency control mechanisms. . . . .	19
4	Strict two-phase locking. . . . .	19
5	Overview of the system and its environment. . . . .	23
6	STD for the protocol subsystem. . . . .	26
7	STD for the communication subsystem. . . . .	28
8	OO Diagram conventions. . . . .	32
9	Mapping from subsystem to modules. . . . .	33
10	Full class/module hierarchy . . . . .	35
11	The comminfo and Util modules. . . . .	35
12	Classes in the command module. . . . .	37
13	Example MailCommand rendered as a string. . . . .	38
14	The server module . . . . .	40
15	The port module . . . . .	43
16	The protocol module . . . . .	45
17	The demonstration setup. Email messages follow the arrows: from the participants/coordinator, to Sendmail, to Qpopper, and back to the participants/coordinators. . . . .	51

# 1 Introduction

## 1.1 Background

Two of the most visible developments in the computer industry are the decrease in cost of computer hardware and the increase in network bandwidth and availability. We expect these trends to continue and to introduce many new possibilities, for example that each person can afford to own several computers, like a PC at home, another at work, a PDA and maybe a laptop. These computers will be connected to the Internet in various ways with varying bandwidth and connectivity, allowing the owner to use a computer whenever he wants. This will introduce at least one challenge: If this person wants to work with the same data on several of his machines or wants to share this data with others (i.e. replicate the data), it will become inconsistent if it is simultaneously modified on two or more computers.

This is a well-known problem with many different solutions, but few of the existing solutions take into account large-scale usage of computers that are often unavailable to the network because of varying communication capability. This is (part of) the focus of the Pasta project, under which this report is written. The Pasta project has a research vehicle under development called the New File Repository (NFR). The primary task of NFR is to provide a distributed storage architecture which is to operate successfully under the conditions described above. In this report we hope to present a solution to the consistency problem which is usable by NFR.

## 1.2 The problem

As a consequence of the extensive use of portable machines in the New File Repository, large parts of the system may be unavailable at any time.

The user may choose to replicate files to any machine where he/she has access. To keep the replicated files consistent (to some standard defined by the user and his applications), access to data in NFR must be controlled by some concurrency control protocol. NFR will provide a lock distribution mechanism that this protocol can use. To guarantee atomic distribution of locks, we need an *atomic commit protocol*. Two-phase commit (2PC) is a well known protocol of this type.

2PC is usually designed for synchronous environments with some expectations to the availability of the participants. In the New File Repository we will have neither, and we need to know if an asynchronous 2PC protocol operating over an asynchronous transport mechanism (like email) will be able to provide the lock distribution that concurrency control protocols require.

This report describes the theory behind, the requirements to, the design and implementation of this protocol and an asynchronous message system for the New File Repository.

### 1.3 Method

We have used an evolutionary development method, with several cycles of research, requirements specification, design, implementation and testing. The progress of this method is difficult to describe clearly in an academic report, however. So, for clarity, the result will be presented as if we had followed the linear sequential model<sup>2</sup> [12]:

1. **Software requirements specification:** Find the required function, behavior, performance of and interface to the program.
2. **Design:** Translate the requirements into a description of the program that satisfies the requirements.
3. **Implementation:** Translate the design into a program that runs in the environment described in the software requirements specification.
4. **Testing:** Check if the program actually satisfies the requirements.

Note that the evolutionary model implies a relatively loose design specification, with details of the system described in the implementation.

### 1.4 Conventions and definitions

A Note about conventions used in this document: Capitalized names, like “File”, denote abstractions in NFR, uncapitalized names, like “file”, denote their respective real-world counterparts. Fixed text or a capitalized name followed by “object”, like “File” or “File object” denote classes or objects in our design/implementation, and names in *italic* denote definitions and first-time usage of an important term.

Some often used acronyms and definitions are listed for reference:

*2PC*: Two-Phase Commit (see section 2.6.1)

*ACP*: Atomic Commit Protocol (see section 2.6).

*DFD*: Data Flow Diagram.

*Email*: Short for electronic mail (see section 2.2).

*MTA*: Mail Transfer Agent (see section 2.2).

*NFR* The New File Repository. (see section 2.5).

*POP3* The Post Office Protocol, as defined in RFC 1939 [16] (See section 2.2)

---

<sup>2</sup>Or: The waterfall model.

*RFC*: Request For Comments, a type of document that describes an Internet standard.

*SMTP*: The Simple Mail Transfer Protocol, as defined in RFC 821 [14] (See section 2.2).

*STD*: State Transition Diagram.

*TCP*: Transmission Control Protocol, a reliable, connection-oriented bytestream protocol that runs on top of a network-layer protocol (e.g. the Internet Protocol, IP).

*UDP*: User Datagram Protocol, a protocol for sending single packets called datagrams over a network-layer protocol. Unlike TCP, it is not connection-oriented.

## 1.5 Organization of the rest of the report

The rest of the report is organized as follows:

Section two presents our theoretical framework.

Section three presents the requirements to our system, with a focus on behavioral specification.

Section four contains the design: Discussion of alternatives that satisfy the requirements, and the design choices we have made.

Section five describes the implementation.

Section six contains a demo, describes what tests we have done, their results and how to reproduce them.

Finally, section seven contains our conclusion, with achievements and suggestions to future work.

## 2 Theoretical framework

Protocols for keeping replicated data objects synchronized have been studied extensively in the past, and it is beyond the scope of this report to provide a comprehensive overview of the field. What we will do in this section is to present the basic theory behind our asynchronous messaging system, describe the New File Repository and focus on a 2PC protocol plus recovery protocol that is well suited for our needs. We conclude this section with a brief description of how our 2PC protocol can be used to implement an application level concurrency control protocol.

### 2.1 Network and failure models

In this section we will define some important concepts in the asynchronous network model, and the failure models of an asynchronous distributed system.

A *network* is a graph where the vertices are called *nodes* (or *sites*) and the edges called *channels*. Each node contains a *process* that executes some task, and each channel is used to send *messages* from one process/node to another [13].

Such a system is *synchronous* if there are known maximum upper bounds on process execution times and message delivery delays. If not, the network is *asynchronous* [8]. The transport layer in the Internet is an example of an asynchronous system.

A system that uses both processes and channels is called a *distributed system*<sup>3</sup> [13].

An asynchronous *store-and-forward* channel may wait an arbitrary amount of time from a message has been sent on the channel until it is delivered to the recipient, and, if the message has been successfully and completely sent on the channel, it is still delivered even if the sending process fails. An example is electronic mail over the Internet, the subject of our next section.

A process is *faulty* if its behavior deviates from that prescribed by the algorithm it is running. Otherwise, the process is *correct*. A *failure model* specifies in what way the process deviates from its algorithm. Commonly considered failure models of asynchronous systems are listed below [7]:

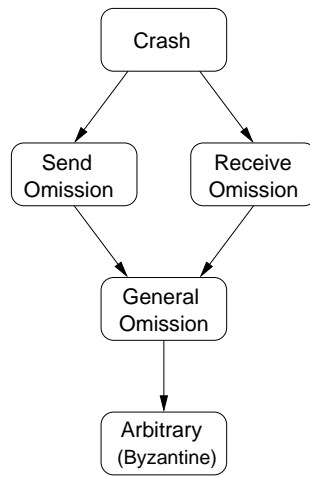
1. *Crash*: The process stops and does nothing from that point.
2. *Send omission*: The process crashes or omits to send messages that it is supposed to send.
3. *Receive omission*: The process crashes or does not receive messages sent to it.

---

<sup>3</sup>Or, “several computers doing something together”

4. *General omission*: The faulty process is subject to send omissions, receive omissions, or both.
5. *Arbitrary* (sometimes called *Byzantine*): The faulty process can exhibit any behavior, including malicious actions that will cause the system to fail.

Figure 1: Process failure models



One model is more *severe* than another if the set of faulty behaviors allowed by the second model is a subset of the behaviors allowed by the first [7]. Figure 1 shows the failure models in increasing level of severity.

Asynchronous communication channels can exhibit the following types of failures [7]:

1. *Crash*: The channel stops transporting messages.
2. *Omissions*: The channel omits to transport some of the messages sent through it.
3. *Arbitrary* (sometimes called *Byzantine*): The faulty channel can exhibit any behavior. For example, it can generate spurious messages.

The assumptions we make about our system are described in the requirements later in this article.

## 2.2 Electronic mail

Electronic mail, or email, is a system for sending messages over a network. In this report, “email” refers to a system that implements RFC 821 (transmission protocol, SMTP) [14] and RFC 822 (message format) [15], and “an

email” refers to a single message sent through this system. When an email arrives at its destination, it can be fetched by a user or user agent for example through the POP3 protocol. This section describes the relevant properties of such a system. Most of the material in this section is adapted from chapter 7.4 in [10].

RFC 822 compliant message consist of an “envelope” described in RFC 821, a number of header fields, a blank line, and then the message body. The envelope and header fields are on a “Field name: Field body crlf” form, where crlf is a carriage return and a newline character. They can be “folded”, allowing the field body to continue over several lines as long as the crlf is followed by a space or a horizontal tab character on the next line. Principal fields include the sender, the recipient(s), the date and the subject. Messages are limited to 7-bit ASCII text with a maximum line length of 1000 characters<sup>4</sup>. Binary messages can be sent safely if they are base64 encoded<sup>5</sup> (See for example section 5.2 in [18]).

SMTP is a simple ASCII line-oriented request/response protocol that runs on top of reliable byte-streams (e.g. TCP). It is used to send messages from user agents to Mail Transfer Agents (MTA) and from one MTA to another.

POP3 is a similar protocol, used by user agents to read and administrate mail boxes, the final destination of an email. An email moves from sender to recipient like this: The user agent contacts an MTA (which may be on the same machine as the user agent), and uses SMTP to send it the email. The MTA then contacts an MTA closer<sup>6</sup> to the recipient, and uses SMTP to forward the message. This continues until the email reaches an MTA that has access to the recipients mailbox, where the email is stored. Finally, the recipients user agent uses POP3 to fetch the email from his mailbox. Note that most new POP3 server implement the UDIL command, which returns a unique id for one or all messages in the mailbox. It can be used to determine what messages have arrived since the last time we checked the mailbox.

We can see that an email system on the Internet gives us an asynchronous store and forward channel. Email has the additional attractive property that messages are not removed from the channel until the recipient explicitly does so (unless there is a failure). One important thing to keep in mind about email is that messages regularly take minutes to arrive at their destination, so it is important that the selected two-phase commit protocol uses few messages and lets different sites execute the protocol concurrently. Also, emails do not necessarily arrive in the order they were sent, i.e. we do not have a FIFO channel. This is no problem because we do not need to remove the messages from the channel (mailbox) in the order they arrive.

---

<sup>4</sup>The ESMTP protocol described in RFC 1425 [17] is designed to remove these (and other) limitations in SMTP.

<sup>5</sup>Also called ASCII armor.

<sup>6</sup>Closer in the network, usually meaning that it is on the same subnet.

We do not know of any existing implementations of the two-phase commit protocol that uses email as a message carrier. The RelNet layer of SDD-1 [6] allows asynchronous communication by buffering messages for unavailable replicas, which is similar.

### 2.3 Replicated data

Replicating an object means storing copies of it on different sites. Replicating objects to the sites of a distributed systems gives us many advantages [4]:

- *Greater availability*, because we can access the replica that is closest to us.
- *Increased performance*, because each replica can be accessed independently and concurrently.
- *Independent operation*, an object can be accessed even if the node it is being accessed on is currently disconnected from the distributed system.

We also get redundant storage, although this rarely is the motivation for the replication. These advantages come at a cost, however [4]:

- *Extra storage*: Obviously, N replicas take up N times as much storage space, plus storage for control information.
- *Extra communication*: Depending on the protocol used to keep replicas up-to-date or equivalent, the communication cost of accessing a replicated object can be greater than accessing a non-replicated object.
- *Complexity of implementation*: Again, depending on policy and protocol, implementing a system that maintains replicas can be an expensive task.
- *Administration*: Some person must decide what objects to replicate, where the replicas should be, etc.

NFR is primarily a distributed storage architecture, implemented under the assumption that the benefits outweigh the disadvantages.

### 2.4 The Open-End Argument

Here is a summary of [3]. Distributed systems are designed as layers of abstractions, with peer entities communicating with each other over the network. The *end-to-end argument* states (among other things) that if one level of the system should enter an undesired state, this should be handled by

the level above. In a private computing environment the user is the highest level in the system.

Given the complex nature of such systems, they are usually designed to shield the user as much as possible from their inner workings and state. This is called the “transparency design principle”. It has been applied successfully to many systems, removing any requirement for the user to be competent to make decisions based on the internal state of the system, thus making it user-friendly. In private computing, however, the rigid application of the transparency design principle tends to decrease user-friendliness:

- Users will store “real-world” private assets like diaries, keys and money in their PDAs. They will not do this unless they *trust* the system. Trust is a feeling that can not be measured or quantified by the system, and is difficult for the user to specify in advance. The user can not be expected to trust the system unless he understands how it works.
- The user will fully own both the PDA and the private data on it, and needs to control the system rather than being “just a user”.
- PDAs often operate at the limits of what can be achieved by current technology, and operational problems due to resource shortage must be considered part of normal operation. Almost all failures will occur in what is normally considered “infrastructure”, and users are not normally supposed to interact with problems at such a low level. However, when operating a PDA each user must be involved in these issues in order to keep the system usable when parts of the system fail.

Given that users need to understand and control their systems, it is not acceptable for the system to simply stop functioning if there is an error that lower layers are unable to handle. When designing systems for private computing, the user can and should be the arbiter of what it is important for him to control. This is the *open-end argument*:

*The system should be designed in such a way that in all situations where qualitative assessment of information is needed to make a decision, the user is informed and consulted.*

This argument guides the system designer in choosing when to follow the transparency design principle and when to violate it by informing and consulting the end-user.

## **2.5 The New File Repository**

The New File Repository (NFR) is a system designed under the guidance of the open-end argument. NFRs primary task is to provide a distributed storage infrastructure. NFR aims for high data availability and proper access

control mechanisms. Design decisions are made with the open-end argument in mind. In practice this means that NFR tries to enforce user-specified policies, instead of making and enforcing policies of its own. Typically, the user will specify policies for the common case beforehand, and NFR will inform and consult the user when policy decision need to be made that are not covered by the common case policies.

Currently, two types of policies are enforced by NFR:

- *Availability*, determined by the number of replicas of the data kept by NFR on distributed servers. The replication policy specifies the (number of) replicas and the method of replication.
- *Accessibility*, which specifies the accessibility of data to users.

NFR has five core architectural abstractions to provide this functionality, each of which is identified by a globally unique Name:

*Names* are handles to reference particular objects in NFR. Names are *pure*, i.e. used only for identification purposes and containing no structure. They are currently implemented as 128-bit random numbers<sup>7</sup>. Names are generated solely by NFR servers.

*Files* are the unit of storage in NFR; a sequence of bytes with no associated type. Three operations are defined on Files:

1. Read, which returns all data in a file.
2. Write, which creates a new File with a new Name as a *child* of the File written to. Each File is part of a family tree of Files. The User is not prevented from creating multiple children of a File, but is notified of this according to the open-end argument.
3. Delete, which removes an entire tree of Files.
4. Move, which moves an entire tree between Modules.

In short, Files have write-once, read-many semantics.

*Modules* are the unit of replication in NFR. Every File is associated with a Module, and all Files in a tree are in the same module. Currently, the replication policy of a Module specifies what servers the Module is replicated to, and how to reach these servers. Concurrency control is separate from this—Locks are the tool provided for this purpose. Modules have write-once, read-many semantics like Files, and modifying a Module produces a new child Module with a new Name. If this causes different replication policies for the same File, NFR will make an “educated guess” to what the Users intentions are, perform some action to correct the replication policy, and notify the User of this. The rationale behind this is that on the short

---

<sup>7</sup>Represented by 32-character hexadecimal strings

term progress of policy enforcement is more important than enforcing the correct policy at all times when it comes to replication.

*Safes* are the unit of access control in NFR: a policy for access control for the Files and Modules “in” it. All Files and Modules in a particular tree are always associated with the same Safe, and a Safe is always in a Module. Safes can also contain other safes (including itself, which is default). Safes also have write-once, read-many semantics. If two children of a Safe have equal access control policies, NFR will simply merge the two Safes. If they differ it may mean that a File is in multiple Safes. The Users intentions are not clear in this case, so he will be notified and able to resolve the ambiguity.

*Users* are the unit of authority in NFR. Each File, Module and Safe is owned by the User that created it. Users have full authority over their objects, and may delegate authority over them to other users.

*Locks* are the basic building blocks of concurrency control in NFR. Conventionally, a lock refers to an object that can actually prevent access to data or a code block. In NFR, however, Locks do not provide any concurrency control by themselves; they are merely a tool NFR provides for implementing concurrency control policies by Users and their applications. A Lock is in effect a piece of information tied to a File. An application or user can ignore a Lock if he so chooses.

Replication policies in NFR are enforced on a best-effort basis; after a new File is stored in an NFR server, NFR starts to spread the file according to the replication policy of the Module of that File. NFR gives no guarantee to how many replicas of the file actually exists at any point in time. For Locks, NFR does give guarantees to the number of replicas that exist if a Lock is created: The User specifies the minimum number of replicas for a lock to exist. NFR will not create the lock until it can meet this requirement.

Currently NFR supports two kinds of Locks: the first requires the Lock to be present at a majority of the possible replicas. The second requires the Lock to be present at all possible replicas.

Note that NFR does not prevent a User from creating several Locks on a File, but setting the same Lock several times is not allowed.

We see that Locks can be used by Users and their applications to implement a concurrency control policy. We also see that a lock is information which the User may ignore at will, for example because he has extra-system information about whether it is safe to do so<sup>8</sup>. If a Locked file is written to, a “shadow child” File is created, and the owner of the Lock is notified of the existence of this child.

---

<sup>8</sup>Example: Bob has created a Lock on a File and taken a vacation. Alice, knowing this and needing to modify the file, decides to violate the Lock. Bob is notified of this when he gets home, and the Locked File and the new modified File can be merged/replaced.

## 2.6 Transaction protocols

We call an operation on replicated data a *Transaction*, which we define as a basic unit of consistent and reliable computing that accesses some resource, executed as an atomic action [11]. If a transaction is run successfully and its results made permanent, we say that the transaction has *committed*. If its execution is canceled or its results are not made permanent when it is finished, we say that it has *aborted*. Protocols for executing transactions are discussed here; the material in this section is largely from chapter 12 in [11].

A transaction is executed consistently and reliably if the execution has the following properties [9] (we say that it “passes the ACID test”):

1. *Atomicity*: The transaction is executed fully or not at all.
2. *Consistency*: The transaction takes the accessed data from one consistent state to another.
3. *Isolation*: No transaction may see (or modify) the state of a resource accessed by another transaction until the other transaction has finished accessing that resource.
4. *Durability*: Once a transaction commits, its modifications are permanent.

Protocols for maintaining these properties will be executed at two levels: In NFR, to distribute Locks in a manner that can give guarantees about the number of replicas of a File that has a Lock created on them, and by the User/application, who will use the Locks to implement concurrency control.

Durability is the responsibility of the storage mechanisms of NFR and the OS, and is beyond the scope of this report. Consistency and Isolation are maintained by concurrency control protocols, in the case of NFR in the domain of the User/application. We will discuss this in the next section.

Atomicity is maintained by three protocols:

- Commit protocols.
- Recovery protocols.
- Termination protocols.

Termination and recovery protocols are two opposite faces of the recovery problem: given a site failure, the recovery protocol deals with the procedure that the process at the failed site must go through to recover its state once the site is restarted, while the termination protocol addresses how the operational sites deal with the failure. Thus, the termination and recovery protocols on a system ensure that a transaction is run to completion (commit or abort) even if there are failures. If a site has no way to determine if

another site has failed, the termination protocol becomes unnecessary, and operational sites will just have to wait for failed sites to recover. This will be the case if the sites are connected by an asynchronous network.

The guarantee that transactions will run to completion is one requirement for the isolation property. The other requirement is that the concurrency control protocol does not let two (or more) concurrently executing transactions commit or act upon modifications to the same resource.

In NFR this is accomplished simply by locking items for one transaction at a time: functionality for doing this already exists in the NFR code.

A termination protocol is said to be *nonblocking* if it permits transactions to terminate at operational sites without waiting for recovery of failed sites. Otherwise it is said to be *blocking*. A recovery protocol is called *independent* if it can determine how to terminate a transaction that was executing at the time of a failure without consulting any other site<sup>9</sup>. Note that if a system does not have a termination protocol it will simply block until failed sites recover. This will be the case for our system. If no messages are lost and no channels fail, we will eventually recover because the delays associated with processing and message transfer are finite (we will state our assumption about synchrony and reliability in the requirements specification). Also, a User may terminate a transaction at any point in time, for example because he learns that a failed site will not recover soon enough to suit him.

Commit protocols, or more precisely *Atomic Commit Protocols* (ACP) maintain the atomicity of transactions. This means that even though the execution of the transaction involves multiple sites, some of which might fail, the effects of the transaction is all-or-nothing: Either all sites commit or all sites abort. This is called atomic commitment.

### 2.6.1 The Two-Phase Commit Protocol

The Two-Phase Commit (2PC) protocol is a simple and elegant ACP with two types of processes: A single *coordinator* that decides whether to reach a global commit or abort decision, and the *participants* that execute the transaction's resource accesses and vote whether to commit or abort. The commit decision is made according to the *global commit rule* [11]:

- If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.
- If all participants vote to commit the transaction, the coordinator has to reach a global commit decision.

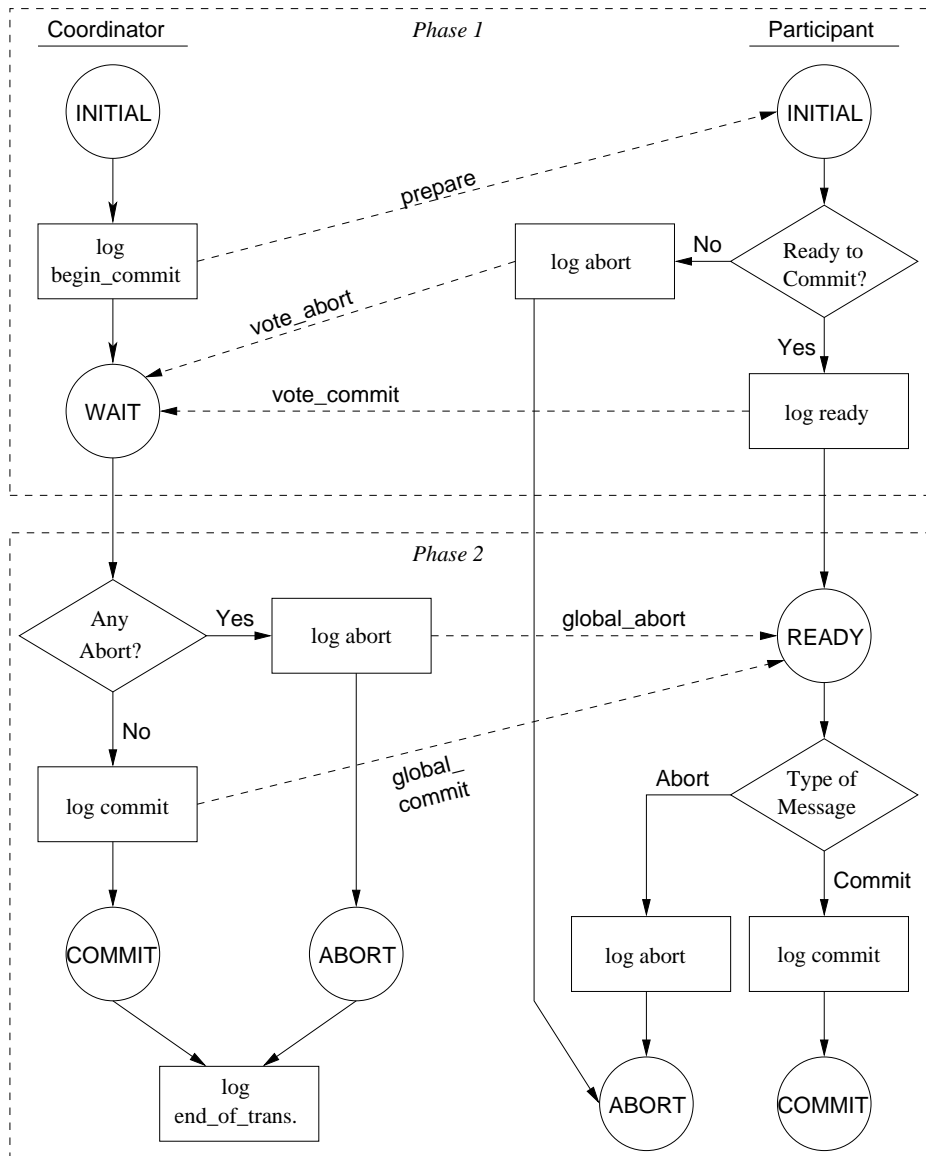
A description of 2PC that does not consider failures is provided in this section. Failure scenarios and a recovery protocol is described in sec-

---

<sup>9</sup>The existence of independent recovery protocols implies the existence of nonblocking termination protocols. The reverse implication is not true, however.

tion 2.6.2. Figure 2 should be helpful: The circles represent states, the whole lines represent state changes, the dashed lines represent messages (labeled with the message types), and the rectangles represent logging actions.

Figure 2: The Two-Phase Commit protocol



2PC has, not surprisingly, two phases:

*Phase 1:* A coordinator process is started (usually at the site where the transaction is initialized), writes a begin commit record in its log, sends a

`prepare` message to the participants, and enters the wait state. This message also contains a unique transaction id (TID), which is also in all further messages in this protocol run.

When a participant receives a `prepare` message, it checks if it can commit the transaction. If it can, the participant writes a ready record in its log, sends a `vote_commit` message to the coordinator, and enters the ready state. Otherwise, the participant decides to unilaterally abort<sup>10</sup> the transaction—it writes an abort record in the log and sends a `vote_abort` message to the coordinator. It enters the abort state and can forget about the transaction.

*Phase 2:* After the coordinator has received votes from all participants it decides whether to commit or abort according to the global commit rule, and writes this decision in the log. If the decision is to commit, it sends a `global_commit` message to all sites. Otherwise, it sends a `global_abort` message to all sites that voted to commit. Finally, it writes an end of transaction record in its log. The participants finish the transaction according to the decision and write the result in their logs [11].

To be accurate, the protocol described here is centralized 2PC with reliable message channels. There are many variations on 2PC that account for different network topologies, channel reliability and synchrony. These will not be discussed further here, as the described protocol is usable in our network model (see section 2.1) and strikes a balance between few messages and high concurrency.

## 2.6.2 Failure scenarios and a recovery protocol for 2PC

In this section we will describe all possible crash failures of the coordinator and participant processes in the protocol described in section 2.6.1 and how to recover from them, i.e. we will describe the recovery protocol, which is adapted from [11] to fit our asynchronous store-and-forward network. We still assume that messages are never lost (although they can take an arbitrarily long time to arrive at their destination). Note that if a process fails during message transmission, we regard the message as unsent, and if a process fails during logging, the last entry in the log is invalid and not used for restoring state. Figure 2 should still be helpful. First, the coordinator:

1. *The coordinator fails in the initial state:* The coordinator must simply be restarted and reinitialized.
2. *The coordinator fails after the begin commit record is written to the log but before the `prepare` command is sent:* On recovery, the coordinator knows from the log that the commit process is started, but it does not know whether the `prepare` command has been sent, so it must be (re)transmitted. Note that this may cause duplicate messages. We will

---

<sup>10</sup>Effectively a veto.

explain later how this protocol can be implemented in a manner that makes duplicate messages harmless.

3. *The coordinator fails while in the wait state:* If there are votes on the channel, it must have sent the `prepare` command, so it can reenter the wait state. If not, the coordinator does not know if it has sent the `prepare` command, so it is (re)transmitted.
4. *The coordinator fails after logging its final decision, but before informing the participants of the decision:* The decision has been recorded on the log, so it only needs to transmit it to the participants.
5. *The coordinator fails in the commit or abort states:* The decision is already logged and broadcast, so it only needs to write an end of transaction record in the log.

Then the failures and recovery for the participant:

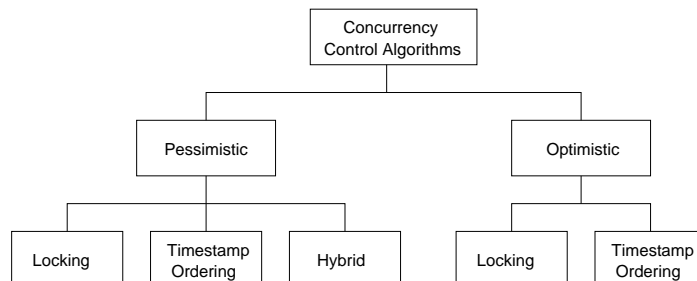
1. *The participant fails in the initial state:* The participant will automatically be restarted when the system is restarted, and it will find the `prepare` message on the channel.
2. *The participant fails after writing the abort record in the log but before sending `vote_abort`:* Simply retransmit the `vote_abort` message.
3. *The participant fails in the ready state:* If there is a `global_commit` or `global_abort` message on the channel, act on it. Otherwise, (re)transmit the `vote_commit` message.
4. *The participant fails while writing the data to be committed:* The participant still has locks on the items the transaction is executed on, so it can just rewrite the data (which will be available on the channel).
5. *The participant fails in the abort or commit states:* The transaction is complete, so nothing needs to be done.

We see that with an asynchronous reliable store-and-forward channel we can, given enough time, recover from any site failure without using a termination protocol.

## 2.7 Concurrency control mechanism

Concurrency control maintains the consistency and isolation properties of transactions. This is the responsibility of the User/application, so we will not give a detailed description of concurrency control here (but see chapter 11 in [11]). However, as NFR provides Locks as the tool for building concurrency

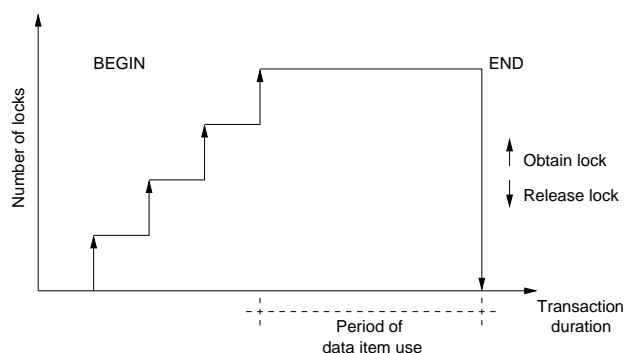
Figure 3: Classifications of concurrency control mechanisms.



control, we will provide a brief overview of *locking-based* concurrency control and show how it can be implemented on top of NFR.

The main categories of concurrency control mechanisms are shown in figure 3. *Pessimistic* algorithms synchronize the concurrent executions of transactions early in their execution life cycle, whereas *Optimistic* algorithms delay the synchronization for transactions until their termination.

Figure 4: Strict two-phase locking.



The main idea behind locking-based concurrency control is to ensure that the data shared by conflicting transactions is accessed by one operation at a time. This is accomplished by associating a lock with each data unit, and not allowing a transaction to access the data unit if a lock has been created on it by another transaction. One often-used locking scheme is (pessimistic) *strict two-phase locking* (2PL, figure 4). It is popular because it is relatively simple to implement and avoids *cascading aborts*<sup>11</sup>. Strict 2PL acquires locks before data items are accessed, holds all locks until all accesses are finished,

<sup>11</sup>Assume a transaction  $T_1$  modifies a data item, and a  $T_2$  later reads or modifies the same data item before  $T_1$  commits. Then, if  $T_1$  aborts,  $T_2$  must also abort (and so on, if another transaction had accessed any of the same data items as  $T_2$ ). This is called cascading abort.

and releases all locks at once when the transaction commits/aborts. An NFR application can implement strict 2PL like this: While the transaction is executed, ask NFR to create Locks on Files before they are accessed. When the transaction commits/aborts, ask NFR to remove all Locks.

To conclude, we list some possible applications and the locking schemes they are likely to use:

<b>Application</b>	<b>Likely locking scheme</b>
<i>Distributed Database</i>	Strict 2PL.
<i>Distributed Data-mart</i>	None, updates are made by infrequent, sequential writes.
<i>CVS</i>	Strict 2PL.
<i>Distributed Filesystem</i>	User-defined, depends on environment/usage pattern.
<i>Web Cache</i>	None, same as Distributed Data-mart.
<i>DNS</i>	None, Same as above.
<i>Distributed object persistency system</i>	User-defined, could be different for each class of objects.

## 2.8 Conclusion

We have looked at the theory behind a solution to the problem stated in section 1.2, focusing on describing the properties of an asynchronous network and a 2PC protocol for such a network. We have also looked at NFR and the open-end argument, and in the next section we will use this knowledge to define requirements to a system that solves the problem in section 1.2

### 3 Requirements specification

The requirements describe *what* a system should do (the required behavior of the system), while the design describes *how* the system should do it. Because the main function of the system is to execute a protocol, we will focus on behavioral description rather than data flow or data object descriptions.

We will first describe some non-functional requirements, provide an overview of the system, and finally describe in more detail the requirements to each subsystem. The subsystem requirements are numbered so we can match them to the solutions presented in the design section.

The phrase *must* in a requirement means that this requirement is absolute. *May* means that the requirement describes a feature that would be nice to have, but that is not essential.

#### 3.1 Non-functional requirements

Before we go into specifics we will state some overall goals and requirements to our infrastructure. Detailed requirements are covered in later subsections

*Function:* In short, the system will use an asynchronous store-and-forward channel to provide guaranteed distribution of Lock objects in NFR, by following an ACP.

*Performance:* The system, and indeed NFR itself, is a prototype or proof-of-concept implementation. The intention is of course to find out how useful a system implemented with the open-end-argument as design guideline will be in a distributed environment. Therefore rapid development and correctness are more important than performance (meaning speed of execution), and we will not attempt to optimize performance.

*Reliability:* We do not have the resources (time) to prove the validity of the system or to measure its reliability in any way. Instead we hope to induce reliability by working from a solid theoretical framework, creating an accurate requirements specification and a clean design, implementing in a suitable language on a stable platform, and comprehensive testing.

*Usability:* The system will not be used directly by users of NFR but will be more of an API to the rest of NFR. Thus, we focus on making the interface clean and simple (We will discuss a more precise statement of this later), rather than providing services to users. Nevertheless, in accordance with the open-end argument we will make the system flexible enough to accommodate the diverse demands users will make on NFR, and provide feedback on system state to users.

*Security:* Security is an important issue in an open, distributed system. It is also a large field far beyond the scope of our project. Instead of trying to fit a minor security model and policy into this project we will attempt to make later integration of a comprehensive security system simple.

*Infrastructure:* There are two categories of assumptions to the infrastructure our system will live in: First, what functionality does the infrastructure provide, and second, what are its failure modes. The former helps us fit our system into the existing system, and determine what modules are provided and what we must build ourselves. The latter helps us determine what behavior is necessary in our system to meet our reliability requirements. First, the existing functionality:

- We have access to libraries commonly provided by a Unix-like environment.
- We have access to libraries that can perform common network related tasks, like talking to SMTP and POP3 servers.
- We have access to libraries that can perform data processing tasks like base64 encoding.
- The NFR modules described in section 2.5 are available.

This is the “base” our system is built on in figure 5. Then the failure models:

- The communications subsystem, the protocol subsystem, and the NFR interface must only fail by crashing.
- For the asynchronous 2PC protocol, the network is assumed to be reliable, i.e. once a message is sent on the channel the network may not fail.

These assumptions must hold if the protocol described in sections 2.6.1 and 2.6.2 is to execute reliably. Note that other Lock distribution protocols may require that other assumptions hold. We will attempt to make integration of other protocols simple.

### 3.2 Overall system description

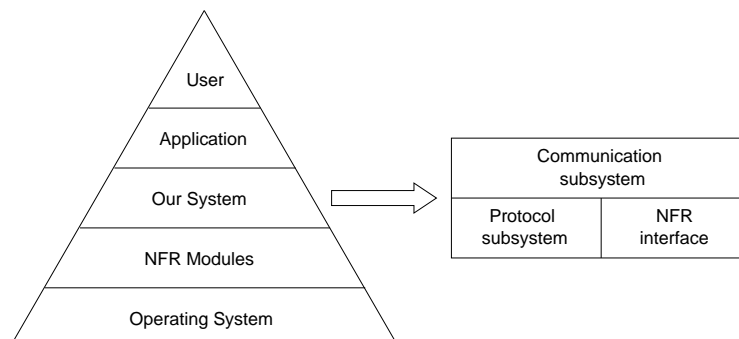
As shown in figure 5 our system fits roughly between the applications and the existing modules in NFR, i.e. those described in section 2.5. The applications interact with our communication subsystem, which sends commands to other NFR servers and to the NFR interface and protocol subsystem. These two subsystems modify the state of the local NFR through the NFR modules APIs.

However, the users may bypass the application layer and communicate directly with the communication subsystem<sup>12</sup>, or applications may call methods in the local NFR modules directly, bypassing our system.

---

<sup>12</sup>Compare with using a web browser versus talking directly to a web server using telnet.

Figure 5: Overview of the system and its environment.



### 3.3 Communication subsystem

The communications subsystem is responsible for parsing and generating commands, and sending and receiving them over a network and between the network, the NFR interface and the protocol subsystem. To do that an unambiguous command set (“The NFR command set”) must be defined:

- R 3.3.1** The system must have a command set unambiguously describing the content and meaning of commands sent over the network.
- R 3.3.2** The system must be able to parse and generate Lock distribution command messages conforming to this command set.
- R 3.3.3** The system may be able to respond to commands that are not consistent with the command set with an error reply.

A message containing a command from the NFR command set is called a “command” for the rest of this section. The requirements to the communication subsystem itself are:

- R 3.3.4** The system must be able to send and receive commands over email.
- R 3.3.5** The system may be able to send and receive commands over other transport mechanisms.

Note that we use the word “transport” mechanism to mean both transport layer and application layer protocols. Also note that we do not include the ability to handle malformed messages in the list of requirements.

### 3.4 NFR interface

The requirements to the NFR interface are simple; much of the required functionality exists in the NFR modules API described in section 2.5. What we need is a way to call the correct methods in NFR or the protocol subsystem based on the contents of commands.

**R 3.4.1** The system must be able to call methods in NFR based on the contents of a command.

**R 3.4.2** The system must be able to respond to a command to its originator, via the communications subsystem, possibly with an error message.

### 3.5 Protocol subsystem

The protocol subsystem is the central part of our system. Its task is to execute lock distribution protocols correctly, in response to commands received from the communication subsystem. The protocol subsystem must be able to execute any Lock distribution protocol that we care to implement.

**R 3.5.1** The system must be able to execute a Lock distribution protocol, including storing and deleting Lock objects in NFR.

**R 3.5.2** The system must be able to execute a recovery of any Lock distribution protocol it is able to execute.

**R 3.5.3** The system must be able to execute several Lock distribution protocols concurrently.

**R 3.5.4** The system must be able too keep stable transaction logs.

**R 3.5.5** It must be possible to integrate other Lock distribution protocols into the system in a later project.

The concurrency requirement is a necessity, because, for the asynchronous 2PC protocol, the system may need to execute both a coordinator and a participant at the same site. It is a convenience, because simultaneous execution of several protocol will allow higher throughput.

### 3.6 Conclusion

We have described the requirements to a solution to the problem statement in section 1.2. We have stated the non-functional requirements to our system and the infrastructure, and specific requirements to a command set, a communication subsystem, a NFR interface and a protocol subsystem.

## 4 Design

In this chapter we present and discuss various design choices based on the requirements in section 3.

The design goal of a “clean design with a simple interface” is better stated as a design with *high cohesion* and *low coupling* [12]. Cohesion is a measure of the functional strength of a module: If a module performs tasks that are related in some way it exhibits some form of cohesion. Stated simply, a module has high cohesion if it does just one thing. Coupling is a measure of the interdependence among modules, or the interface complexity between modules.

In addition to being useful for modeling the real world, object orientation can help structuring a design for high cohesion and low coupling. For this reason, and because the implementation will be in an object oriented environment we will use an object oriented design: Each subsystem will be split into classes that encapsulate the required behavior and information, and classes will be gathered in modules. Generally, a module contains a single “parent” class and all classes that inherit the parent. The content of modules will be discussed in detail in the implementation section.

The non-functional requirements will not be discussed explicitly, but will guide the subsystem design. The design will focus on the subsystems described in the requirements section. We will indicate what requirements are fulfilled by each subsystem design. We start with the protocol subsystem.

### 4.1 Protocol subsystem

We will discuss the design of the protocol subsystem first because the other parts of the system exist (for now) mainly to support it.

Protocols are driven by messages: changes in a protocol state machine happen only when it is initialized, and later when it receives a message. If a protocol (or the entire system) crashes, the protocol may need to be recovered from the contents of a log file. The protocol module should be responsible for this, using the communication subsystem as necessary (for example, if an instance of the asynchronous 2PC protocol is recovering, it may need to read “old” protocol messages from the POP3 inbox.)

Therefore it makes sense to define an interface to a protocol class with three methods:

1. An initialization method, which sets up the state of the required protocol machinery.
2. An advance method, which takes a protocol command message as argument, and advances the protocols state according to the command, possibly sending out further commands (via the communication subsystem) as a consequence.

3. A recover method, which is responsible for recovering the pre-crash state of a failed protocol execution.

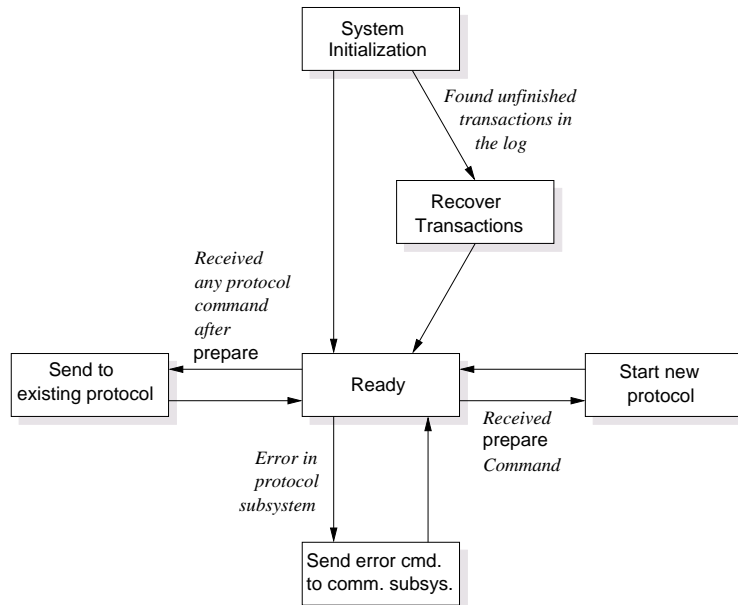
Most ACPs should fit in this interface. A class that performs these functions fulfills requirement R 3.5.5.

If we are to let several protocols execute concurrently, we must allow several coexisting protocol machineries with independent state. This can be implemented as a Protocol class with an object (instance) for each protocol execution. Because Protocol objects do not initiate action by themselves (but rather when their advance methods are called), we need not run them as separate threads or processes. The part of the system that calls the advance method might be run as separate threads or processes, however. This fulfills requirement 3.5.3.

Applying the interface discussed above to a Protocol object gives a design with a small interface that is easily implemented by different protocols.

Figure 6 shows a state transition diagram of the actions of the protocol subsystem.

Figure 6: STD for the protocol subsystem.



Unsurprisingly, the protocol we will implement is the asynchronous 2PC protocol. This is the answer to requirement 3.5.1. Other protocols, like synchronous three-phase commit<sup>13</sup>, can be added in later projects.

<sup>13</sup>A fault-tolerant, non-blocking ACP, see for example [5].

Although the implementation of the recover method is the responsibility of each protocol implementation, we believe it always makes sense to place log files in NFR (as Files), because NFR already implements a stable storage with a consistent interface on whatever platform we are running on. We will therefore assume that the log is in NFR for all Protocols that require logs. Each NFR server should have its own log-file Module. The Protocol object will then need to access NFR to manage Lock objects, manage logs and retrieve meta-data information about Files and Modules, for example to list the replicas of a File. This fulfills requirements 3.5.2 and 3.5.4

The remaining issue is error handling. As mentioned in section 3.1 we assume that the only failure for the system is crashing. This will be handled transparently by the protocol subsystem upon recovery, and we do not need to inform remote users about the crash. If the state of NFR prohibits our protocol from committing (i.e. if a File is already Locked), the protocol will simply abort. If there are errors in NFR itself, the owner of the server will be interested to know, and we should print an error message to the terminal or a local log. We should also abort the protocol if possible, or if not, send an error report command to the initiating user<sup>14</sup>. The transmission of error commands must be postponed to a later project, because we will not implement the report command.

## 4.2 Communication subsystem

The system described in this section is designed to fulfill requirement 3.3.4, giving the system the ability to send and receive commands from the NFR command set over email.

Most transport protocols have similar interfaces: Connect, disconnect, send and receive, so it makes sense to define a Server class that has these methods, and that is implemented by Servers for different transports. Objects of this class should also be run as separate thread, for the following reason: We wish to use blocking IO, because we think it is simpler to implement than asynchronous IO. It makes debugging more difficult, though, because a multi-threaded program's state usually differs from execution to execution<sup>15</sup>. We also wish to be able to execute multiple Protocols concurrently. Therefore we will need to be able to receive messages from several transports concurrently, and if we use blocking IO this can only be accomplished with separate threads for each receiving object.

As we need to use email as transport, we will implement a mail Server. To simplify the implementation, this server will rely on existing SMTP and POP3 servers, and act more as an interface to these. In addition to the methods listed above, the mail Server will need a remove method to allow

---

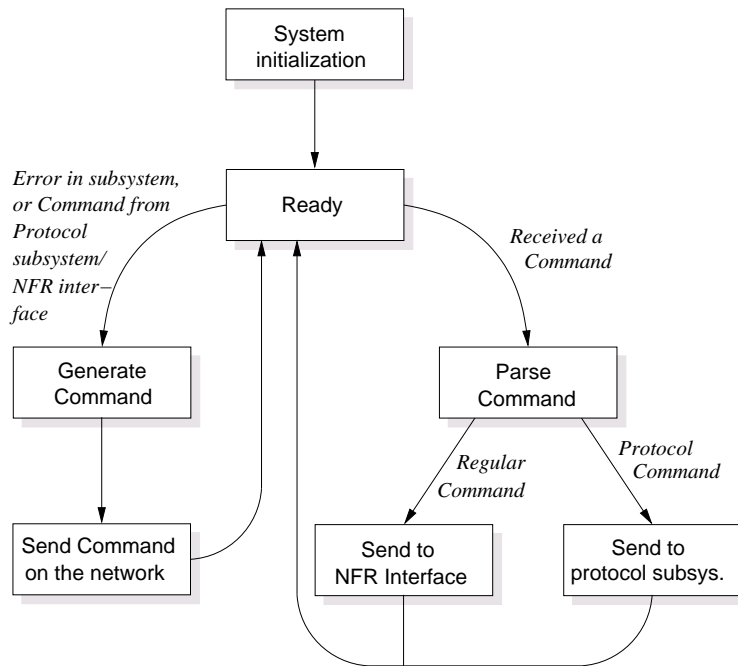
<sup>14</sup>For example, the 2PC protocol may not be aborted by a participant after the vote has been sent.

<sup>15</sup>Due to scheduling of other tasks by the OS, etc.

the protocol subsystem to control when messages are deleted from the POP3 inbox. This will allow the asynchronous 2PC protocol's recovery protocol to operate as specified in section 2.6.2.

If it is not the mail Servers task to remove messages from the channel, it must keep track of what messages it has read so it will not retrieve the same message twice. If the system crashes, this information will be lost, and the same command may be executed several times. For our 2PC protocol, multiple messages are, except for degrading performance by using more bandwidth, harmless. We will explain how this accomplished in the implementation section. This is also true for the commands described in the next section. It may not be true for other protocols and commands added in the future, however. The solution may be to write the mail Servers state in stable storage every time it retrieves a message. This will not be implemented in our project.

Figure 7: STD for the communication subsystem.



The STD in figure 7 shows the states and actions of a single Server in the communications subsystem.

We also need some way to send NFR commands and NFR item data between subsystems. The Command objects are of a class designed to encapsulate data from NFR items and commands from the NFR command set. Command objects should be responsible for parsing messages received over the network, and generating messages to be sent over the network. When a

Server receives data, it should generate a Command object with this data (parsed to the Command object's internal representation), and send it to the NFR Interface or protocol subsystem. They are mostly meant to be a programming convenience.

Crashes in the communication subsystem effectively prevents the NFR server with a failure from informing other servers about its state. Thus, errors in the communication subsystem should only be recorded locally. Errors in the parsing of received NFR commands should be handled by replying to the sender with an error message. While this is important, we will only implement the lock command, and not the command intended for sending error messages (the report command).

#### 4.2.1 The NFR command set

The Command objects effectively encapsulate commands from the NFR command set. This command set is used to exchange requests and information between NFR servers (as opposed to between applications and servers). NFR commands are embedded in messages and sent over some transport protocol, like TCP/IP, SMTP or NNTP, and a reply may be send by the recipient of the original message, and so on. This message exchange is asynchronous in nature, and message delivery guarantees are only as good as those of the transport protocol used.

Although the most interesting part of the command set for this report is the part related to Lock distribution (which is the only part implemented in this project), we give an overview of the entire command set here for reference. This command set is liable to change, and we try to design the system so changes are easy to incorporate.

*<name>* is the globally unique NFR name of either a single file, safe or module.

*<meta\_data>* is the meta data of a safe/module/file, including creator, replication policy, and so on. It will vary for each object type and individual object.

Fist we give an overview of the general command set:

- fetch *<name>* *<meta\_data>*: Ask a server to fetch the named item. The result is either a report command (if there was an error) or a store command with the requested item. If the item was a module, the result is a string of store commands with all files in the module.
- store *<name>* *<meta\_data>* *<data>*: Request that the server store the named item. The result is a report command with either an error or an "ok" message with the new name of the stored item.
- stat *<name>*: Request meta-data for *<name>*. This is returned in a report command.

- `destroy <name> <meta_data>`: Request the deletion of them named item. The result is a report command.
- `report <previous command> <result>`: Reports the result of a previous command., possibly an error message.

Then, a precise description of the Lock distribution commands.

- `lock <protocol> <tid> <File>`: This is the **prepare** command for initiating a lock distribution protocol. For initiating our 2PC protocol the string `"simple_asyn_2pc"` is used for `<protocol>`. `<tid>` is a unique transaction ID (in effect a Name generated by NFR) which is generated by the protocol coordinator. If a site receives a lock command with a `<tid>` null Name, that site should start the coordinator. If the protocol commits, a Lock object is created for `<File>` at the participating sites.
- `lock <message> <tid>`: A protocol message other than **prepare** in the given protocol. For our protocol, `<message>` will be one of the strings `{"vote_abort", "vote_commit", "global_abort", "global_commit"}` corresponding to the messages described in section 2.6.1. The `<tid>` will link this command to the correct protocol run.

This command set, together with the Command object described in the previous section fulfills requirements 3.3.1 and 3.3.2.

### 4.3 NFR Interface

The exchange of NFR command set messages is asynchronous, so there is no reason for making Server objects wait for the NFR interface to process a Command before returning to its receive state. Thus, it makes sense to allow Server objects to insert Command objects in a working queue in the NFR interface subsystem, and let the subsystem process the Commands one at the time.

NFR commands will always affect NFR Modules or items inside a Module, so we should have one NFR interface object for each NFR Module. To improve throughput and allow processing of Commands on the queue independently of the Server objects actions, this object should also be run as a thread. Server and NFR interface objects will then run independently, with only two methods in the interface between them: `insert`, in the NFR subsystem, which a Server calls to insert a Command object in a NFR interface object, and `send`, in the Server objects, which NFR Interface objects call to send Command objects on the network. We will call this object a Port.

One problem with running Ports as separate threads is that a large repository may have a large amount of Modules, the system may be unable to handle such a large amount of threads.

While parsing and generating data conforming to some transport protocol is the task of the Command object, the Port object must be able to interpret the contents of Command objects into actual commands from the NFR command set, and execute these commands, either calling some method in the NFR API or in the protocol subsystem. Lock distribution protocols only operate on a single Module (they lock Files, which are always in a Module). So we should let Ports administrate all Protocols that operate on Files in the Ports Modules. Also, this makes the interface between the communication subsystem and the rest of the system simple, because all a Server needs to do with a Command object is to send it or insert it into a Ports working queue.

The Port object implements requirements 3.4.1 and 3.4.2, except only for lock commands (which means that errors messages are not implemented).

#### **4.4 Conclusion**

We have described the design of subsystems which fulfills the absolute requirements described in section 3, and the reasoning behind these design choices. The unfulfilled requirements will be discussed in the Further work section. The next section describes our implementation of the design.

## 5 Implementation

In this section we will describe how we have implemented the design choices from the previous section.

We will use the following naming convention: Module and attribute names are written in lower case. Class names have capital first letters in each word that make up the name (e.g. MailServer). In method names, all words after the first have capitalized first letters (e.g. addPort()). Constants are capitalized (e.g. PROTO\_SMTP) Filenames, Input and output to programs are in a typewriter font (e.g `server.py`).

Figure 8: OO Diagram conventions.

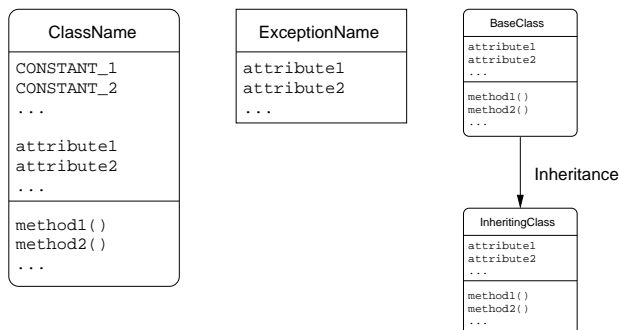


Figure 8 shows an example of the diagram style we will use. Classes are rounded squares with the class name on in the top, constants and attributes in the middle, and methods in the bottom. Usually, only important methods and attributes or those that are meant to be accessed from outside the class (in this projects) are drawn. An arrow denotes inheritance (the class pointed to inherits the class pointed from). Exceptions are represented in a similar manner, except that their squares are not rounded. Note that exceptions are just classes that inherits the Python built-in class Exception.

### 5.1 Environment

The system is implemented and tested on Mandrake Linux 7.1, kernel version 2.2.16. The programming language used is Python, version 2.0 final.

Python is an interpreted, interactive, object-oriented programming language. Its high-level syntax and extensive library makes it ideal for quickly constructing a prototype system like ours. Extensive documentation can be found at [2].

We will just note a few things that will clarify the description of the implementation: Method and attributes that are prefixed with double underscores (example: `__POP3connect()`) are considered “private” to their containing class, and methods with both leading and trailing underscores are

“special” methods that implement operator overloading and other functionality for classes (example: `__init__()` is a class constructor) The `None` value is Python’s “null” or empty value.

### 5.1.1 Existing NFR modules

We have built our system on several existing NFR modules. The ones we have explicitly imported/inherited in our modules are listed here:

**cache:** Provides functionality for searching for items in NFR, using a name cache.

**exception:** Contains `ExNFR`, the “standard” NFR exception.

**file:** Contains the implementation of the File object.

**lock:** Contains the implementation of the Lock object.

**module:** Contains the implementation of the Module object.

**name:** The Name object. All objects in NFR have their names generated by `name.Name`.

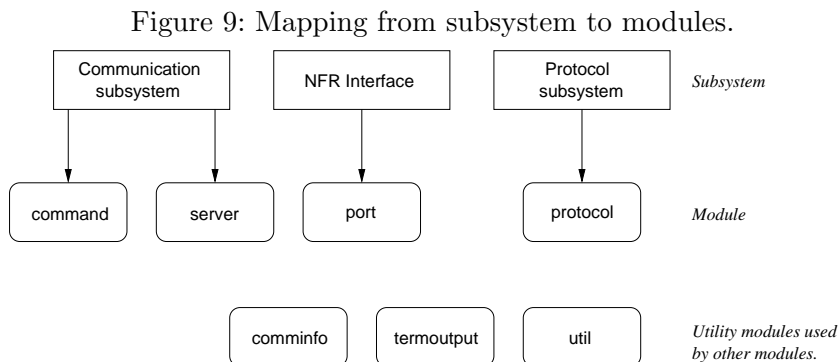
**nfr:** Initializes NFR, and provides the NFR constants.

**no:** Contains the `NO` class, which is the base class for all named objects.

The file names are the same as the module names with “.py” appended.

## 5.2 Structure

Figure 9 Shows a rough mapping from the designed subsystems to the implemented modules.



Before we describe each module, we will give a brief account of the behavior of the entire system.

On system startup, one Server thread for each transport and one Port thread for each Module in NFR is started. The Ports then scan the log Files (in NFR) for unfinished transactions, and attempt to recover any failed transaction by initiating new Protocol objects for all unterminated log Files.

When the Server threads receive data, it is parsed and converted to a Command objects, which are sent to the Port objects serving the destination NFR Modules. The Ports then take action depending on the contents of the Command objects:

- Either imitating new Protocol objects (and sending the Command objects to them), or
- Sending the Command objects to existing Protocol object.

Either way, new Command objects may be generated by the receiving Protocols, and sent back on the network through the Servers.

### 5.2.1 Full class hierarchy

Figure 10 shows the a complete class hierarchy of the modules we have implemented.

Some of our classes also extend `exception.ExNFR` or `threading.Thread`, which are external to our system. These are not shown in the figure. Also, some modules contain methods, variables and constants not shown in the figure.

We will now proceed to give a detailed description of each module.

## 5.3 CommInfo module

File: `comminfo.py`

The `comminfo`, `util` and `termoutput` modules are helper modules that encapsulate functionality and data needed by the protocol, server, command and port modules.

The `comminfo` module only contains the class `CommInfo`. It is purely a programming convenience: It contains often used fields such as transport, servers, etc., initialized to appropriate empty values. When a class inherits `CommInfo`, it also inherits these fields. This helps eliminate typing errors<sup>16</sup> and increase code readability by the using consistently named attributes.

---

<sup>16</sup>In Python, attributes can be assigned to objects dynamically. This can be a source of errors if we do not take care to use the correct attribute names.

Figure 10: Full class/module hierarchy

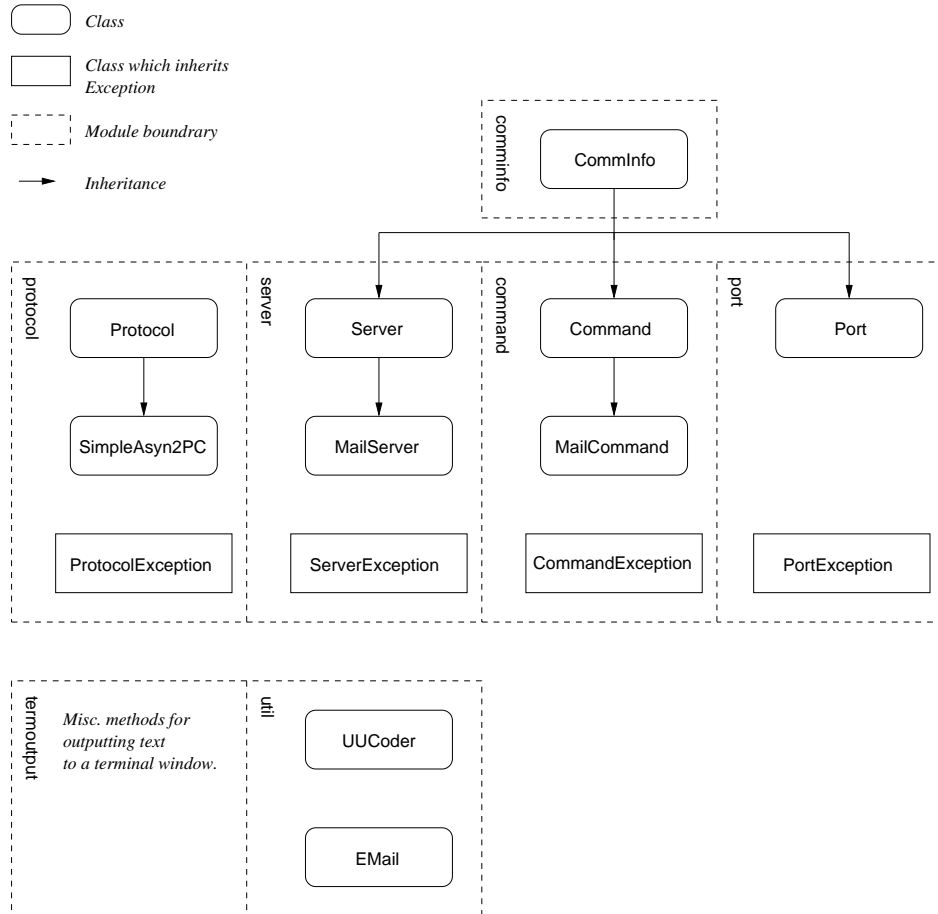
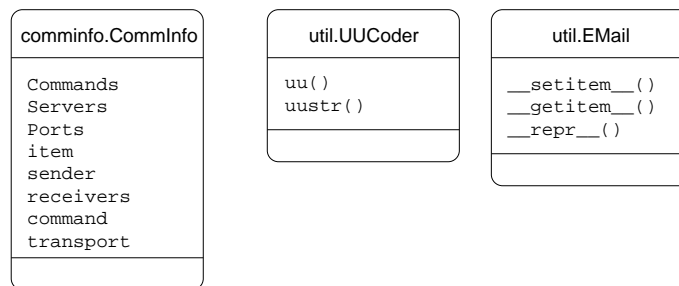


Figure 11: The comminfo and Util modules.



## 5.4 Util module

File: `util.py`

Two helper classes, `UUCoder` and `EMail`, are contained in the `util` module (see figure 11). The `UUCoder` has methods for converting between binary data strings and lists of base64 encoded strings. It can be used to convert binary data before/after transmission through an ASCII transport such as SMTP. It uses the Python standard library module `binascii` to accomplish the actual en/decoding.

The `EMail` class represents an email as a Python dictionary<sup>17</sup> by overloading the dictionary indexing operators. It can either parse a list of strings as retrieved from a POP3 server with the Python `poplib` library, or parse an existing dictionary. It handles folded lines and the separation of the header/envelope and the message text.

It is used by the `MailServer` for convenient lookup of header fields, by the `MailCommand` for parsing and generating email data, and by the `SimpleAsyn2PC` Protocol class to parse Commands found in the transaction log Files.

## 5.5 Termoutput module

File: `termoutput.py`

This is a small helper module with customizable output to a terminal (see figure 11). It uses ANSI escape codes as “markup” to change the attributes (color, boldness, etc.) of displayed text. It has four output functions:

0. `error()`, for fatal error messages.
1. `warning()`, for warning about suspicious conditions.
2. `info()`, for outputting standard information.
3. `debug()`, for debugging information.

The module has a “global” field called `level` that controls the amount of output: The higher the level, the more output (corresponding to the numbers in the list above). Error messages are always printed.

## 5.6 Command module

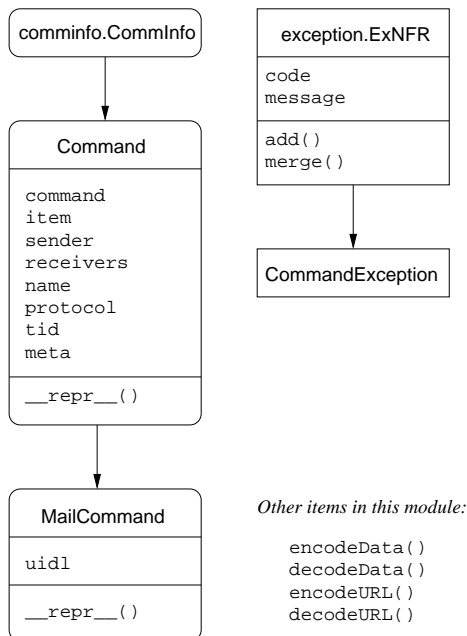
File: `command.py`

The `command` module is responsible for parsing and generating commands conforming to the NFR command set described in section 4.2.1.

---

<sup>17</sup>Python datatype that acts as an associative array, which can be indexed by any datatype.

Figure 12: Classes in the command module.



The module has three classes:

Command is the base class for all commands. It has no methods except the constructor. Its attributes correspond to the fields in the NFR command set, plus the URLs and Modules of the sender and receivers of the command. A class that inherits command should be defined for each transport.

CommandException is a simple Exception that inherits exception.ExNFR.

We have implemented MailCommand, a Command class suitable for SMTP/POP3. It uses the same attributes as the Command class, plus an uidl attribute that uniquely references an item in a POP3 inbox [16], and a date attribute that indicates when a MailCommand object was created.

Its constructors (`__init__()`) behavior depends on its first argument: If the argument is a `util.Email` object, the `Email` objects relevant fields are retrieved and stored in the new `MailCommand` object (see section 5.4). If not, the first argument is interpreted as a `Command` object, which data is inherited by the new `MailCommand` object.

The `MailCommand`'s `__repr__()` method uses `util.Email` to generate a string representation of a `MailCommand` object which in effect is an email message. When a `MailServer` sends a `MailCommand` object to a SMTP server it simply uses Python's `smtplib` to send the string version of the `MailCommand`.

```
ms.sendmail(sender, receivers, repr(cmd))
```

Where `ms` is the SMTP server object and `cmd` is the MailCommand.

Figure 13 shows an example of data generated by the `__repr__()` method.

Figure 13: Example MailCommand rendered as a string.

```
X-UIDL: e196a9563f053c3cd36053595338998b
Date: Sun Dec  3 17:42:18 2000
From: andrer@server.pasta.cs.uit.no
To: nfr0@server.pasta.cs.uit.no.
Subject: NFR Command f0ba[...]f398 aa01[...]5771

lock simple_asyn_2pc 0000[...]0000 63a1[...]0b18
```

We have also created four helper methods to parse and generate Commands:

1. `encodeData()`, which takes arguments corresponding to the fields in a command from the NFR command set (see section 4.2.1), and generates a string, and:
2. `decodeData()`, which takes a string previously encoded by `encodeData()` and decodes it into a dictionary containing keys from the NFR command set and their corresponding values from the string.

Currently, only coding of the lock command is implemented. The other two methods are discussed in the next section.

### 5.6.1 Uniform Resource Locators

NFR uses URLs as defined in [19] to identify replicas and describe how to they are reached. We will therefore mention a few relevant facts about URLs here:

In general, URLs are on the form

```
scheme://user:password@host:port/path
```

Scheme and host are required fields, and, if password is specified, user must also be specified. Of course, not all fields are valid for all schemes (e.g. path is meaningless if the scheme is SMTP).

URLs identifying replicas are stored in Module meta-data in NFR. URLs identifying replicas may look like this:

```
frtp://nfr.pasta.cs.uit.no:12445
smtp://nfr0@server.pasta.cs.uit.no
```

They tell us that the first replica is available over FRTP<sup>18</sup> at port 12445, and that the other is available over SMTP through the email address nfr0@server.pasta.cs.uit.no. A replica can usually be reached over several transport mechanisms.

The other two stand-alone methods in the command module are implemented to handle URLs:

3. `encodeURL()`, which takes a string on the form listed above, parses it (using a regular expression), and returns a dictionary with all the fields described above (some of which may have the value `None`), or throws a `CommandException` if the string is not a valid URL.
4. `decodeURL()`, which takes a dictionary as returned by `encodeURL()`, and generates and returns a URL string. It will also throw a `CommandException` if the dictionary defines an invalid URL.

In effect, `MailCommand` using `en/decodeURL()` and `util.Email` parses and generates transport headers, while `en/decodeData()` parses and generates NFR commands.

## 5.7 Server module

File: `server.py`

The server module is responsible for sending `Command` objects between NFR servers and `Ports`. Together with the `Command` module it implements the communication subsystem.

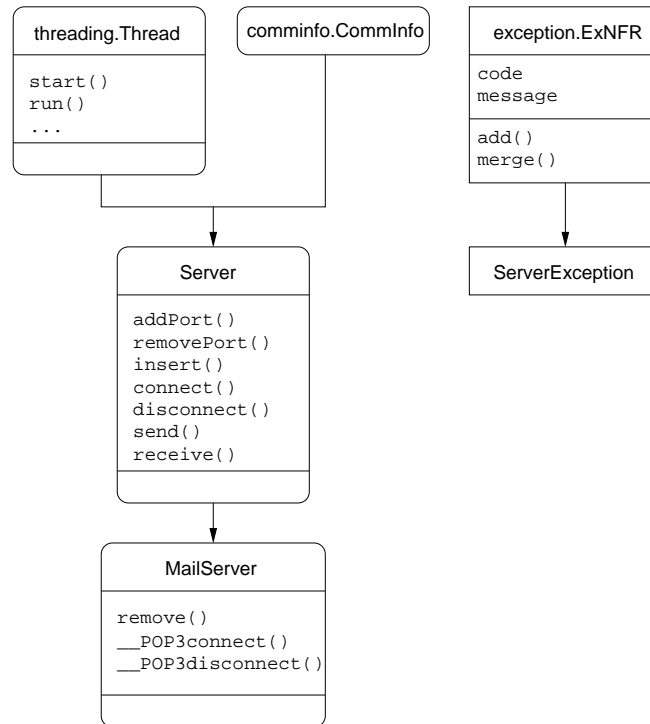
The `Server` class is the base class for servers for different transports. All Servers should implement the following methods defined in the `Server` class:

1. `connect()`: Set up a connection to the transport this `Server` is designed for.
2. `disconnect()`: Disconnect from the transport this `Server` is connected to.
3. `receive()`: Block until a command is received, generate a `Command` object from it, and `insert()` it into the receiving `Ports` queue. Do this until `disconnect()` is called.
4. `send()`: Send a `Command` object over the transport this `Server` is designed for.

---

<sup>18</sup>A protocol for sending NFR commands. It runs on top of TCP/IP

Figure 14: The server module



Methods 1 and 3 are invoked when the system is initialized, and method 2 is invoked when the system is shut down. The `send()` method is invoked by Port objects to send Command objects of the appropriate type over the transport that the Server uses. For example, `MailServer.send()` sends MailCommand objects. It is expected that a Command object of the class intended for a Server is able to render itself into a data format accepted for transmission over the Servers transport mechanism. Our MailCommand does this, rendering itself into a string complying with the RFC 822 message format (see figure 13).

As Ports run as independent threads, and several Ports may use a single Server, the Server must synchronize access to its transport mechanism. This is accomplished by forcing the sending thread to acquire a lock before it is allowed to send. We use the Python standard library `threading.Lock()` class for this kind of synchronization.

In addition, the Server base class implements three methods to manage Ports:

1. `addPort()`, which adds a Port object to the Servers list of Ports. Called once for every Port during system initialization, when all Modules in

NFR are given their own Port, and later to add Ports when Modules are created.

2. `removePort()`, called by the system to remove Ports from the servers Port list when Modules are deleted.
3. `insert()`, a convenience method used by Servers to insert a newly received Command into the receiving Ports working queue.

Servers inherit `threading.Thread` from the Python library. Two important methods in the Thread class are `start()` and `run()`. The `start()` method is called to initialize the underlying thread machinery. It calls the `run()` method, which is executed as a separate thread. Thus, classes that inherit Thread should implement/call their functionality from the `run()` method.

Servers are intended to execute their `receive()` method from the `run()` method. This is done in the MailServer class.

### 5.7.1 The MailServer class

The MailServer class is our implementation of the Server. As described in section 4.2, it acts as an interface to an existing POP3 and SMTP server, sending and receiving MailCommand objects. It uses the Python standard library modules `poplib` and `smtplib`, which provide convenient methods for accessing external POP3 and SMTP servers.

The MailServer implements the four communication methods described in the previous section (in addition to inheriting the Port management methods). It has an additional `remove()` method used to remove messages from the inbox. We will discuss the usage of this method later. It also has two internal methods used to manage connection with its POP3 server (`__POP3connect()` and `__POP3disconnect()`).

The list of arguments to the MailServers constructor is quite long, including user name and password (or shared secret, if APOP<sup>19</sup> is used, host names and ports for the SMTP and POP3 servers, and a check interval (in seconds).

It does not make sense to continuously poll the POP3 server for new emails, since new messages will be prevented from arriving in the mailbox while we are connected. The check interval determines the amount of time between each login to the POP3 server.

The pseudocode for the `receive()` method is described below.

---

```
while 1:
```

---

<sup>19</sup>A method for logging into (some) POP3 servers. It uses a hash of a string given by the server and a shared secret instead of a password.

```

acquire the receive lock

if i am not connected to the POP3 server:
    call __POP3connect()

make a uidl list of (message number, uidl) tuples
for the messages in the inbox

for uidl in uidls:
    if uidl is not in the read_messages list:
        retrieve the message
        generate an util.Email object from the message
        retrieve the "Subject:" line
        if the "Subject:" line indicates a NFR Command:
            generate a MailCommand from the EMail object
            insert the MailCommand in a Ports queue
        append the uidl to the read_messages list

call __POP3disconnect()

release the receive lock
sleep(check interval)

```

---

The `remove()` method is used by SimpleAsyn2PC Protocols to remove MailCommands from a MailServers inbox. The parameter is an uidl, and, like `receive()` it generates a list of (uidl, message number) tuples from the POP3 mailbox, and deletes the message with the message number corresponding to the uidl parameter. Like the `send()` method, it is synchronized by a `threading.Lock` primitive.

## 5.8 Port module

File: `port.py`

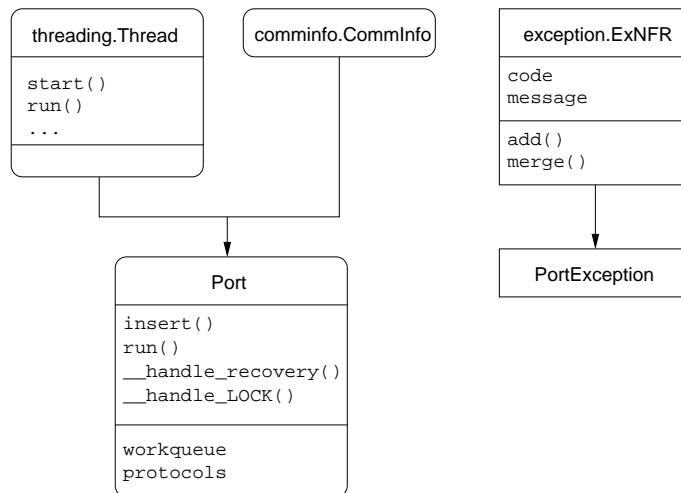
A Port is the active interface to a NFR Module, and implements the NFR interface subsystem.

When the system is started, all Modules in NFR should be found (using the NFR API), and a list of Port objects, one for each Module, is generated and sent to the Server objects. Thus, every Server object has a list of Ports, one for each Module.

Command objects contain the name of the Module they are addressed to. As Commands arrive at the Servers, they are inserted into the working queue of the Port belonging to the addressed Module.

The working queue is an instance of a synchronized queue primitive provided by the Python library (`Queue.Queue`) It supports a fixed-length

Figure 15: The port module



circular buffer of objects, and (optionally) blocks threads that attempt to insert objects when the buffer is full or remove items when the buffer is empty<sup>20</sup>. It enables Servers to send Commands to Ports independent of the Ports actual processing of Commands.

### 5.8.1 Recovery

Ports inherit `threading.Thread`, and implements the `run()` method which is started when the `start()` method is called.

The first thing the `run()` method does is to start the `__handle_recovery()` method. This method scans the Ports Module for log Files. These are identified by the `protocol.LOG_FILE_HEADER` constant and the name of a protocol. If any are found, new Protocols of the type specified in the log Files are instantiated, and their `recover()` methods are called. If a Protocols `recover()` method returns true, it is inserted into the Ports list of running transactions.

Each Protocol subclass must implement their own recovery scheme. Our asynchronous 2PC protocols recovery is described in section 2.6.2. Its implementation will be considered later.

### 5.8.2 Command handling

After the `__handle_recovery()` method is finished, the `run()` method enters a loop retrieving Command objects from the work queue. After a Command object is retrieved, the Port invokes some internal method depending on the command string in the object. This is the first string in the commands

<sup>20</sup>It uses semaphores to implement this.

described in section 4.2.1. Currently, only the lock command is implemented, so when the Port retrieves a lock Command, it is sent to its `__handle_LOCK()` method, which pseudocode is as follows:

---

```
if command string in the protocol.protocols dictionary:
    find the class of this Protocol
    instantiate a Protocol object of this class
    set the Protocol objects TID to the TID in the Command
    add the Protocol to the transaction list.

    if the new Protocols TID == NFR_ZERONAME:
        set the Protocols iscoordinator field to true

    call the new Protocols advance()
else:
    for existing_protocol in the transaction_list:
        if existing_protocol.TID == new Protocols TID and
           existing_protocols module ==
           the Commands destination module:
            call existing_protocol.advance(Command)
            break
    else:
        discard the Command, and print a warning.
```

---

If no existing Protocol instance is found for the new Command, and it is not an initiation/prepare command, it is assumed to be a duplicate Command for an old transaction, and is simply discarded. We will discuss duplicate Commands further later in this section.

## 5.9 Protocol module

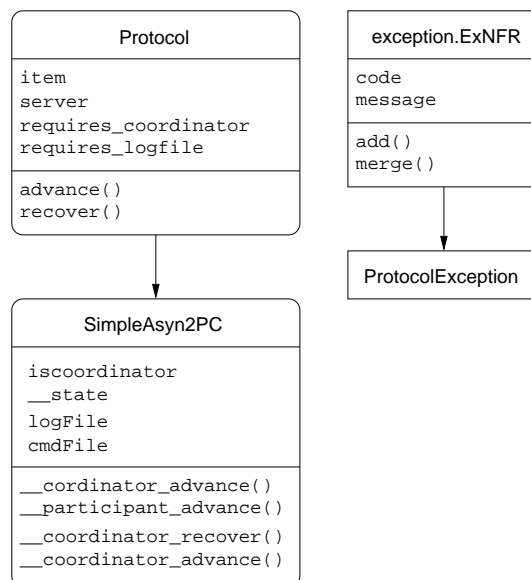
File: `protocol.py`

The protocol module implements the protocol subsystem.

As shown in figure 16, the module contains an exception class, a Protocol class that specifies the high-level interface that all ACPs should implement, and a SimpleAsyn2PC class, that inherits the Protocol class and provides functionality for executing our asynchronous 2PC protocol.

The SimpleAsyn2PC class contains the bulk of the code in the protocol module. We will now describe its function and how it interacts with the other modules.

Figure 16: The protocol module



### 5.9.1 Protocol initiation

To initiate a transaction controlled by the SimpleAsyn2PC class, the command

```
lock simple_asyn_2pc NFR_ZERONAME <File>
```

should be sent to the *coordinator Modules* Port. The NFR\_ZERONAME is the NFR empty name<sup>21</sup>, which signifies that a new transaction should be started. <File> is the name of the File to Lock.

The coordinator module is a special Module which Port runs protocol coordinators. This is necessary to allow a single site to execute both a participant and a coordinator: If both a coordinator and a participant were run by the same Port, the Port would need to interpret the protocol Commands to be able to send them to the right Protocol instance (for example, a prepare Command should go to the participant, while a vote\_commit command should go to the coordinator), rather than just looking at the TID. This is undesirable, because we wish to separate the implementation of Ports and Protocols with a simple interface. We should not have to modify the Port code when a new Protocol is implemented.

An initiation Command like this is not sent by Protocols, but by other NFR servers or applications/users. When the coordinator Modules Port receives the initiation Command object, it starts a coordinator Protocol, which

<sup>21</sup>Currently, a string of 32 '0' characters.

is inserted it into the Ports list of ongoing transactions. The Coordinator Protocol is responsible for sending prepare commands to the participants. Section 5.8 describes how the Port object acts when it receives a lock command.

The parameters to the `Protocols__init__()` method are the Server, the name of the item to be Locked, the name of the Module of the Protocols Port, the name of the log Module, and a specific log File if there is going to be a recovery.

For our SimpleAsyn2PC protocol the Server must be a MailServer. If there is going to be a recovery, the log File parameter is used as the this transactions log File, both during recovery, and normal protocol operation. If not, the Protocol generates a log File in the log Module.

After the initiation of a new Protocol object, its `advance()` method is called, which, if the object is a coordinator, causes prepare Commands to be sent to all participants through the MailServer.

### 5.9.2 Logs

The SimpleAsyn2PC protocol keeps two kinds of log Files in NFR:

- A state log (`logFile`), which stores the MailCommands sent.
- A Command list file (`cmdFile`), which stores all MailCommands received from the Port.

The root Files contain headers that identify the File trees as log Files and Command list files. Children of these root files contain string representations of MailCommands. Logging state as MailCommands simplifies both parsing of log entries and retransmission of commands. When a transaction completes, end-of-transaction records are written in the “youngest” children of the log and Command list Files. This enables a recovering Protocol to determine if a transaction has run to completion, or if the recovery should continue.

On recovery, the Command log acts as a copy of the channel at the time of failure, which serves two purposes:

- Guarding against duplicate Commands: If we know what Commands are on the channel, we can more easily determine what commands we have sent. For example, if a recovering coordinator finds a participant vote on the channel, it knows it must have sent the prepare command, so the coordinator does not have to retransmit it.

When the system goes down, the MailServer loses its list of read messages. The MailCommand will then reinsert old messages in the Ports working queue. If know what messages we have received based on the contents of the `cmdFile`, we can simply discard the old messages.

- Increasing performance: If we can retrieve messages from the log instead of waiting for the Port to send them to us, we can more quickly recover to the correct pre-failure state.

### 5.9.3 Protocol execution

SimpleAsyn2PC objects execute transactions more or less as described in section 2.6.1 (and figure 2).

As we have already observed, ACPs are driven by messages. When a Port receives a protocol Command, it looks at the Commands TID, and calls the `advance()` method (with the Command as argument) of the Protocol instance with the same TID.

The `advance()` methods returns true to the Port if the transaction should continue, or false if it should be stopped. If the return value is false, the Port removes the SimpleAsyn2PC object from its list of active transactions.

Pseudocode for the code followed by the SimpleAsyn2PC coordinator when `advance()` is called (the method name is `__coordinator_advance()`):

---

```

if __state == "initial":
    store the init Command in the cmdFile
    generate a new TID
    find the replicas of the file we are to Lock.
    generate a prepare Command
    log the prepare Command

    commit_voters = abort_voters = []

    set __state to "wait"
    send the prepare Command to all replicas

    return true

elif __state == "wait":
    store the received vote in the cmdFile
    if the Command is a vote_abort:
        add its sender to the abort_voters list
    else:
        add its sender to the commit_voters list

    if we have received all votes:
        if there are any abort votes:
            generate a global_abort decision Command
            set __state to "abort"
        else:
            generate a global_commit decision Command
            set __state to "commit"

```

```

        log the decision
        send it to everybody in the commit_voters list
        remove log Files and messages in the mailbox.
        return false
    else:
        return true

```

---

The pseudocode for the participant (`_participant_advance()`) is similar. Note that the participant may unilaterally abort the transaction, and if so, it does not expect to receive the decision from the coordinator.

---

```

if __state == "initial":
    store the prepare Command in the cmdFile
    store the prepare Commands TID

    if the File has an INVALIDNAME Lock on it or
       the File has a non-ZERONAME Lock on it:
        construct a vote_abort Command.
        log the vote
        send the vote to the coordinator.
        set __state to "abort"
        remove log Files and messages in the mailbox.
        return false
    else:
        set the Lock on the File to an INVALIDNAME Lock
        construct a commit vote Command
        log the vote
        send the vote to the coordinator.
        set __state to "ready"
        return true

elif __state == "ready"
    store the coordinators decision in the cmdFile
    if the decision is global_commit:
        log the decision
        set a new Lock on the File
        set __state to "commit"
    else:
        log the decision
        restore the old Lock on the File
        set __state to "abort"

    remove log Files and messages in the mailbox.
    return false

```

---

#### 5.9.4 Duplicate Commands

Commands sent by email need not arrive in the order we send them. A look at figure 2 shows that this will not be a problem for our protocol, as votes are not sent before prepare commands are received, and decisions are not sent before all votes are received. Other protocols may be sensitive to the order messages arrives in, and must implement their own methods for handling this (for example using logical clocks).

Duplicate commands, on the other hand, are a real possibility, and must be handled by our protocol. They may occur either because a channel duplicates a command, because a command is retransmitted or re-inserted during recovery.

We have chosen to let the receiver of a command handle duplicate commands:

1. All SimpleAsyn2PC objects have a `received_messages` list of all received MailCommands. When a MailCommand is sent to a SimpleAsyn2PC object, it is discarded if the MailCommand's uidl matches any in the list.
2. All received MailCommands are written to the `cmdFile` File. Upon recovery, the MailCommands in this file is copied to the list of received MailCommands, before any other MailCommands are accepted from the Port.
3. If a initiation or prepare Command is duplicated, Two transactions will be started. If they are concurrent, one or both will abort (as the relevant File is locked by one of the transactions). No harm is done except that some time has been wasted.
4. If a protocol command other than init or prepare is duplicated, and this command belongs to a finished transaction, it will simply be discarded by the Port, as no running Protocol has the same TID as the duplicated Command.

Thus, duplicate Commands should not cause errors in the SimpleAsyn2PC protocol, although they may degrade performance somewhat.

#### 5.9.5 Recovery

When a Port is initialized, it scans all root Files in its Module looking for log Files. When one is found (by comparing the first string in the file with a constant defined in the protocol module) it is parsed to find out what type of Protocol it belongs to. This type of Protocol is then initiated, and its `recover()` method is called. If it returns true, the Protocol is added to the

Ports list of running transactions. Otherwise, the Port forgets about that Protocol.

It is the Protocol implementations responsibility to return false if the log File has an end-of-transaction record in the “youngest” child File.

As mentioned in section 5.9.2 the SimpleAsyn2PC Protocol keeps two kinds of logs in NFR: a state log (logFile) and a Command log (cmd-File). When this Protocols recover() method is invoked, it retrieves all Mail-Commands in both logs, and figures out if it is a coordinator or a participant like this: If the first command in the state log is a prepare command, it is a coordinator. Otherwise it is, a participant. After that, either `__coordinator_recover()` or `__participant_recover()` is called.

Their code is somewhat involved (especially the participants, as it can abort in both phases of the 2PC protocol for different reasons), and mostly modifies the same state variables as the the code in the `__coordinator_advance()` and `__participant_advance()` methods. Basically, the algorithm described in section 2.6.2 is executed, by deducing the pre-failure state of the channel and the protocol from the contents of the two log Files.

The recovery protocol will be tested and described in detail in the testing section.

## 5.10 Conclusion

We have described our implementation of the design discussed in section 4, including the port, command, server and protocol modules. We also have given an overview of our system, and in-depth descriptions of how he handle data transmission, protocol execution, protocol recovery and duplicate messages.

## 6 Testing

By testing, we do not mean debugging, but confirming that our software conforms to our design. We believe, given our non-functional requirements (see section 3.1) that the most appropriate tests for our system are for function (or correctness), reliability and performance, in that order.

Thus, the goal of this section is to demonstrate the existence of our system, test its failure tolerance and performance, and explain how these tests may be reproduced.

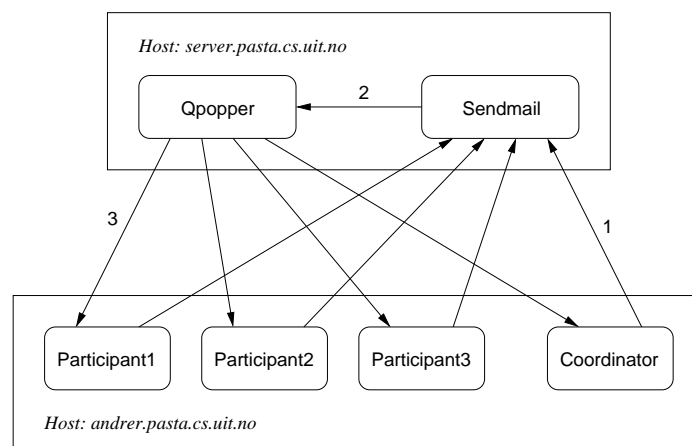
We will first explain the test system configuration, Demonstrate how the test scripts are used, say a few words about performance, and finally describe our testing of single failures for both the participant and the coordinator.

### 6.1 Test configuration

Briefly, our system was run on a Linux workstation with a Pentium III 450 MHz CPU and 128 MB RAM, running Mandrake Linux kernel version 2.2.16 with Python version 2.0. The host name of this machine was `andrer.pasta.cs.uit.no`.

The SMTP and POP3 servers (Sendmail version 8.9.3 and Qpopper version 2.53) were run on `server.pasta.cs.uit.no`, which is a server running NetBSD version 1.5\_ALPHA2 on an Intel Pentium III.

Figure 17: The demonstration setup. Email messages follow the arrows: from the participants/coordinator, to Sendmail, to Qpopper, and back to the participants/coordinators.



The machines were connected by a 100 Mbit/s switched Ethernet.

We used four different mailboxes for sending MailCommands, three for participants and one for the coordinator. Detailed software configuration is

given in the next section.

## 6.2 Demonstration

First, we will describe how the test scripts were set up (They can be found on the accompanying CD-ROM). A word of warning: The test scripts will delete *all* mail in the POP3 inboxes they use.

1. First, the necessary directories for three participants and a coordinator were created. In our case this was `/tmp/nfrN/`, where  $N$  is a number between 0 and 3 (0 for the coordinator, 1, 2, and 3 for the participants). These directories stored “replicas” of NFR data. We also created a `/src` subdirectory for each of the four NFR directories. The `/src` directories stored configuration and test scripts.
2. We Copied the files `/mnt/cdrom/test/startN.py` and `/mnt/cdrom/test/nfrconf` files to the corresponding `/tmp/nfrN/src` directories.
3. We configured the `NFR_DIR` variable in the `nfrconf` files to point to the relevant `/tmp/nfrN` directories.
4. `send_init.py` and `setup.py` was copied to `/tmp/nfr0/src`
5. A command shell was started from each of the four directories (from now on, we will call the prompt in shell  $N$  “%N”).
6. The variables configuring mail server, NFR source path etc. in the `.py` files were configured to match our local resources. Each participant and coordinator must have its own POP3 mailbox.
7. The `NFRINIT` environment variable in each of the four shells was set to `/tmp/nfrN/src/nfrconf`.

For demonstration, we ran two tests. The first was a regular protocol run with a global commit result. We began by creating a new repository with a coordinator Module and a data Module with a data File.

```
%0 python setup.py
```

Then, we start three participants:

```
%1 python start1.py -r
```

```
%2 python start2.py -r
```

```
%3 python start3.py -r
```

The `-r` option indicates that a participant should reinitialize its NFR replica by copying it from the coordinator. When we use `-r`, participants must be started before coordinators, because when the coordinator is started, a cache the participants need to copy is deleted by the coordinator.

Finally, we sent the `init MailCommand` and started the coordinator:

```
%0 python send_init.py
%0 python start0.py
```

When our system runs, some debug output describing the initiation of the processes and polling of inboxes will scroll by the shell terminals. Eventually, the strings

```
%0 SimpleAsyn2PC: Logging decision commit
%0 SimpleAsyn2PC: Sending the decision to the yes-voters.
```

were be displayed in the coordinators terminal, and the string

```
% SimpleAsyn2PC: new Lock = [some long NFR Name string]
```

were be displayed in the terminals of the participants. This signifies that each participant had successfully set a new Lock on the data File.

The second test was similar, except that one of the participants voted abort, thus causing a global abort decision. We did exactly the same as in the previous test, except we gave the `-a` option to the `start3.py` script. This caused a Lock to be set on the third participants replica before the system was started, thus forcing participant 3 to vote abort.

When participant 3 received the prepare command, it printed

```
%3 SimpleAsyn2PC: Abort! Abort! lockname = [several '1'es]
```

and replied to the coordinator with a vote abort.

After the coordinator had received all votes, it printed

```
%0 SimpleAsyn2PC: Logging decision abort
%0 SimpleAsyn2PC: Sending the decision to the yes-voters.
```

When participants 1 and 2 (the “yes-voters”) received this decision, they printed

```
% SimpleAsyn2PC: The decision was global_abort, log it.
% SimpleAsyn2PC: Restoring the old Lock
```

Meaning, of course, that they had executed the abort decision.

There is one remaining option to the test scripts `start2.py` and `start0.py` that we have not described: `-fS`. It causes all executing Protocol objects at a single site to fail in state  $S$ , for example

```
%0 python start0.py -f3
```

will cause the coordinator to fail in state 3. Failure and recovery is described in detail later in this section.

### 6.3 Performance

Although optimizing performance is not a goal for this project, we should at least make sure that things not go so slowly that the system is unusable.

We ran the first test from the previous section with the following modifications:

1. Mailbox poll interval was set to 1 second.
2. The SimpleAsyn2PC Protocol class was reprogrammed to kill the running process when a transaction committed.
3. The `time` command was used to measure the time used by the coordinator from its initiation until it was killed by the modified SimpleAsyn2PC class.

The `time` command typically reported that the coordinator used 4.5–4.7 seconds from initialization to transmitting and logging the global commit decision, a result which is certainly usable considering the transport used. Also interesting to note is the fact that the coordinators CPU utilization was about 5%, indicating that most of the time was spent waiting for network events.

Of course, there are a large number of factors that can determine the performance of a distributed system. Some particular factors that we believe influenced our experiment were:

- + The test was run on a small LAN during off-hours, which means low network traffic and machine resource usage.
- + There was only one active transaction.
- The system is an unoptimized prototype implemented in an interpreted language.
- All participants and the coordinator were run on the same machine, sharing CPU and network bandwidth.

Also, while retrieval of messages is faster when the system connects to its POP3 inboxes more frequently, this also causes the inboxes to be locked for a larger portion of the time, thus slowing down delivery of new messages.

## 6.4 Failure and recovery

In order to test correctness and reliability we will test all single failures of a participant and the coordinator, i.e. we will demonstrate the recovery protocol.

Of course, these tests cover only a small portion of the possible failure modes of our system<sup>22</sup>. However, as the system is designed around the need for a termination protocol (as it must be, given our asynchronous messaging system), failed participants and coordinators are always on their own when faced with recovery. As the system handles duplicate messages, and out-of-order messages are impossible (for the SimpleAsyn2PC Protocol), we believe that our system should be robust against multiple failures—if it can tolerate single failures.

We remind the reader that the SimpleAsyn2PC Protocol stores 2PC log entries in the “log list” and incoming Commands in the “command list”.

Recoverable state is then reflected by the log list and command list Files, and it only makes sense to test failure and recovery after data is added to these Files. This corresponds well with the failure modes described in section 2.6.2 (see also figure 2).

### 6.4.1 Participant failure scenario 1

The participant fails in the initial state. This is true if there is a prepare command in the command list. Recovery is accomplished by making a decision in the usual way and sending it to the coordinator, either entering the wait state or the abort state.

Test procedure: Regenerate the repository, and send the init MailCommand.

```
%0 python setup.py
%0 python send_init.py
```

This procedure is performed in all tests.

Start participant 1 and 3 as usual, and participant 2 with the `-f1` option:

```
%1 python start1.py -r
%3 python start3.py -r
%2 python start2.py -r -f1
```

This procedure is performed in all participant recovery tests (except that the parameter to the `-f` option varies): The `-r` parameter is always given when a participant is started and never when it is restarted after a failure.

---

<sup>22</sup>The participants each have 6 recoverable failure states and 1 correct state. The coordinator has 3 recoverable failure states, and 1 correct state. With three participants, the total number of possible states of the system is  $(6+1)^3 \times (3+1) = 1372$ , clearly too many tests to describe here.

All processes except participant 2 will run as usual. Participant 2's failure is signified by a warning message:

```
#2 SimpleAsyn2PC: Participant failure, scenario 1
```

Participant 2 is then restarted without the `reinit` or `failure` options.

```
%2 python start2.py
```

Participant 2 will recover its state from the log, send its vote (in this case, a vote commit), wait for the coordinators decision (in this case, a global commit), and act on it when it is received:

```
%2 SimpleAsyn2PC: The decision was global_commit, log it.  
%2 SimpleAsyn2PC: new Lock = [some long NFR Name string]
```

Meaning that the data File has a new Lock with the printed name on it.

#### 6.4.2 Participant failure scenario 2

The participant fails after writing the abort vote in the log but before sending the abort vote. This will be the case if the only item in the log is an abort vote Command. We recover by retransmitting the abort vote and entering the abort state.

The test is started as in the previous scenario, except that `start2.py` is given the `-f2 -a` options

We terminate participant 2 after the protocol fails, and initiate recovery as in the previous test.

We see on the terminal that participant 2 reads its vote from the log, transmits it to the coordinator, restores the old Lock on the data File, and aborts the transaction.

The coordinator, upon receiving the abort vote, sends a global abort message to the other two participants, who similarly restore the old Lock and aborts.

#### 6.4.3 Participant failure scenario 3

The participant fails in the ready state. This will be the case if a commit vote is the only item in the log, and the prepare command is the only command in the command list. This is simply handled by retransmitting the vote and reentering the ready state.

The test is performed as previous scenario, except that `start2.py` is given the `-f3` option, and the `-a` option is removed.

The system is waiting for participant 2's vote. When the coordinator receives it, it transmits the global commit decision, and the participants act on it as usual, setting a new Lock on the data File.

#### **6.4.4 Participant failure scenario 4**

The participant fails while aborting or committing. This is true if the final decision is on the command list, but no commit or abort record exists in the log. The corrective action is to execute the global decision command found in the command list.

This is tested by starting participant 2 with the `-f4` parameter for a global commit decision. To test with a global abort decision, set the `-a` parameter on participant 3. The other parts of the system are started as usual.

As can be seen on the terminal, the coordinator receives all votes, decides, and transmits the decision the participants. Participants 1 and 3 execute the decision, while participant 2 goes down.

When it comes up again it can proceed with the transaction independently of the rest of the system. It will find the decision in its Command list File, and can simply execute the decision.

#### **6.4.5 Participant failure scenario 5-1**

The participant fails in the abort state. True if there are two items in the log and the last item is the global abort command. We recover by aborting the transaction.

This scenario is tested by setting the `-a` parameter on participant 3 and the `-f5-1` parameter on participant 2.

As in the previous test scenario, the participant is able to continue independently of the other parts of the system, and simply executes the global abort decision.

#### **6.4.6 Participant failure scenario 5-2**

The participant fails in the commit state. True if there are two items in the log and the last item is the global commit command.

As in the previous scenario, we recover independently of the rest of the system by executing the decision found in the log. The parameter to participant 2 is then `-f5-2` (and there is no `-a` parameter to participant 3).

#### **6.4.7 Coordinator failure scenario 2**

The coordinator has fewer states than the participants because it does not have alternative paths to the commit/abort states, and it does not change the state of items in NFR.

Our system can not recover from failure scenario 1 described in the recovery protocol (section 2.6.2) because no state has been written to the log or Command list Files.

Failure scenario 2 is the first recoverable scenario: The coordinator fails after the prepare Command is written to the log File, but before the prepare Command is sent. We know that we must recover from this condition if there is a prepare command in the log but no votes on the channel. We recover by (re)transmitting the prepare Command and entering the wait state.

To test this recovery, we initiate the system as in the participant failure scenario 1, except that the `-f2` option is given to the `start0.py` (coordinator) script.

The terminal will print

```
%0 SimpleAsyn2PC: Logging the prepare command
%0 SimpleAsyn2PC: Coordinator failure, scenario 2
```

And we terminate the coordinator.

When we restart it (without the `-f2` option), it finds the prepare command in the log, and retransmits it. This will start a “duplicate” transaction at the participants. At least one of the transactions will be aborted. In our most recent test, the first transaction committed, and all participants set a new Lock on the data File.

#### 6.4.8 Coordinator failure scenario 3

The coordinator fails while in the wait state. This is true if the prepare command is in the log and there are messages on the channel (or in the Command list). We rectify this by entering wait state, and updating the coordinators list of votes. If all votes are received, we make a decision and transmit it.

This test is started in the same manner as the previous test, except the `-f3` parameter is given for the coordinators test script.

After the failure, the coordinator is restarted, and, if all votes are the channel, may make a decision immediately. Otherwise, it will wait until all votes are received.

The participants commit the new Lock as usual.

#### 6.4.9 Coordinator failure scenario 4

The coordinator fails after logging its final decision, but before informing the participants of the decision. This is the case if the final decision in the log File but the end-of-transaction record is missing. To recover, we simply transmit the decision. If it turns out that the decision was transmitted before the failure (impossible in our tests, as the failure code executes before the decision transmission), the participants will just discard the extra decision as a duplicate message.

It is tested by giving the `-f4` option to the coordinator test script.

Again, the participants commit the new Lock as usual.

## 6.5 Conclusion

We have described how to demonstrate and test our system, commenting on usability and performance, but focusing on reliability by testing all single failures for the participants and the coordinator.

We have shown that our system in itself uses few resources, and that most of the time is spent waiting for network events. The total time used indicated that our systems performance was adequate, at least in our particular setting.

We have also exhaustively tested all single failure modes, and demonstrated our systems robustness in the face of such failures. We also believe that the fact that participants and coordinators do not rely on other participants/coordinators for recovery (i.e. we have no termination protocol) implies that the system should be able to handle multiple failures gracefully.

## 7 Conclusion

### 7.1 Accomplishments

This thesis has addressed theory and problems related to distributed systems, replicated storage, asynchronous communication, transaction protocols and the New File Repository.

We have used the knowledge gained from this to design, implement and test an asynchronous Lock distribution protocol based on the 2PC protocol.

Specifically, we have:

- Designed and implemented an asynchronous communication channel using email through external POP3 and SMTP servers. While this approach to asynchronous messaging has its problems, namely performance and dependence on external systems, it does have some advantages: First, that it provides a store-and-forward channel, which simplifies the implementation of our 2PC protocol, and second, that utilizes a well-tested and pervasive infrastructure.
- Designed and implemented an infrastructure for interpreting and executing commands from the NFR command set.
- Designed and implemented a 2PC protocol for Lock distribution that uses the asynchronous communication channel for its protocol messages. The purpose of this protocol is to provide guaranteed all-or-nothing distribution of Locks in NFR. These Locks will be used by user-level concurrency control protocols to ensure consistency of data stored in NFR.

Based on our tests, we believe that our protocol is stable and able to recover from all single site crash failures, and that it will be able to handle the concurrency protocols required by NFR applications.

### 7.2 Future work

Future work will consist mainly of expanding our system and integrating it better into NFR. We have identified four major areas for improvement:

- Implementing the remaining NFR command set commands. This is of course necessary to make NFR usable as a distributed storage architecture, and a natural parallel development effort to creating the first NFR applications. Particular effort might go into generating good error report Commands to give the users the best possible information about the state of the system in case of failures. We should also include the ability to handle malformed messages.

- Implementing a minimal internal MTA for sending and receiving emails, instead of relying on external servers like Sendmail and Qpopper. This will allow us to send and receive emails concurrently, without any need for polling (and locking) an external server.
- Adding Servers and Commands for other transport mechanisms. The most likely candidate is UDP. It is fast and more naturally suited to the asynchronous nature of NFR than TCP, and a suitable complement to SMTP.
- Adding new Lock distribution protocols, either for using new transport protocols, or for other reliability requirements than our 2PC protocol. For example, the requirements to no failure in the SMTP channel needed by the SimpleAsyn2PC protocol, while working fine in the lab, is not very realistic for the real world. As our protocol is able to handle duplicate Commands, we could extend it by using some (long and increasing) timeout to “detect” lost messages, and retransmit them until we get a reply.

As we can see from this list, there are several possible improvements that may turn the New File Repository into a very useful tool in the distributed world of the future.

We hope that this report provides some of the knowledge needed to do this, and that our system provides a solid base to work from.

## References

- [1] An overview of the PASTA project.  
<http://www.pasta.cs.uit.no/Pasta/overview.html>
- [2] Python language resources. <http://www.python.org>
- [3] Tage Stabell-Kulø, Feico Dillema and Terje Fallmyr (1999): *The Open-End Argument for Private Computing*. International Symposium on Handheld and Ubiquitous Computing.
- [4] Gaute Moxnes (1997): *Design og implementasjon av replikering i File Repository*, Cand. scient. thesis in Computer Science (Norwegian).
- [5] Christine Lund (2000): *Design og Implementasjon av 3PC for New File Repository*, Master of engineering thesis in Computer Science (Norwegian).
- [6] Michael Hammer and David Shipman (1980): *Reliability Mechanism for SDD-1: A System for Distributed Databases*. ACM Transactions on Database Systems, Vol 5, No. 4.
- [7] Vassos Hadzilacos, Sam Toueg, Fault-tolerant Broadcasts and Related Problems, page 97-145. Sape Mullender (red.) (1993): *Distributed Systems*, Addison-Wesley/ACM Press, ISBN 0-201-62427-3
- [8] Fred B. Schneider, What good are models and what models are good, page 17-25. Sape Mullender (red.) (1993): *Distributed Systems*, Addison-Wesley/ACM Press, ISBN 0-201-62427-3
- [9] Ramez Elmasri, Shamkant B. Navathe (1994): *Fundamentals of Database Systems 2nd. ed.*, The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-1753-8
- [10] Andrew S. Tanenbaum (1996): *Computer Networks 3rd. ed.*, Prentice Hall International, Inc., ISBN 0-13-394248-1
- [11] M. Tamer Özsu, Patrick Valduriez (1999): *Principles of Distributed Database Systems 2nd. ed*, Prentice Hall, Inc., ISBN 0-13-659707-6
- [12] Roger S. Pressman (1997): *Software Engineering, a Practitioner's Approach 4th. ed., European adaption*, McGraw-Hill/Cambridge University Press, ISBN 0-07-709-4115
- [13] Nancy A. Lynch (1996): *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-348-4
- [14] Jonathan B. Postel (1982): *Simple Mail Transfer Protocol (RFC 821)*.

- [15] David H. Crocker (1982): *Standard for the format of ARPA internet text messages* (RFC 822).
- [16] J. Myers and M. Rose (1994): *Post Office Protocol - Version 3* (RFC 1939).
- [17] J. Myers, M. Rose (1996): *SMTP Service Extensions* (RFC 1425).
- [18] N. Borenstein, N. Freed (1993): *MIME (Multipurpose Internet Mail Extensions) Part One* (RFC 1521).
- [19] T. Berners-Lee, L. Masinter, M. McCahill (editors) (1994): *Uniform Resource Locators (URL)* (RFC 1738).

## A The CD-ROM

The CD-ROM uses the ISO9660 filesystem, with the Joliet extension. There is a browseable index.html file and a README file with further information about the contents on the root directory of the CD-ROM.

It contains various material related to our work, including:

- This report in PostScript, PDF and HTML format.
- Some of the references from the bibliography, in various formats. See the README file for details.
- The source code of NFR and our project, including test scripts.
- Compressed versions of the entire CD-ROM