A File Repository Based Distributed Diary in a Mobile Environment

Dag-Frode Olsen

Thesis for the Degree
`‘‘Sivilingeniør’’`

Department of Computer Science,
Institute of Mathematical and Physical Sciences
University of Tromsø, autumn 1996

December 13, 1996

Preface

Project Background

The background for this project is the projects Moby Dick, PASTA and the Global Distributed Diary (GDD), which are all described below. All project focus is on mobile computing, but from different angles. The Distributed Diary (TDD) application is an application which is influenced from all these projects.

Moby Dick

An exciting prospect for the next decade is the deployment of a new generation of hand-held computers. Wireless networking greatly enhances the usability of these mobile computing devices. They will be used as means to participate in an on-line information community.

The overall goal of the Moby Dick project is to design an architecture that releases the full potential of the hand-held computer as we envision it. The Distributed Diary is a demonstrator application made to demonstrate this architecture.

This is some of the background for the Moby Dick project, a project founded by Esprit-IV[1]. The project is a cooperation between the University of Tromsø[2], the University of Pisa[3] and the University of Twente[4]. One of the goals is to make a Personal Digital Assistant (PDA) - "the Pocket Companion". This is a small computer and communication system which could replace many of the paraphernalia one carry, including cash, passport, keys, diaries, phone etc. It should also let the user utilize the location independent services already offered on computer networks like electronic mail. Further this PDA should offer location dependent services like ticket buying and looking at time-tables on a local railway station.

Global Distributed Diary (GDD)

The GDD project is supported by "Norges Forskningsråd" [5]. Its goal is to examine consistency problems in the integration of mobile and stationary machines in a computer network, and the security consequences following that. The GDD

---

[1] LTR 20422

[2] Norway

[3] Italy

[4] The Netherlands

[5] The Research Council of Norway

project should construct a distributed diary, with the necessary system support for dealing with consistent read/write in replicated data between a large set of heterogeneous and mobile computers.

Pasta, the Project

The Pasta project addresses the consistency problems that occurs when data are replicated in a distributed system with mobile machines; characterized by frequent disconnection and varying communication capabilities. The Pasta approach is to provide tools for finding the balance between consistency and desire for progress in a system subject to variable connectivity and numerous write-write conflicts [Fallmyr et al., 1995].

Motivation

Many people look at mobile computing as a new paradigm in computer science. Many new problems and solutions have appeared, many which belong to the old fields of research in computer science, like consistency, file systems and database systems.

As one tend to get more and more small electronic devices associated with oneself, like cellular phones and bleepers, the idea of combining these with credit cards and keys to a mobile computer sounds very alluring.

This, along with the fact that mobile computers like PDAs are predicted to have a major breakthrough among ordinary people have made mobile computing an interesting field to investigate further, and greatly motivated me for doing an effort with this thesis.

Acknowledgments

This master of engineering thesis was written at the University of Tromsø during some hectic months the autumn of 1996, and mark the end of 4 1/2 year of “blood, sweat and tears”. As supervisors I have had professor Dr. Gunnar Hartvigsen and Research Scholar Tage Stabell-Kulø. I wish to thank them for their cooperation and professional supervising.

Thanks to all members of the PASTA/Moby Dick/GDD project, especially assistant professor Terje Fallmyr, master students Ronny Arild, Magne Rassmussen, Gaute Moxnes and Lars Slettjord and master of engineering students Dag Brattli, Tor-André Grenersen and Ingeborg Østrem Hellemo for technical and moral support.

I would also like to thank all the students of our class, Vegard Berg Johansen, Tor Murvoll, Vegar Næss, Haiyan Yang and Kåre Jørgen Lauvset.

And of course, thanks to my family Anne-Lill, Astrid and Dag, for all patience and love.

Tromsø, December 13, 1996

Dag-Frode Olsen

Abstract

This thesis deals with a specific approach to the consistency problem in distributed systems. By writing a distributed diary which works in a mobile environment consisting of heterogeneous machines with greatly variable communication abilities, we get a means to investigate some of the special aspects mobility gives to consistency. Our solution uses the File Repository (FR) as a way of distributing diary entries between users.

Since mobile machines are dependent on battery power, and since mobile machines moves into areas without communication possibilities they will during some periods of time be unreachable and must therefore support off-line operations.

When a system tries to handle off-line operation like the distributed diary, one is certain to have consistency problems.

We show that the user should be involved in the decision loop, meaning that the user has the final word when a computer system makes a decision concerning consistency. We show that the machines *eventually* become consistent and that conflicts resulting from the reconnection of the machine are *not* a big problem due to the semantical knowledge of the user.

We also show that this semantical knowledge is greatly reduced when communicating diary entries with other people, and that this leads to a more pessimistic consistency model.

List of Figures

Contents

Chapter 1

Introduction

> *PUDDER'S LAW:*
> *Anything that begins well will end badly.*
> *(Note: The converse of Pudder's law is not true.)*

1.1 Background

Rapid advances in the development of portable and mobile computers, combined with the possibilities wire-less communication offers have rapidly increased the interest in mobile computing.

The growth in mobile computing has made both the networks and the computers in most organizations more heterogeneous. The users of computers tend to get more and more computers associated to them. Applications are forced to cope with this heterogeneous environment.

This environment has also different demands from the operating system and the services normally supported by the operating system like file systems and networking protocols. This has lead to renewed interest in these traditional fields of research.

Mobile machines will undoubtedly enter areas of poor or none network services. The mobile machine is also due to battery limitations, and will try to save power by reducing the activity of the machine. Disconnection may therefore happen frequent and in unpredictable ways, leading to problems for the application and system programmers.

The problem is that the ordinary way of solving computer problems are information hiding. The system programmers tries to hide all problems for the user. This often causes problems for mobile applications, since many of the situations are non-transparent. One can't hide the fact that the machine is off-line to the application and should not hide it for the user.

This thesis is based on my semester thesis which focused on the implementation of a distributed diary for *one* person [Olsen, 1996].

1.2 Problem Definition

There are two aspects of building a "Pocket Companion". Both the hardware and the software must be built. The Distributed Diary is meant to be a demonstrator of a software system running on a "Pocket Companion". By being a demonstrator our distributed diary is part of a feasibility study which goal is to examine in depth what is possible and what is not on a PDA like the "Pocket Companion".

Although there already exists numerous solutions to diary applications, a diary application embraces many of the problems one need to answer when building distributed applications, and should give more information about many of the real world parameters involved. Therefore, the main *problem definition* for this thesis is:

> *To construct and evaluate a distributed diary which support a number of heterogeneous machines with greatly different resources and which uses communication with varying quality.*

Since The Distributed Diary support a heterogeneous number of machines one has to distribute data between those machines. Distribution of data between computers involves well-known problems like consistency. Consistency in distributed systems have been the subject of great research efforts over the years.

However, sharing data with a computer which often is unaccessible due to disconnection is a problem. It may lead to updates in partitions not seen in other partitions, thus making the system inconsistent. An example of this is you making an appointment with a friend without his knowledge.

This leads to a further refinement in the problem definition with focus on consistency problems:

- Which consequences do off-line interaction have for a mobile application?
- Which consequences to mobility do consistency have in a mobile application?
- What level of trust[1] could people have in systems not being guaranteed consistent due to mobile clients.

1.3 Method

Denning et al. uses three paradigms to describe how to solve computer science problems [Denning et al., 1989]. The three are: The theory paradigm which has its foundation in mathematics, the paradigm of abstraction with basis in experimental natural science and the design paradigm, which origin is in engineering.

---

[1] We do not mean trust in a security context, but rather "trust" to the correctness of the application.

The method used in this thesis is based on the design paradigm. This means that from the problem we first decide which types of requirements the problem has. Then we design a solution based on these requirements. After designing we try to implement it seeing how the solution satisfy the requirements. If the solution is not good enough, the whole process is repeated. We redesign and re-implement. This is repeated until the solution satisfy the requirements.

In this thesis we start by defining and specifying the problem. This is done on the basis of an examination of theoretical aspects and existing systems. Then we design a solution to this problem, which we implement. The implemented solution is then tested against the requirement specified. The tests and the system are then evaluated.

1.4 Restrictions

This thesis uses the FR to distribute appointment files between clients. The FR is a system under development and does not support everything that it is meant to support when finished. Some of the supported primitives have not been heavily tested. All functionality that FR says it will support is assumed to work.

The system presented in this thesis is a local network variant, meaning that only a single file repository will be used. Only persons with access to that file repository will be able to use the application. A global system is beyond the scope of this thesis, even though the FR handles some of this by replication of files in different File Repositories[2]. There are two aspects of a global system. The clients should be able to roam the world freely, and everyone should be able to access other people's diary entries from all over the world. The first is a problem with the current network technology, as we will discuss further later, while the other problem could be solved by replication. Both aspects are not vital for this thesis.

Security and privacy are great problems in systems that are very personal, like a PDA will be. These challenges have not been considered much in this thesis, because of the higher level of complexity, and because security is not the issue of this thesis.

1.5 Major Results

We have designed and implemented a distributed diary application that supports one user with many machines to have a consistent view between those machines. The Distributed Diary also supports multiple users by using the File Repository for distributing diary files.

The application survives disconnection and partitioning of the network, with updates being done in any partition. The application is consistent within partitions during a disconnection period, and is brought back to a consistent state as the partitions reconnect. This consistency scheme is based on an optimistic consistency model, the *eventual* consistency model.

---

[2] This is work currently being done, and will be finished during spring 1997.

The application has been tested in a rough environment where the possibility of message loss is big. It shows that the application eventually gets consistent with the other clients.

1.6 Organization of the Thesis

The rest of the thesis is ordered as follows. In chapter two, we look at more theoretical issues like mobile and consistency problems, existing systems, and evaluation criteria. Then, in chapter three, we consider the requirements for the system and for the tests. Chapter four is about the design of the system and the tests, while chapter five is about the implementation. The testing and results will be discussed in chapter six, while the thesis is ended by a discussion chapter (chapter seven) and a concluding remark and further work chapter (chapter eight).

Chapter 2

Theory

> *GREEN'S LAW OF DEBATE:*
> *Anything is possible if you don't know what you're talking about.*

We shall in this chapter try to give the reader an overview of the theoretical framework this thesis is built upon. We begin this with a discussion of some the theories. Then we look at mobility and consistency problems. We then introduce some existing systems. The chapter is concluded by looking at how to evaluate these systems.

2.1 What is Mobile Computing?

Mobility is a term that often is misunderstood by many people. It is easy to think of the term as "every machine that could move from A to B is mobile". This is a too wide definition since it includes *all* computers made. We start by looking at mobile computing from the machine and communication viewpoint.

In this thesis we talk about a mobile system that uses some sort of communication when operating normally. The system should be able to communicate while moving. This means that a mobile machine must has a wireless communication unit. Mark that the size or weight of a mobile system doesn't influence on whether or not it's mobile. A big computer in a ship is definitely mobile if it has good enough communication with the rest of the world.

This rules out many of the *portable* systems. These systems are often defined by having no communication while moving. An example is a notebook with a modem.

This machine would be much more mobile if it has a wireless communication device which could communicate while moving. This implies something like a radio communication card or a GSM cellular phone for example.

However, for applications many of these problems are hidden, and should be so. When writing applications for mobile machines other things should be focused on, like being able to continue running an application even when the communication link is broken for some time. One could argue that an application is not mobile if one has to restart the application if the communication is lost and comes back. This could happen because the mobile machine hibernates to save energy, or is in a room which block its signals. We want an application which handles off-line operation.

Routing is also an issue here. A mobile application should survive moving from one domain to another. This is not as easy as it seams. (A further discussion of this can be read below.) Because a communication link is addressed as an IP address and socket number pair, problems occur when a machine leave its natural domain. Where should the packets be routed, if the machine is not able to receive the packages?

This implies different levels of mobility.

- Local mobility. The machine could move freely within a physical subnet. An example would be to have machines move freely under the domain cs.uit.no.
- Organizational mobility. The machine could move freely within a number of subnets with is provided by the same provider. Example is uit.no.
- WAN mobility. The machine could move freely around the world.

One could say that the difference in these levels are the transparency of mobility. The less the level of mobility, the less the level of transparency. The goal for a mobile system is to have true transparency, meaning that applications should not worry about mobility at all.

2.2 Mobile Computer Problems

The rapid growth of laptop and smaller computers have been the basis for a great deal of research about mobility. These machines have different demands to resources like distributed databases, file systems and networking. The computers tend to get smaller and we see that machines like the personal digital assistant (PDA) has arrived [Koch et al., 1993, Forman and Zahorjan, 1993]. This trend will undoubtedly continue and we will see a growth of very small mobile computers, small enough to fit in a pocket, which could replace many or all of the paraphernalia of modern people. Being a computer carried along at all times it will include a large amount of personal information, like keys, driver license and credit cards, but also serve as a way to access information easily, like scheduling a meeting with somebody or look at the local rail station departure times. It could also serve as a plain cellular phone, or an e-mail sender and receiver.

These machines will need network access in order to communicate with the world. The machines tend to have little resources, and is greatly dependent on battery life. Having little resources they tend to depend more on remote information since they have little storage capacity. They need to access remote resources like other phones (phone-call), bank accounts (wallet applications), and the national vehicle register (in order to drive), or the time-tables of the local railway station.

This lead to problems like maintaining a communication link between the mobile host and other hosts, because the mobility leads to problems caused by moving between different communication cells with great variation in bandwidth. The bandwidth may differ from 0, when the machine is disconnected, to several Mbps if connected to a LAN.

When the machines move around, one get two main problems, the first is to get the data to the machine physically, while the other is to support a consistent address space when machines move from one LAN to another.

Several solutions to the problems moving around introduced have been offered. Most of these conclude with a transparent look, meaning that the complexity of maintaining the link is up to the communication system. The most natural place to add this increased level of complexity is at the network layer (Like IP in the TCP/IP world [Stevens, 1990].) [Badrinath et al., 1993]. Then the network layer deals with routing of the data to the right place, even though the actual machine has moved.

There exist solutions to the physical moving problem. Cellular phones have had these problems, and have solutions to how to locate a telephone and sending information to it within a cell and between cells. This is because, if you look at it from a computer science point of view, the telephones belong to the same administrative net segment, like NetCom or Telenor here in Norway. The companies have solved possible problems by letting customers of either company use eachother's network.

This can also be said for computers, but the number of computer networks introduces other problems, due to machines moving from one local network to another. One solution to this is indirect IP [Badrinath et al., 1993]. This work by using forward pointers so that the messages are routed to the right place. See fig. 2.1

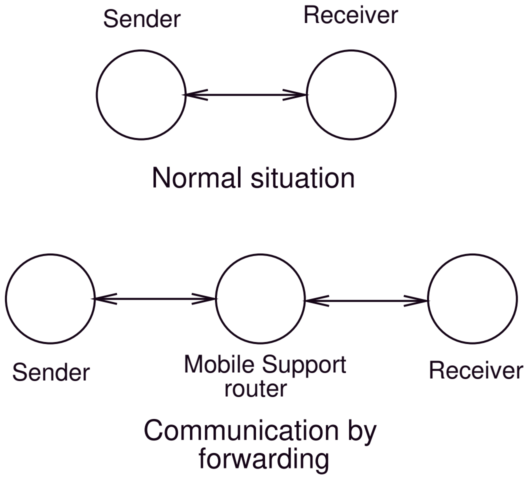


Figure 2.1: Indirect IP

Indirect IP is not an optimal solution since the route is not optional and since single failures on a site can result in a loss of possibility to communicate with the remote host. Cellular Digital Packet Data (CDPD) has solutions to these problems and to the whole physical moving problem [Garg and Wilkes, 1996].

Shortly the system is a connection less system. The mobile hosts tells their mobile base station where they are and thereby updates all routing tables on the way. If a packet to the mobile host doesn't see any of these updated routing tables the mobile base station is contacted (this is not visible for the sender) and forwarded to the mobile host. Any replies from the mobile host will then update all routing tables for optimal routing.

The bandwidth problem is one which also has been the issue for many research projects. This issue interests researchers from file-systems to multimedia applications. The file system Coda for instance has a system which allow disconnected operation by locally caching data before going off line, a very nice property for portable systems [Satyanarayanan et al., 1993]. The problem with Coda is that the system assume that the bandwidth between going off-line and getting back on line is 0. Truly mobile system will need access to resources while not physically connected, by using infrared or radio links. One could use experience from the multimedia research when dealing with these problem. Quality of Service (QoS) and renegotiation of it is one of the multimedia society's way of solving variation in bandwidth [Vogel et al., 1995].

Another problem by using the Coda file system to solve mobile problems is that the amount of secondary storage for pre-fetched data is low, or even non existing in small PDAs. One could argue that the demand for files are less on a mobile machine than on a portable, but in such a resource less environment as a mobile machine's, every resource spent in vain is very expensive. The Coda file system has to pre-fetch much more data than is actually used, with the hope that enough data was pre-fetched.

Mobile machines have many physical restrictions which greatly influences all systems that include mobile machines. The greatest restriction is that of battery life. Advances in battery technology are slow. Advances in processor technology are fast, and the new chips tend to be more power consuming than old chips. This is due to larger chips and more electronics.The clock frequency of the chip also greatly influence the power consumption. Discs are also very power consuming, which makes nearly all of the small mobile machine hard-disc less by using special types of RAM that is non-violate (Flash-RAM). Portable machines uses starting and stopping of disks to save power. Other equipment also draw power, like the communication device. Another factor is that the cost, in power, of sending a message is about 10 times higher than receiving, something an application should exploit by making the client send as little data as possible [Forman and Zahorjan, 1993].

Another problem for mobile machines is that they often require special hardware. One has to make special versions of different hardware so that they fit the strict requirements mobile machines have, making the machines more expensive for the same amount of computation power. An example of this is the Nokia 9000 PDA.

Special problems for mobile computers

There are more problems that are special for mobile computers. We will only mention them briefly in this thesis since they are not the main issue.

Privacy is one big problem due to the fact that a mobile machine tend to get more personal as more sensitive information are placed in the mobile machine. The mobile machine must also be able to do things on behalf of the owner, like saying "I'm buying this item on behalf of my owner, who is holding me" (authentication). Further the mobile machine must depend on that transactions done by it was done in a secure way. This has lead to the necessity of developing secure software on the mobile machines, even though security could be a big cost for the small mobile computers, in terms of extra hardware and/or software.

The chance of data loss increases on mobile computers, since they're more exposed to physical damage, loss and theft.

The size of a mobile machine require small user interfaces. The GUI environment used on todays portable machines may not be good for small mobile machines. The size inflicts also the type of input device used to communicate with the computer. The ordinary keyboard may disappear. An example of an other input devices used on small mobile computers are pens with handwriting recognition. Speech recognition could also be used.

Small mobile computers do not provide large amounts of non-volatile storage, if the computer provide any at all. Disks are often considered too power expensive for a mobile computer. Other techniques used are to store the usual applications in ROM.

To visualize some of the problems we have collected some information about various mobile and portable machines.

Product	Ram	MHz	CPU	Time	Weight	Display
Apple MessagePad 130	2.5MB	20	R-610		480 g	320x240px 96x72mm
IBM Model 800	8MB	50	i486		2.16kg	40x191mm
HP OmniGo 120	1MB	16	i186	2-8w	331 g	240x240px
NOKIA 9000	8 MB	24	i386	2-30h	397 g	640x200px 115x35mm
Omnibook 800*	16 MB	133	i586	<3 h	<2.4kg	800x600px

Table 2.1: Characteristics of different mobile and palm-top machines (Numbers are the developers own, found at their WWW home pages.)
* A portable machine

2.3 Application Specific Problems

This thesis is anchored theoretically near two major research issues. In this section we look closer at these issues which are consistency in distributed database systems and consistency in file systems.

2.3.1 Consistency

One could argue that it is theoretically not possible (nor practical) to have total consistency at all possible times in a distributed system. This is because there is no such thing as atomic update in a distributed database system. Note that atomicity is only defined from the perspective of the caller of an operation [Gray and Reuter, 1993]. Every local update can be divided into a sequential set of machine instructions, in which any machine instruction could fail. This does not mean that nothing can be done. It does mean that one have to make certain assumptions about a system, in order to be able to do anything at all.

Since mobile computing must handle disconnected operation this often result in a certain degree of inconsistence between the disconnected node and the rest of the world. Files and databases could be changed locally at the disconnected host.

This could be seen as a partitioning of the network. We know that consistency in partitioned networks are a problem without one single solution, rather a tradeoff between the cost of "lost opportunity" and the cost of guaranteeing "absolute" consistency, or correctness. The cost of maintaining consistency over a number of distributed sites may be unbearable if not impossible if the number of sites are large and the number of partitions are large [Pitoura and Bhargava, 1995].

Partitioning of networks have been studied much in respect to distributed database systems and distributed file systems. We first look at how the database community have tried to solve the problem.

Distributed database systems

There are traditionally two ways of looking at consistency in distributed database systems. These are syntactical and semantical, both are further divided into optimistic and pessimistic approaches [Davidson et al., 1985].

The background for this division is that data are replicated to increase the level of *availability* and performance of a distributed database system. However, the *correctness* of the data must be maintained. All copies of one data item should agree on one value.

Since the data are replicated on different processors, with possible faulty communication links between them, the probability that one processor or a network link is faulty increases (it is often impossible to distinguish a network failure from a processor failure), thus making the data unaccessible, and making the mutual consistency harder to enforce.

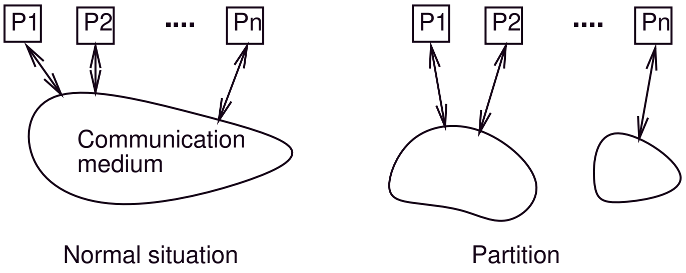


Figure 2.2: Partition of networks

A *Partition failure* as shown in figure 2.2 is one of the most malicious of these faulty conditions. This means that the distributed database is divided into more than one partition, without any possibility to communicate between those partitions. If not detected, updates could be done in different partitions of the distributed database, violating the correctness demand.

Since availability and correctness isn't two independent goals one has to make trade-offs between them.

It follows from the argumentation over that the difference between optimistic and pessimistic strategies are that pessimistic strategies prevent inconsistent states in a database by limiting availability, thus making worst case assumptions. This leads to restrictions in what are allowed to do, but easier consistency handling.

The optimistic strategy tries to maximize the availability, allowing transactions to be issued even though these transactions may lead to an inconsistent state when partitions reconnect. The strategy assumes that conflicts rarely occur, and/or are easy to detect and correct. In other words the correctness criteria is less strict.

Syntactical approaches uses serializability as the correctness criterion and check correctness by comparing read-sets and write-sets, while a semantical approach use either the semantic of the transaction or the database to define correctness [Elmasri and Navathe, 1994].

Davidson and Garcia-Molina describes some classical consistency strategies in [Davidson et al., 1985]. We will mention some of the most important here.

Of the syntactical approaches the *version vector* protocol is one of the known syntactical optimistic protocols [Parker et al., 1983]. This protocol was proposed to detect write-write conflicts between copies of files. Each file has a *version vector* associated with it. Conflicts could be detected by comparing version vectors.

Of the syntactical pessimistic protocols the voting protocol is a well known one [Gifford, 1979]. Here every copy of a replicated item has assigned a number of votes to it. To read or to write that item one need to collect a read or a write quorum of r or w votes. These quora must satisfy two constraints:

- $r + w$ exceeds the total number of votes v assigned to the item, and
- $w > v/2$

Any read quorum are guaranteed to have the current copy of the item since the intersection between every read and write quorum is non-null. This constraint guarantees further that an item could be read in a partition and written in another during a partition. Due to the second constraint two writes could not happen at the same time in two different partitions.

An optimistic semantical approach protocol is the "Weak Consistency" protocol, which looks at the possibility to weaken the correctness criteria for many read-only transactions [Garcia and Weiderhold, 1982]. This is especially nice in partitioned networks since it would allow more transactions to be done. A read transaction does not alter any database state, so their execution does not generate inconsistencies.

The voting protocol is extended into the "Quorum Consensus" protocol which uses type information to increase data availability [Herlihy, 1984]. The abstract type defines type-specific criteria for correctness. These are serial specification and behavioral specification. Each data-item is also represented by its history, a sequence of time stamped operations. A site uses read and write quora along with sub-histories from the other sites to decide whether to perform the action or not.

The goal of transactions are much the same as the goal for our distributed diary. Both systems want to agree on one value. But there are obvious weaknesses in the traditional transaction model. These weaknesses are in particular visible when looking at commit/abort rules. A system must use timeouts to be able to proceed, and timeouts are not good for mobile systems which may stay off for a long time, and still be part of the system.

Distributed file systems

The calendar system that has been made here, lies in many ways close to both distributed database systems and distributed file systems because it uses the FR to distribute the calendar information. A file repository could be looked upon as a database of files, or as a coarse grained database using flat files.

One common way to look at consistency in file systems are the Coda way [Satyanarayanan et al., 1993]. The Coda file system is based on the assumption of a very low rate of write-write conflicts, which is valid in a multi-user setting dominated by read sharing[Kistler and Satyanarayanan, 1992].

In the introduction to their paper, Fallmyr et al. argues that based on experience the assumption of low write-write conflicts are not valid in a setting with one user controlling many machines. They further argue that in essence the sharing of data between one user with several machines, is the same as sharing data between several users [Fallmyr et al., 1995].

The Coda file system is one of the most used file system for portable machines. It uses the well known check-in check-out paradigm much used in distributed file systems.

The question about the check-in check-out paradigm is if it's good enough for mobile machines. In Coda the user checks out a user defined 'work-set' before going off-line, usually by hooking the portable machine to a high speed, low error rate LAN, and check the files back into the file system while returning.

Since mobile machines have less resources than portable machines they will not necessarily be able to even connect to a LAN by wire, thus needing their radio or infrared link for all accesses between the machine and the world. The datasets down-loaded by Coda tend to be large, in order to have any meaning at all. Since we use non-physical networking links and since using these devices are expensive for a mobile machine, the Coda way is not good enough.

2.3.2 Discussion

We have to discuss if the pessimistic or the optimistic approach is the best choice for a distributed diary.

Since the distributed diary is thought to run on mobile or at least portable computers it should be easy to see why the pessimistic approach is not good enough. Since a mobile or portable computer can be turned off and stay off forever, and still be part of the system, it is clear that a truly pessimistic approach will fail. In a system with mobile machines, the chance for at least one of them is disconnected is large at a given moment of time, making it impossible to guarantee global consistency at any moment.

We need a more optimistic way of maintaining consistency. The growth of mobile computing has led to the notion of *weak* consistency and *eventual* consistency [Downing et al., 1990, Golding, 1992]. This must be done to ensure that the system is able to do something without all members knowing, hoping they will eventual know and do something "intelligent" to fix conflicts which could appear in large systems.

The term "intelligent" is not intended to lead to an AI solution. The semantic of the data in a distributed diary includes the user of the application in a much stronger way than normal in distributed database systems. If a user has scheduled two meetings at the same time, there is no way for the program to know what to do without knowing the semantical meaning of the appointments. An example of this is if the user has an appointment with a friend for lunch one day. If the users boss wants a meeting, it would be impossible for the calendar system to know anything else but the fact that it has indeed has been an inconsistency and warn the user about it. The user knows what to choose, or he could do an action to remove the inconsistency by telling the friend to cancel the appointment.

In the general case the system should have a method to restore the correct value. The method used in database system often uses logs to try to rollback the database to a conflict free state. In the Coda file system files are just marked inconsistent and it is up to the user to repair the file.

Since most inconsistencies was inflicted due to a human action, and since humans are used to fix the inconsistency, that human should have some state information about what the contents of the file should look like. The human is part of the decision loop. See figure 2.3. We argue that this human interaction should be part of the application rather than to let the system decide on behalf of the user.

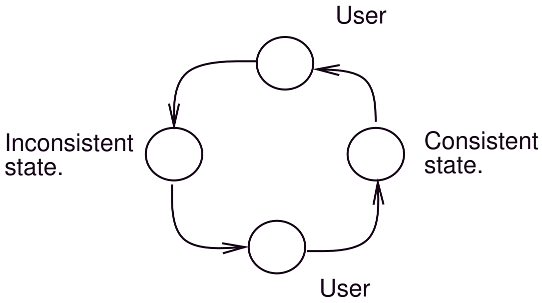


Figure 2.3: The decision loop

In Coda the user is only used to fix inconsistencies after the system has given up. In our system the user is more active, a natural consequence of having state information about his own system. The user is therefore more involved in this decision loop.

One way of showing how the user could be part of the decision loop is the "no entry delete problem". The problem is illustrated in figure 2.4. The user adds an appointment while not in contact with the diary server (machine A). Then he later uses an other machine (machine B), which doesn't know about the appointment to try to delete the appointment. This appointment exists neither in the server nor in the client, even though it will, after some time. What should be done in this situation? The normal application solution would be to say that you can't do this. The question is then; is the computer or the user right. Clearly the user know what he is doing and the system should support the ability to remove "non existing" entries (pseudo removing ability), and the ability to really remove the entry when it arrives at the server at last.

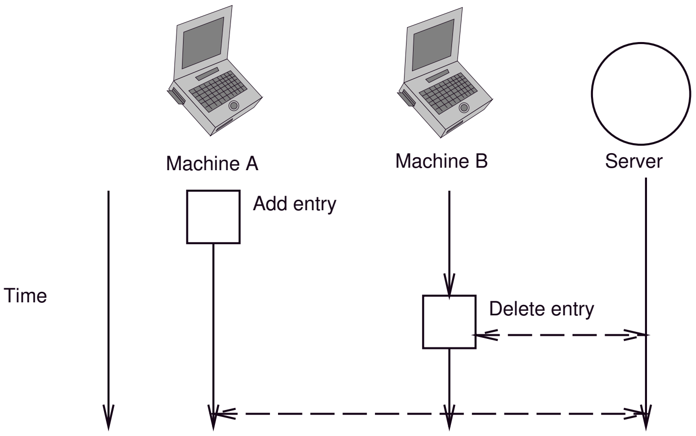


Figure 2.4: The no entry delete problem

2.4 Existing systems

There exist already a number of diary systems. Many of them are commercial, and difficult to examine in depth without buying them. We will try to look at some of the characteristics of the systems and compare these later with those our own system gives in the Design chapter.

2.4.1 Calendar

`Calendar` is a reminder service for one or more people using the same file system. `Calendar` reads calendar files in different users directories and uses mail to send results to the user. `Calendar` supports just month/day entries. `Calendar` is an application that follows the HP-UX operating system.

2.4.2 Ical

`Ical` provides a X interface for maintaining a calendar [Ghemawat, 1993]. A calendar is basically just a set of items. An item is either an appointment, or a notice. An appointment starts at a particular time of the day, and finishes at an other. A notice does not have any starting or ending time. Notices are useful for marking certain days as special. For example, a calendar may contain a notice for April 15th indicating that taxes are due.

You can include other calendars into your private calendar. This facility is mainly useful for allowing a group of people to share a common set of appointments. For example, members of a particular group might have a calendar that contains the birthdays for each member of the group. This calendar can be included in each group member's private calendar.

`Calendar` and `Ical` both uses the local file system to distribute calendar data, and is not usable for mobile applications.

2.4.3 Plan

`Plan` is a schedule planner based on X/Motif[1] [Driemeyer, 1996]. It displays a month calendar similar to `xcal`[2], but every day box is large enough to show appointments in small print. By pressing on a day box, the appointments for that day can be listed and edited.

The action being taken when a warn or alarm time is reached is programmable; by default a window pops up. In addition, a program can be executed, or a mail can be sent. Other methods of listing appointments (today, this week, next week, or a keyword search for regular expressions) are also available. `Plan` can be configured to display times in 12-hour or 24-hour formats, mmddyy and ddmmyy date formats, and can show either Monday or Sunday in the leftmost column. Three view modes are supported: one month, one year, and one week. The week view plots appointments as colored and labeled bars in a hour versus day chart, and allows other users' appointments to be included.

Version 1.5, adds a 365-day year overview and an IP server daemon that allows safe concurrent multiuser access with sequenced modification and automatic client updates. `Plan` is not designed to support mobile users.

2.4.4 Memo

`Memo` is an enterprise-wide business communication system with a client/server architecture that provides integrated services across multiple computer platforms [Verimation AB, 1995]. In addition to mail services, `Memo` also provides services that enable each user to coordinate activities, share information and participate in structured business processes.

The calendar module provides the `Memo` user with a personal diary for reserving time and making reminders. `Memo` is also a true company calendar that

---

[1] *X Window System* is a registered trademark belonging to *Massachusetts Institute of Technology (MIT). OSF/Motif* is a registered trademark belonging to the*Open Software Foundation(OSF)*

[2] Xcal is supported on most UNIX systems

enable meeting scheduling with both `Memo` users and other people. Resources such as meeting rooms can be booked in the same process. Designated users can also be allowed access to other people's calendars. In the meeting scheduling process, `Memo` offers free-time-search and overlayed displays of available time.

2.4.5 Time Vision Network Scheduler (TVNS)

`TVNS` is a product from CE software [CE Software Inc, 1995]. `TVNS` coordinates calendars for small to big organizations. It can both be used as a personal time manager, and as a group scheduling application. The system uses email for communication between the servers, thus making it unnecessary to use any special communication, due to the store and forward property of the e-mail system. In other words, a user makes an appointment with an other user by an automatic dispatch of an E-mail message detailing the event.

The `TVNS` system also supports off-line group scheduling, by down-loading individual and group calendars, and automatically synchronize the calendar data upon re-connecting with the network.

`TVNS` is supported only for Dos and MS-Windows.

2.4.6 CommonTime

`CommonTime` is a Lotus Notes application which schedules and manages events between office based and remote users [CommonTime Ltd, 1996].

`CommonTime` is a work-group calendaring and scheduling system which combines the graphical presentation features of a personal organizer with the powerful communications infrastructure of Lotus Notes.

Lotus Notes has features such as doclinks, attachments, encryption and replication, are all fully supported in `CommonTime`, providing tight integration with an existing Lotus Notes environment.

The principal strength of `CommonTime` is its ability to schedule and manage events involving groups of people, whether office based employees or remote users. Availability of personnel is identified through the Work-group Viewer which highlights activity types using corporately defined color categories. Various levels of access to users' calendars can be granted through Lotus Notes Access Control settings and encryption. Events can be reviewed, amended and rescheduled as required.

Because `CommonTime` is built on standard Notes databases, it has the possibility to integrate with existing and future Notes applications. Developers are able to create powerful applications that can read and update calendars, track time allocated by categories and more.

Non `CommonTime` Notes users can fully participate in the scheduling process by replying to mail invitations sent out by `CommonTime`. Even non Notes users can receive invitations as long as an appropriate mail gateway is in place.

Principal features of `CommonTime` include:

- Organizes & Tracks Work-group Activity
- Support for Mobile Users via Notes Replication

- Built in Security
- Integration with other Notes Applications and Gateways.

2.4.7 Lotus Organizer97

`Lotus Organizer 97` exploits Windows 95 and Windows NT and offers new Calendar and Notepad functionality, TAPI support and other time management features that are build upon Organizer's user interface.

`Lotus Organizer 97` is one of the industry's popular personal information and time manager for scheduling appointments, tracking to-dos, maintaining address lists, managing contacts and more. Organizer's multi-access, multi-user database format allows multiple users to share the same file for group calendaring, contact and task management. Organizer's daybook-style user interface appeals to individuals from all types of organizations regardless of size, from home offices to large enterprises.

Organizer uses a check-in check-out system for handling off-line clients. New entries added while off-line will be merged with the original Organizer file when returning to the LAN.

2.4.8 Schedule+

Microsoft `Schedule+` supports the ordinary calendar functionality as the other systems do [Microsoft, 1996]. It also includes features like "To Do" lists which track projects, work assignments, and deadlines using a comprehensive task manager, and the "Contact Manager" which easily store and access names, phone numbers, addresses, and much more.

`Schedule+` also has portable solutions. One down-load "To Do" lists, appointments, and phone numbers to the laptop PCs and PDAs and use the system as normal.

`Schedule+` also support group scheduling. Setting up meetings are easy using the "Meeting Wizard". Answer a few questions and it takes care of everything from suggesting an available time to preparing a pre-addressed meeting request. Information sharing is also supported. It is possible to attach documents to meeting requests so everyone can be prepared before the meeting.

2.4.9 Summary of characteristics

We here present a summary of the characteristics. The characteristics are:

1. Single/Multiuser.
2. How are data shared.
3. Off-line operation.
4. How is off-line operation maintained.

	# users	Data sharing	Off-line	How, off-line
`Calendar`	Multiuser	File-sharing	None	None
`Ical`	Multiuser	File-sharing	None	None
`Plan`	Multiuser	Client/Server	None	None
`Memo`	Multiuser	Client/Server	None	None
`Time Vision`	Multiuser	E-Mail	Supported	Down-load files
`CommonTime`	Multiuser	Client/Server	Supported	Down-load files
`Organizer97`	Multiuser	File-sharing	Supported	Merging files
`Schedule+`	Multiuser	Client/Server	Supported	Down-load files

Table 2.2: Characteristics

2.4.10 Lotus Notes

We will use some space to describe one of the best known application in the groupware world. Lotus has in their version 4.5 included a strong diary support [Louts Corporation, 1996]. Group scheduling and resource management are some of the features offered. Unfortunately is Notes an application which only supports portable machines with replication of files before going off-line. This is the Coda approach yet again [Satyanarayanan et al., 1993].

Other factors with reduces Notes' usability for mobile machines are the amount of resources it needs from the computer. Notes need a minimum of disk space and computational power which exceed most of the mobile machine's capabilities.

2.4.11 The Bayou project

The Bayou project at Xerox's Palo Alto Research Center tries to make a platform for replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications [Demers et al., 1995].

The system is intended to run in a mobile computing environment that includes machines with less than ideal network connectivity, ranging from wireless communication devices to using floppy disks as a communication device.

One of their goals is to support mobile machines with limited resources by using a flexible client-server architecture, where mobile machines both act as clients and servers.

Weakly consistent replication is used to increase availability for reads and writes. This means that a user is able to read and write to any copy of the database. They further try to reach eventual consistency while minimizing assumptions about communication characteristics and by having system support for detection of update conflicts, which issue dependency checks for every write order.

Since consitency conflicts often are application-specific the Bayou project uses an application-specific resolution to update conflicts, or semantical conflicts. They further permits applications to choose an appropriate point in the consistency/availability trade-off.

The first Bayou application made, has been a meeting room calendar manager and scheduler, which have been tested against a rudimentary server.

2.5 Evaluation of the existing systems

We will in this section try to evaluate the qualities of the existing systems. We will focus on mobility and consistency aspects.

2.5.1 Weaknesses in the existing systems

We have omitted a discussion of the system's weaknesses in their own subsection because they all have almost the same weaknesses. We have collected most of the common weaknesses in this subsection.

Most of the existing systems described have some weaknesses when dealing with mobile computers. This is mainly because most of the systems are primary workstation systems made for big enterprises, with only extra features to support portable machines. The solutions used are often influenced by this. Many of the systems uses down-loading of files before going off-line, hoping that enough files are down-loaded. This is similar to the method used in the Coda file system. While this is good for machines without any mobile communication possibilities, it is far too restrictive for mobile machines, which are able to communicate while moving.

The lack of mobility support is the main reason not to use these systems. By mobility support we also mean that the applications do not support resource-weak machines. Down-loading of files are expensive in terms of secondary storage which normally are scarce on PDAs. The need for communication must also be considered when one want to have a mobile distributed application.

Further, the systems need a way of store while off-line and forward information when re-connecting with the network. This could be done by construction a system where the application itself does this, or one could using existing systems, like e-mail. A further discussion of this can be read in the Design chapter.

2.5.2 Strengths in the existing systems

Most of the system described have strong groupware functionality. This is especially true for the commercial applications. These systems efficiently uses other persons diaries to schedule meetings, and along with resource allocation information to further improve the possibility to arrange meetings efficiently.

Often these diary application is part of an big system, which is increasingly important in the business world. The systems integrates the the diary with e-mail services and web services for example, making the diary part of a whole solution.

Most systems are made on top of a distributed database. Many of the systems offer the possibility to take these databases with them when moving around the world, and update these databases by i.e. modem whenever possible.

This is often enough for these applications, even if the user manually has to tell the system to replicate the database onto his portable computer.

Chapter 3

Requirement Specification

> *RULE OF ACCURACY:*
>
> *When working toward the solution of a problem it always helps you to know the answer.*

We will in this chapter look at the requirements for the application. First we look at the system model before looking at the data flow and the functionality requirements. These requirements are based on the Theory chapter.

3.1 System Model

We are going to look at the possibility to share appointments between one or more persons, or between one person with many machines, or a combination of the two. Based on the services already offered in `Plan`, we will look at the new requirements to the system in regard to consistency and communication problems. The choice of `Plan` for the GUI for our application is just because we have personal knowledge of `Plan` and know its functionality, and because it's a public application free to use.

The system model illustrated in figure 3.1 shows an easy model of how the system is constructed. The system consists of three parts, the File Repository (FR) server, the Diary Server (DS) and the clients. The File Repository handles things like replication and authentication, making the system more available and fault-tolerant. The set of FR servers also deals with consistency between themselves. This is easier since the repositories are placed on machines connected to a high-bandwidth, low error-rate network.

The problem with the FR in a mobile context is that it's based on a "check in check out" system originally design for portable systems. The system is based on files. As explained before, mobile machines have few resources and therefore is a "check in check out" scheme based on files too expensive since appointment files tend to be large. This is especially true since the FR itself has no way of telling the client if changes have happened. This means that the client has to poll the FR, meaning that it has to transfer files to a local copy and examine if there have been changes. The fact that the client has to "refresh" itself with the FR every time it wants to know if it's consistent is a problem. Since we

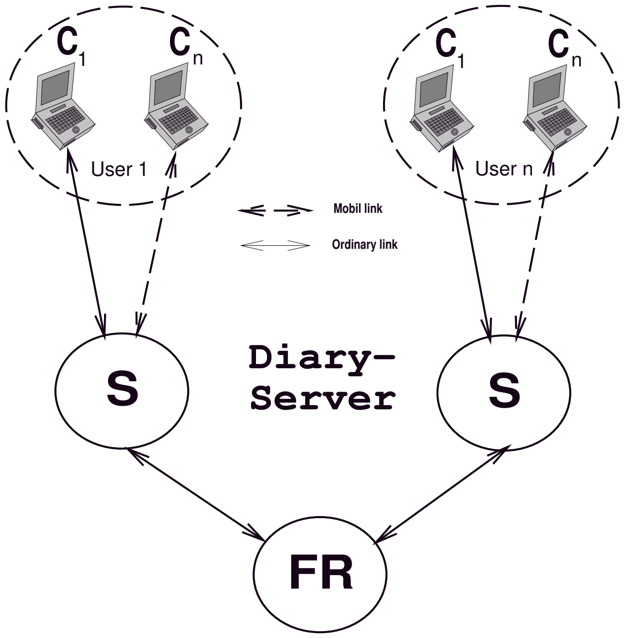


Figure 3.1: High-level system model

want to reduce communication, the user has the responsibility of "refreshing" the client.

However, the "check-in check-out" system is an easier way of dealing with consistency so there should be other ways than to rewrite the Fr. We use the Diary Server for this purpose. The Diary Server talks Udp with the clients and Frtp with the Fr. The Frtp is the communication protocol one uses to talk with the Fr. Further information could be read in [Stabell-Kulø, 1995].

3.1.1 Data flow

In Tdd all the data are generated by humans using the clients. But since this data often are globally interesting we will in this section describe the data flow between the clients and their diary servers, and between the diary servers and the Fr.

This communication is characterized by the fragile link between a mobile host and a diary server, and a good link between the diary server and the Fr. This heavily influence the characteristics of the data flow. First, none of the decisions taken in the client should assume that communication with the server must be successful. Secondly, the server should tolerate more than one equal request. This means that the communication between the client and server is *omnipotent.*

At startup the client will send a request to the server asking whether or not it is consistent with the server (B). The server must poll the Fr to check that they are consistent. This is done by sending a re-synchronize order to the diary server (D). The Fr tries to get the entry-file (E) and will return OK (H) or Not

OK (I) depending of the success of this operation. The diary server will then check the client checksum against the server checksum and answer the client by replying one of five possible answers. These are:

- ERROR if something unpredictable happened.
- OK if the client and server are consistent. The file appointment file is rewritten back into the Fr (O).
- TOO_FEW if the client has items the server doesn't know about (P).
- TOO_MANY if the server has items the client doesn't know about (Q).
- MODIFIED if some other client had modified entries by adding or deleting entries (R).

The client will now know whether or not it is consistent with the diary server. The data flow from the uncertain client to the certain client is illustrated in figure 3.2.

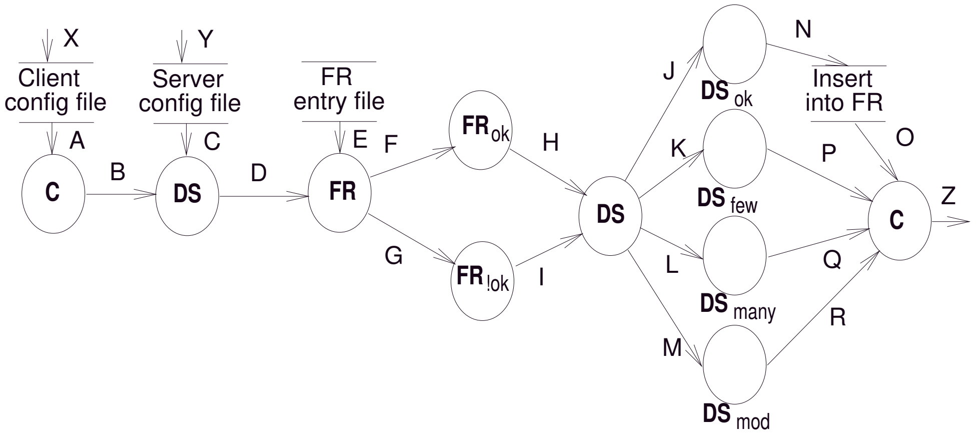


Figure 3.2: Data flow.

When the application is running, one goal is to reduce the communication between the client and the server to a minimum. This means that the only times the client contacts the server except the startup is when deleting or adding an appointment, upon receiving a hint from the server of an inconsistency or when the application is re-synchronizing itself.

Nearly the same data flow occurs for adding, deleting and quitting as for startup, and we have chosen not to show them here.

Since the diary server polls the Fr it will try to contact the clients whenever there is a change in the files in the file repository.

3.1.2 The client

The client software should run on different types of machines, ranging from small Pdas to large workstations. This means that the application should be small in size and demand little of the system in order to support small machines.

We shall make a library that is easy to build a user interface on top of. The user interface could range from fancy graphical user interfaces to command line orders.

3.1.3 The diary server

The DS is an abstraction on top of the FR. The DS has two purposes; keep the set of clients, the set of machines one person possesses, consistent, and write any changes on one client to the FR as soon as it is safe. The DS will further poll the FR to check for changes in the appointment file. If it discover changes in the FR or in any other client it will try to tell the clients by *hinting*. When the clients receive a hint from the DS they will ask for an update.

Hinting is done to reduce the clients need for polling the Diary Server. If no hints was implemented the client had to regularly poll the server in order to be consistent. This is too expensive for a mobile machine. Mark that hinting doesn't give any form for correctness. This follows from the nature of a *hint*.

3.1.4 Communication

Since we use the File Repository in this thesis, *files* are of great importance. The diary we use, `Plan`, uses files for representing its appointments. These files are used to generate a list of appointments. This list of appointments is distributed to the FR, but only when the appointments are considered safe. We consider it safe to distribute the appointment file to the FR when a client's appointment list is consistent with that of the diary server. We consider therefore the copy current in the FR to be the correct copy. It's the DS responsibility to try being consistent with the copy in the FR. This assumes that the FR orders insert_FR and extract_FR are executed successfully. See 4.4.3 for closer description of insert_FR and extract_FR.

As discussed previously, files are not a good item for transmitting between mobile clients. The system will therefore use two sets of communication protocols. The first is a file transfer between the FR and DS. This is assumed to run on a high bandwidth, low error rate LAN. The other one is an entry based transfer between a client and its DS. This communication assumes that we have low bandwidth and high error rate.

It must not be the concern of the system which type of communication that is being used. It should not see, nor care, if the system uses wire or wireless communication. It should also be transparent to the system in which way the messages are routed through the network. This is services offered by layers lower in the networking stack.

3.2 Functionality Requirements

The system must deal with all the requirements that a diary must full-fill, with the possibility to enter and be warned about events. These requirements are already met by the system `Plan`, and won't need any further description.

3.2.1 Mobile problems requirements

We mentioned four general main problems for mobile computers in the Theory chapter. We will look at these problems with regard to the application we shall build.

Networking and privacy

As mentioned in the Theory chapter these types of problems have not been dealt with in this thesis. We assume these problems are taken care of by other parts of the system.

Power management

The mobile machines are machines with few resources, a fact that not can be ignored. This means in practice that the machines our application runs on often will be turned off or only will offer reduced operation.

Many of these issues are machine dependent and could not be influenced much from an application programmers point of view.

However, the amount of disk use, the amount of networking, and the computational demands are of concern for a resource weak machine. We will in our application try to reduce these as much as possible. We will do this by:

- Only use disk as a backup medium if the network fails.
- Try to reduce the sending of data since sending is more expensive than receiving.
- Try to write the application as little and efficient as possible, so that both computational demands for CPU, primary and secondary storage are reduced as much as possible.

3.2.2 Consistency

One of the main requirements to the system is that it eventually will be consistent, meaning that as soon as an inconsistent machine gets back on line with the diary server, it will try to get into a consistent state. There might be conflicts in this process which must be solved.

The application should solve two types of inconsistencies. The first is the conflicts between one person's clients. This should be handled by the diary server. The second is the inconsistency which could occur between the diary server and the FR. This means that there is a difference in the appointment file currently in the FR and the appointment file locally in the diary server.

The application need to support a weak consistency model between the clients and the diary server. This model must have the possibility to store and forward entries, since the application should behave as normal as possible when off-line. The application must also warn the user when these things happen.

If the differences in the files are a consequence of another client changing and writing the changes back into the FR, we must assume that since this change was done by a human, with state information about the diary, he knew what he was doing. The client has to remove any appointments that have been removed from the file in the Fr and add any appointment that has been added in the Fr file. This describes the human interaction in the application. Then after doing this the client can do its changes before the changes are written back into the repository. The file must be locked in the repository while the writing of the file is done to avoid nasty errors.

However, if an other person has done these changes, it implies a different consistency model concerning the consistency between the Fr and the diary server, than we used between the clients and the diary server. This must be done since the user's state information is greatly reduced when dealing with other people. In other words we need a much more pessimistic model. The model must be much more pessimistic due to the fact that the demand for correctness is much greater when dealing with other persons.

Syntactical versus semantical consistency

There are two types of consistency models, syntactical and semantical. The difference in those two types are that the syntactical type consider just the difference in the appointment files, while the semantical type consider the meaning of the appointment files, like conflicts between two appointments during the same period of time. Another way of describing this is that the syntactical inconsistencies are inter-file inconsistencies, while the semantical inconsistencies are intra-file inconsistencies.

If the client can't decide what is right, it has to contact the actual user to warn him. The user can then do the necessary changes to his calendar. Further it must be the client's responsibility to seek a consistent state with the server. The server will be as passive as possible.

3.2.3 Off-line demands

The system must take steps when not able to reach the DS. The ordinary Tcp solution to this is to just give up. The system must:

- Save the uncertain data on a stable medium. (Back-up file)
- When the server is reachable, send the uncertain data. (This is known as store and forward.)
- Maintain as normal operation as possible both in the off-line system and the normal system.
- Warn the user about uncertainties in the system.

It must be possible for users to do meaningful interaction with the application even if one has off-line communication with the diary server. In this case it is the user which make the system inconsistent. The client is inconsistent with the server. (See appendix A.1.) The changes done in the client are saved on stable storage. It's the application's responsibility to notice the user about the uncertainty of such operations. This can be be done by marking uncertain appointments with a red color.

The application must also bring the file repository back into a consistent state with the diary server, by writing the consistent appointment file back to the file repository.

If looking at the characteristics of the systems described in the Theory chapter our application should:

- Support multiple users.
- Share data in a client/server style.
- Handle off-line operation well.
- Store and forward entries.

It must be possible for persons to edit files belonging to other users, and thus making changes to that user's appointment file. The file repository is inconsistent with the client. (See appendix A.2)

The worst thing that can happen is that both the other user and other persons make mutually exclusive appointments. (See appendix A.3.)

3.3 Testing

We will in this section try to discuss some of the demands we should have when testing the application.

Many of the mobile problems previously discussed will not be tested due to their nature. Some of them are assumed to work, like routing packages through the net. It's also hard to test or measure the amount of net traffic inflicted by the diary application, or look at the human/machine interaction due to lacking the actual hardware.

When testing our application we will concentrate on the consistency part of the problem.

However, testing an application for consistency is not easily done. Since the system is in one of two states (consistent or inconsistent) at all times there is no way of measuring *how* consistent the system is, or how good the application behaves.

In order to test the application we will use test cases and show that the application behaves as wanted during these cases. If we manage to get the correct behavior from the test cases, and the test cases are good and general enough, we can hopefully say that the system works. In other words we want to validate the system, and to see that the system performs due to the specifications. In order to design good test cases we need to describe some of the requirements that must be accomplished by the tests.

- One user.
  The system should eventually become consistent. This means that the client will become consistent with the diary server as soon as it contacts

the server, and that the diary server is consistent with the File Repository. If the user has more than one client, these clients should also be consistent with the system as soon as they contacts the diary server.

- Multiple users.
  The system should be eventually consistent when handling more than one user. This means that users can update each other's files in the FR in a consistent way.

- Warnings
  If a client make an entry which is not confirmed by the diary server because the server is unreachable, or messages are lost, the client must warn the user about this fact by changing the color of the entry.

Chapter 4

Design

> *WEINBERG'S LAW:*
>
> *If builders built buildings the way programmers write programs, the first woodpecker that came along would destroy civilization.*

We will in this chapter look at the design of the application. First we decide, based on the Theory chapter and the Requirements how the application should be made. Then we look closer at the construction of the application by first looking at the environment in which the application is made, then looking closer to the design of the client and the diary server. Then we look at the communication system between the client and the diary server and the diary server and the FR. Last we design the tests to validate that the system performs as expected from the Requirement Specification chapter.

4.1 Decisions

As stated in the Theory chapter there are two main problems in this thesis, mobile and consistency problems. In this section we present some of the decisions with an argument for why the decision has been made.

4.1.1 Mobile problems

When discussing mobile problems we mentioned three main problems. These problems was networking, power management and the interaction between the human and the computer.

Networking

To make make a distributed diary in a mobile environment globally accessible is very hard with the communication protocols of today. However, it's easier to have a mobile network within one building, i.e. by using radio communication. This makes things like routing very easy, both because radio is a broadcast medium and because one could use an Ethernet solution with bridges between the radio network and the Ethernet. Routing is handled automatically because of these medias broadcast abilities, and is therefore not considered much in the application.

Writing applications which uses ordinary networking should not be much different than writing code for mobile machines. The new networking problems should be shielded from the application by the networking layer, like IP. The new version of IP, IPv6 has support for mobility [Perkins et al., 1994].

The diary server and the FR are located on the same network and on stationary machines, they could even be on the same machine, so no extra concern has to be taken dealing with networking between these two servers. They are talking the FRTP protocol.

Power management

Power management has to be considered in most systems dealing with mobility since these systems mostly run on machines which are battery dependent. This results in machines which in periods have reduced operation and therefore can't be assumed to be on line at every moment.

Because of these limitations we have to consider the amount of resources used in our application. The resources we have to consider are:

- Computational demands
- Networking demands
- Demand for primary storage
- Demand for secondary storage

A diary application is by nature not an application which runs often on a machine. This, and the fact that most of the client code is for the user interface, leads to the argument that an application of this sort is not very demanding computationally.

Since the application is communicating with a diary server one has to make decisions about the amount of network traffic. The amount of sending and receiving must also be considered since the cost of these two are different. The amount of information sent or received must also be considered.

We tries to utilize the difference between sending and receiving by communicate as little as possible and by using a "hinting" message from the server.

The primary memory usage should not be great. As mentioned above the amount of code actually used for the primary diary application is little compared to the graphical user interface. By changing the GUI to a less resource demanding one by using for example curses or ordinary plain text we should be able to have a client which should be small enough to fit in any PDA.

The demand for secondary storage is also not big. The client has its entry files on secondary storage. The size of the files are dependent of the user's amount of appointments but should not be so big that it makes problems for a PDA system.

Since the resources on the server machines are much bigger than those on a mobile machines, no considerations about resources have been taken there.

Human/computer interaction

This is another issue which has not been considered much in depth. The reason for not looking at this issue is that the machines the application has been developed on, have much greater resources considering screens and keyboards than any PDA.

However, the application is written very loosely coupled to the graphical user interface and could be changed to match other types of human-computer I/O.

Another factor to this has been the introduction of Windows CE operating system. Until now applications for small machines like PDAs have been very machine dependent. This have influenced greatly the look of applications for these small machines. Windows CE will probably change this, and hopefully more machine independent applications for smaller machines, with a graphical user interface will emerge.

4.1.2 Consistency problems

These problems have been the main focus in this thesis and have therefore been considered most throughly. In the theory chapter we showed that optimistic consistency protocols was most suited for mobile environments.

Optimistic consistency control

Since the mobile machines eventually not will be on line with the rest of the computer network, one has to have a less strict consistency control, since these machines have a high degree of off-line operation. In this thesis we are implementing an eventual consistency control mechanism. This mechanism is made based on store and forward functionality as its basic tool.

This is a logical choice since the machine eventually will be on line again. The system guarantees that the appointment files will be consistent, inter-file consistent, but do not do anything to ensure intra-file consistency by looking at conflicts in time. These conflicts are up to the user to solve since he has some state information about the system. The user is part of the decision loop, and we have no intentions of forcing the user to follow our rules.

E-Mail versus UDP. There are some different ways of getting store and forward functionality in a system. Many systems, especially portable systems, uses ordinary e-mail to communicate entries from clients to servers, and from servers to servers.

There are several good reasons for this approach. First, the e-mail approach gives store and forward functionality. The e-mail subsystem handles the case when not being able to send, by delaying and retrying. This means that the application gets store and forward functionality “for free”.

Another important thing is that e-mail is trusted and supported by all organizations. They let e-mails through most of the fire-walls protecting net segments.

Furthermore, most small machines which have networking abilities will support the ability to read and write e-mails.

Given this nice properties why haven't we used this approach? The main reason is resources. Another is the amount of control the application has with the e-mail.

E-Mail is a rather large application stack that an application don't has much control of. While most large machines have the possibility to run "sendmail" as a background job, one can not guarantee that small PDAs have this possibility, e-mails are often just forwarded from a stationary machine. Further, a machine may not have network access for a long time, thus forcing sendmail to timeout and send the e-mail back to the sender. Another issue is whether sendmail is configured or not. The application has no way to tell. Further, the application has no way of knowing anything about the communication system. The e-mail system hides this for the system. The application has no control of how the e-mail system deals with errors that could appear, which further reduces the applications control with the communication which also reduces possibilities to utilize the scarce resources better. Sendmail doesn't have any time limits on its behavior, and guarantees only that the e-mail is delivered or that an error message is returned.

Another issue with e-mail is that of performance. An e-mail typically uses seconds in the best case from the sender to the receiver, but is greatly dependent on the load in the e-mail system. So the latency of an e-mail request and answer will in the average case be too great for an interactive application like a distributed diary.

Pessimistic consistency control

We have chosen to use pessimistic consistency control between the diary server and the FR. This is because of the consequences of conflicting writes in one person's FR. The diary server should be consistent with the FR at all times, which makes a pessimistic approach more useful. It's easier to enforce such a scheme on a LAN with tightly coupled machines.

The argument for this is that the user has state information about his clients, and could be less strict with enforcing consistency in his appointments, while when he contacts other people he doesn't have this state information and could therefore not assume as much as he could when just using his own clients.

Since there exists inconsistencies within the clients of one user, one has to be careful about what lies in the FR. We have decided that the best way is to put a copy of the entry file into FR whenever a client is consistent with the diary server. The only possible inconsistency is those entries existing in off-line clients.

Syntactical and semantical consistency control

In the previous section we discussed some of the chooses considering inter-file consistency. In this thesis we have not considered intra-file consistency much. We have not done much about conflicts in time and space like over-booking time or rooms. These problems have been looked much upon by group-ware systems like some of the systems presented in the Theory chapter.

Although many of those systems deals with such problems these problems often are impossible to solve without human interaction.

4.2 Environment

Ideally the application should work on all types of machines, from small laptops or PDAs to large workstations. These can be connected to a local area network (LAN) or by using infrared or other non-physical connections. The application is made in a library like style to make it easy to write interfaces to the user on top of it.

4.2.1 Developing environment

The system has been tested on two types of hardware, both systems running UNIX [1] based operating systems. The first machine is a HP[2] Apollo 700 series workstation running HP-UX version 09.03 and X11R5. The other is a TARGA laptop using Linux Slackware version 2.0.13, with both wireless and wired connection to the LAN. Even though the final system not necessarily will run on UNIX based systems, it should be enough to show that the system works on one system to argue that it will run on another. This makes it important to make the system as operating system independent as possible by using few system calls.

4.2.2 Communication and network

The system will use UDP and TCP over IP described in [Tanebaum, 1989]. These protocol stacks are supported by most operating systems. The HP workstation is connected to an Ethernet LAN. (10 Mbits/s), while the laptop has the possibility to use infrared light, radio wave or Ethernet to connect to the LAN. Thus the laptop may communicate wireless with the environment. The clients run on the laptop or the workstations, while the diary server runs on one of the workstation. The diary server communicates with the FR which also runs on a workstation.

The client uses UDP when communicating with the diary server, while the diary server communicates with the FR by FRTP which is an abstraction over TCP.

4.2.3 Programming tools

We have chosen to use "C" as the implementation language for our application [Kernigham and Ritchie, 1988]. This in addition to EMACS, MAKE, LINT and XDB make a good environment for implementing and debugging the application.

---

[1] UNIX is a registered trademark of AT&T.

[2] Hewlett Packard

The C language is historically strongly connected to the UNIX society. Thus it has been developed a lot of libraries which are very useful for developing an application. And in addition to that, the system used for a graphical demonstrator, `Plan`, is written in C which makes it practical to use C for the client library and diary server.

`Plan` is a very portable system which runs on most UNIX operating systems. With few OS to test the extensions and the server on, the only measure for portability is to write Ansi-C and keep the number of system calls low. The system will run in the workstation environment and on the portable machine.

4.3 Client/Server

We have already described the functionality of the system. Since the user of the system may have more than one computer and one or more of those are a mobile computer it's possible that the computers are located at different places.

We have chosen to use a solution where every user has one server, the diary server. This diary server is contacted whenever a change is done in one or more of the clients. Whenever the diary server gets a change from a client it will update the appointment file in the FR, if the client and the server is consistent. The diary server will also poll the FR regularly to check if some other persons have made changes in the appointment file. If they have the diary server updates its appointments. Then the diary server tries to hint the clients about the changes. Referring to [Svobodova, 1984] a server is a subsystem offering certain *services* to *clients*. See figure 4.1.

4.4 Design of the Server

The design of the system is very simple. The system is based on an ordinary client/server protocol where the client send requests to the server and gets response. The maintenance of a TCP connection is considered to bee too expensive for a moving computer with very varying bandwidth and little resources. Because of this we use simple UDP packets. The client will wait for the response from the server and timeout after a while if it doesn't get one, assuming that the server is unreachable. See figure 4.1.

4.4.1 Diary servers

The server has to know the number of clients in order to be able to remove an entry from its database. Otherwise it must hold its information forever waiting for new machines to arrive. Upon starting, the server will read a configuration file where all machines connecting to the server will be marked, along with the name of the owner of the machines. Whenever the owner buys a new machine or sells a machine the preference file must be updated and the server restarted.

An example of why the server needs to know about all the clients is when a client issues a delete order to the server. It is not enough to just delete the entry in the server and in the client, thus making the client and the server consistent.

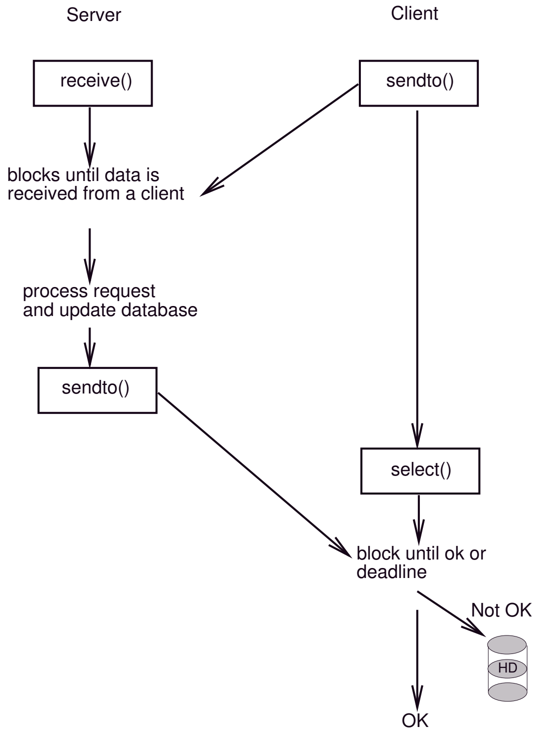


Figure 4.1: Design of the client-server interaction

This is because all other clients will be inconsistent to the server. The server need to keep the entry to be removed until all clients know about it.

An obvious consequence of this it the fact that one never can guarantee "global" consistency. The server will only be guaranteed consistent with the client currently connected. The server will further try to contact the clients to tell them to get any new data. This is better than letting the client poll the server to check for new information. This utilizes the fact that it's less expensive for a mobile machine to receive than to send.

4.4.2 The file repository

The diary server communicates with the File Repository Transfer Protocol (FRTP). Whenever a change in the appointments is propagated from a client to the diary server, the diary server updates the appointment file in the FR. FR is a nice medium to use when distributing a file like the appointment file, because FR handles things like replication and authentication. The diary server must thus be consistent with both the FR and the online client.

We will use the FR_API to make the communication easier for the client [Fallmyr, 1996].

4.4.3 Server primitives

The server maintains two lists for every client. These represent the certain and the uncertain events in the client. The server consists of a number of primitives to handle orders from the clients. These primitives are:

add_entry()

- Description
  Add an entry in the server's entry list belonging to to client. The add primitive also check if the other clients have the entry. If not the entry is also added to that entry's uncertain list, so that the entry can be added the next time that client contacts the diary server.
- Input
  The name of the user which we are adding an entry to.
  The entry to add.
- Reply
  OK is replied if the operation was successful, or an error message saying that the entry already exists in the server. This could occur when an reply from the server has been lost and the client reissue the add order. The other possible reply is the server error message.

find_entry()

- Description
  This function tries to locate an entry in an event list.
- Input
  The entry to be found and the list to look for it in.
- Reply
  True or False depending on the result of the search.

delete_entry()

- Description
  Removes an entry from the client's or another users entry list. If the delete order is for the client's user the delete primitive also check if the other clients have the entry. If they have, delete it and put the entry in the entry's uncertain list as a delete entry, if not put the entry in the entry's uncertain list as a delete entry.
- Input
  The name of the user which we are deleting from.
  The entry to be removed.
- Reply
  OK if successful or ERROR if not.

md5_list()

- Description
  This function uses the md5 algorithm described in subsection 4.5.1 to calculate the checksum of the entry list to the client. This function is used to be able to compare the client's checksum with the server's.
- Input
  The list to calculate the checksum of.
- Reply
  The checksum.

load_entry()

- Description
  This function loads an entry file from stable storage and adds the entries sorted on note and time into a linked list of entries. This is done for all clients.
- Input
  The entry list to load to.
- Reply
  OK if successful or ERROR if not.

save_entry()

- Description
  This function saves the whole database to stable storage. This function is only called when a client is consistent with the diary server.
- Input
  The entry list to save.
- Output
  OK if successful or ERROR if not.

insert_FR()

- Description
  This function is called when a client is consistent with the diary server. It saves the whole entry file in the FR, and is called after save_entry() is called.
- Input
  The name of the user so we can save the entry file in the right FR directory.
- Output
  OK if successful or ERROR if not.

extract_FR()

- Description
  This function is called when a the diary server tries to re-synchronize itself with the FR or when reading someone else's entry file.
- Input
  The name of the user.
- Output
  OK if successful or ERROR if not.

The number of primitives are few. We consider it better to have a small number of easy primitives, thus making complicated orders by issuing a number of small primitives, than to have to make separate implementations for every new order a system like this would need. It's further easier to write efficient and error free code when using small pieces of code, thus getting better control of the code. We argue that the argumentation used in hardware technology could be used software in development. (Compared this with the RISC/CISC discussion.)

4.5 How to Ensure Consistency

The best way would be to send all entries in the client to the server and check if all entries are present in the server. While this would work if the bandwidth between the client and the server is good and the client can contact the server at any time, it would be too expensive for a mobile machine. In the distributed diary application this would mean n requests packets and n answers packets where n is the number of entries in the client.

We are actually looking at a syntactical way of ensure consistency in the distributed diary. We are not in this section considering semantical consistency since we then have to look at the actual contents of the entries.

An other observation is that the messages from the client to the server tend to be larger than the reply messages since often the whole entry is sent in an order. The reply messages tend to be small since only an error code is returned.

4.5.1 Checksum

The system uses a checksum algorithm in addition to two serial numbers to check if a client has the same view of the world as the server. The checksum algorithm is the MD5 algorithm which produces a 128 bit checksum of an input [Rivest, 1992]. This algorithm is designed to make unique checksums for most inputs. This means that two different inputs should never give the same checksum. A consequence of this is that the server and the client must sort the entries the same way, in order to be able to compare the checksums.

The idea works like this: The client send a checksum of its database to the server along with two serial numbers, which represent the client's view of the number of entries in the client and server. If the checksum match the checksum in the database, two possible states are possible. The first is the one where everything is okay and both the client and the server has the same view of the world.

The second is when the server has some entries the client has not heard of. The server detects this by comparing the serial numbers, which the client sent. This happens when there are entries in the client's uncertain-list. Then the client must ask for those entries. Mark that the opposite situation could never happen. The checksums could not be equal while the client has more entries than the server. This is because the remove order works as an add order; one has to know which entry which is to be removed. See fig 4.2.

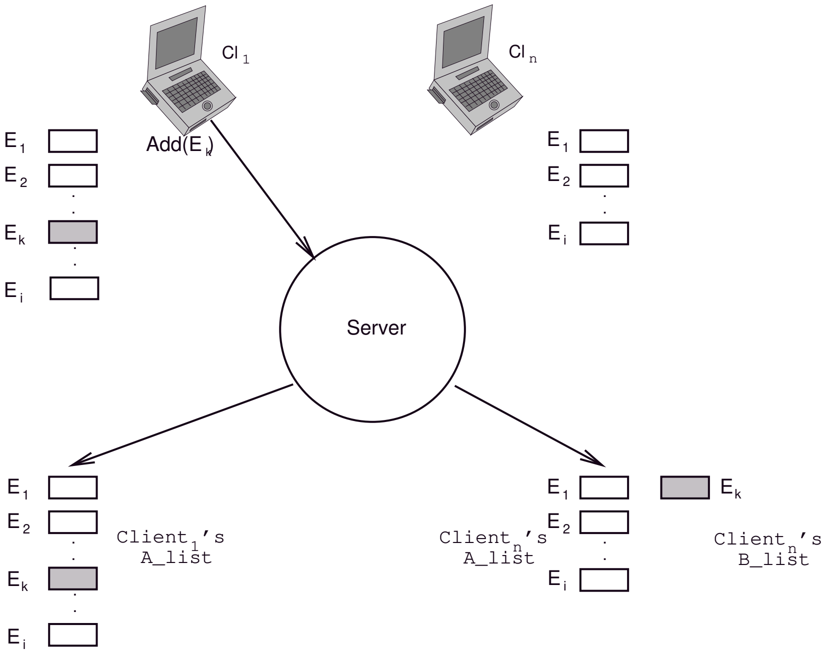


Figure 4.2: Adding and deleting entries in the server.

If the checksum does not equal it means that there has been one or more changes in one client. This could be changes in one entry or an add or delete order. The client will know which entries the server does not know about since in order to be inconsistent the add or delete order could not have reached the server, and would therefore be saved on stable storage in the client. The client will then try to send those entries to the server on a later occasion.

Mark that the client can not decide whether the request or the reply was lost. But since the system is designed in an omnipotent way, it does not matter, the client just reissue the order.

The checksums are only used for present and future entries, thus making the need for communication less.

Depending on the serial number and the checksum the client can then check which entries it hasn't got and ask the server for all entries that it doesn't know about. When the client gets the entry it will be stored in the client and a reply to the server sent. Then the server removes the entry from the uncertain list.

4.5.2 Time

The diary application is heavily dependent on *time*. The problem with this is that such anapplication has a forever growing database. Because of this we make the following assumption: only future (and present) appointments are globally interesting for the application, and thus making a somewhat lesser database. Old entries will be saved at the clients and was, most probably, but not guaranteed synchronized with the server and the other clients at the actual time. The only difference between the assumption and the general case is that one work with a subset of all entries in the database, thus make the system less demanding, which was one of the demands to the system.

By using this assumption we need only to make the future appointments guaranteed synchronized. This is done by getting the "now" time and check this value against the entries in the client, and only send the entries that is due in the future. This includes repeating entries.

The problem about time has been discussed in many papers dealing with timing in distributed systems [Kopetz and Veríssimo, 1993, Veríssimo, 1993]. We will make some assumptions about the system so that this problems will not be so serious as they could have been.

- The time of the computer is correctly set, so that the "now" time is accurate enough compared to the real local time.
- There exists a limit of how long a message could survive in the net. This assumption makes it possible to assume that a message was lost.
- The clients operates in the same time zone such that the local time is the same on all clients and servers. In practice it would be easy to include the local time with the resynchronized message to ensure this.

These assumptions hold since the granularity of the time of a calendar is not measured in milliseconds, so that the worst case possibility is not very likely to happen. The worst case is when a clients send its checksum just before any entry is due, thus including the entry in the checksum at the client but not at the server, thus making the client seem inconsistent with the server without being so.

4.5.3 What to do with semantical conflicts

If a new entry conflicts with an old entry by overlapping the time or duration of an entry there is a semantically conflict in the system. Since the system does not support any form of priority the user will be allowed to over-book a time. The only thing the system will do is to warn the user. This could be done by using color or by a confirm action. Since the user has "state" information about the database the database should allow the user to over-book.

The server serves just as a simple database for the the clients and it doesn't has any semantic information about the entries in the database. This means that checks for conflicts are done in the client. This is very important due to all the different clients that possibly will be served by the server. It's also important when dealing with the users role in the decision loop. It's very much up to the user to define a conflict, thus making it in the server would not necessary be good.

4.6 Design of the Client

One of the main design goals is to make as small changes to the already existing programs as possible, by writing a library with all the functionality necessary. This library makes it possible to use different types of client programs and thus different types of graphical interfaces for the client which is very desirable because of the varying viewing devices used. (Ranging from small PDA screens to large workstation monitors.)

It should also be possible to do command line interaction with the server, saying something like "schedule today" to see todays events or "add (event)" to add an event to the server's database. This depends on the client. Different clients will have different ways of representing entries. It should be easy to redefine the entry structure for the server.

This design of the system is good considering the small amount of resources on a small portable computer like a PDA. The small amount of resources must also be considered when dealing with the network. We will try to send as little as possible over the network.

The main design goal of the client is to make it easy to make different clients. This is important because it should be possible to have different clients running on different types of machines.

This means that it should be easy to add the code to any client. This is done by making a library to compile into the client. Whenever the client adds or removes an entry in its main list, the server is notified through some primitives offered by this library.

Whenever the client starts up it will send a resynchronized message to the server asking for how many entries it know about. If the server doesn't know about any entry (i.e. it has just started up) the whole database is sent to the server. If the database exists the client will check if there exist any backup file and send it. Then the re-synchronize message is sent again. If there are inconsistencies at this point it could only mean that the server has got (or deleted) one or more entries from other servers or clients. The client will then ask the server for information about the entries the client does not know about. The server has this information in the uncertain-list belonging to the client, as either add or delete orders.

If the server is assumed to be unreachable the system will save the data which was not received at the server to stable storage. Whenever the client manage to contact the server a consistency check will be run to check if any not processed requests are waiting (It's also possible to manually give the client order to send a consistency check.) The client will then try do send it one more time. This assumes that orders from the client are omnipotent. (The entry could have reached the server without the reply reaching the client.) Since entries are sent one at a time, it's possible to know exactly which entry did not reach the server, and mark it with i.e. a color. The way of warning the user is the problem of the client application.

4.6.1 Client primitives

The client needs some primitives in order to be able to communicate with the server. These primitives are:

marshall_entry() and unmarshall_entry()

- Description
  These two functions deals with marshalling information like pointers to and from a message. It builds the actual message to send to the server.
- Input
  An order and an entry.
- Reply
  The marshalled or unmarshalled message.

md5_list()

- Description
  Same as in the server. See above.
- Input
  The client's entry list.
- Reply
  The md5 checksum.

backup_entry() and get_backup()

- Description
  The backup_entry() saves entries on stable storage whenever an add or delete order fails, the entry is also put into the uncertain list so that it's possible to warn the user. The get_backup() does the opposite, it reads the entries and sends them to the server. Entries are not removed from the backup file before they are guaranteed to be in the server. The entry is also taken out of the uncertain-list.
- Input
  The entry to backup. (For backup_entry())
  The backuplist. (For get_backup())
- Reply
  None

send_resync()

- Description
  The client calculates the checksum with the md5_list function of all present and future entries (time $> NOW$) and send these to the server along with the client's serial numbers.
- Input
  None
- Reply
  An error message indicating the state of the consistency.

send_entry()

- Description
  Send an entry to the server. If the server does not respond, then the backup_entry function which backs the entry up is called. (If the entry sent was not already a backup entry).
- Input
  A message consisting of the entry and the order.
- Reply
  An error message indicating the success of the send_entry() function.

entry_add() and entries_add()

- Description
  Entry_add() adds an entry to the entry list in a sorted way and calls send_entry() to send the entry to the server. The entries_add() uses entry_add() to send all entries to the server.
- Input
  The entry to add.
- Reply
  An error message indicating the success of the entry_add() function.

get_user()

- Description
  This function asks the server to load the entry file for the user mentioned into its otherlist, so that one could fetch the entries by issuing a get_server_entries() order.
- Input
  The name of the user to get.
- Reply
  OK or ERROR depending on the success in the server.

get_server_entries()

- Description
  This function contacts the server and gets all the entries in the uncertain-list in the server belonging to the client.
- Input
  The name of the user.
- Reply
  The entries from the server.

find_entry()

- Description
  This function tries to locate an entry in an event list.
- Input
  The entry to be found and the list to look for it in.
- Reply
  True or False depending on the result of the search.

send_entry_delete

- Description
  Send an entry to be deleted to the server. If the order is unsuccessful only back up the entry if it's not already backed up.
- Input
  A message consisting of the entry.
- Reply
  An error message which represents the success of the send_entry_delete() order.

entry_delete

- Description
  Update the serial numbers and the client's main entry list. Then call send_delete_entry().
- Input
  The entry to be deleted.
- Reply
  None

When the client adds an event into its main list an event is added in the server's main list by calling entry_add(event) in the client. The entry_add is a library routine which sends the events to the server. This function also updates the library's own list of events. The library must do so because the checksum algorithm used to check the consistency of the databases in the client and server must be sorted the same way. The server will reply with an OK message if everything executed well, or an error code if something else happen. This could happen if the entry already existed in the server, or if the server failed to save the entry.

If a timeout limit is reached before the server could respond, the entry is assumed lost. The entry must be saved at stable storage until the server is reachable once more. The function backup_entry does this operation.

The same is done when an event is removed from the client's main list. Then the library routine entry_delete(event) is used. The library list is updated and the entry will be sent to the server. The server replies with an error message. These are the same as for entry_add.

4.7 Plan and the Graphical User Interface

We have earlier described why we use `Plan`. We will in this section look in more detail on the changes which must be done in `Plan` to give us the needed functionality. Since many of these changes are visual changes we will also look at the Graphical User Interface (GUI).

We have kept the number of changes to the `Plan` as small as possible for reasons given elsewhere. However, changes have to be made to support some of the needed functionality, such as warning the user about inconsistencies. A screen-dump of the GUI can be seen in figure 4.3.

The color of the entries shown in the figure changes if the entry is uncertain, and will thereby warn the user about an inconsistency.

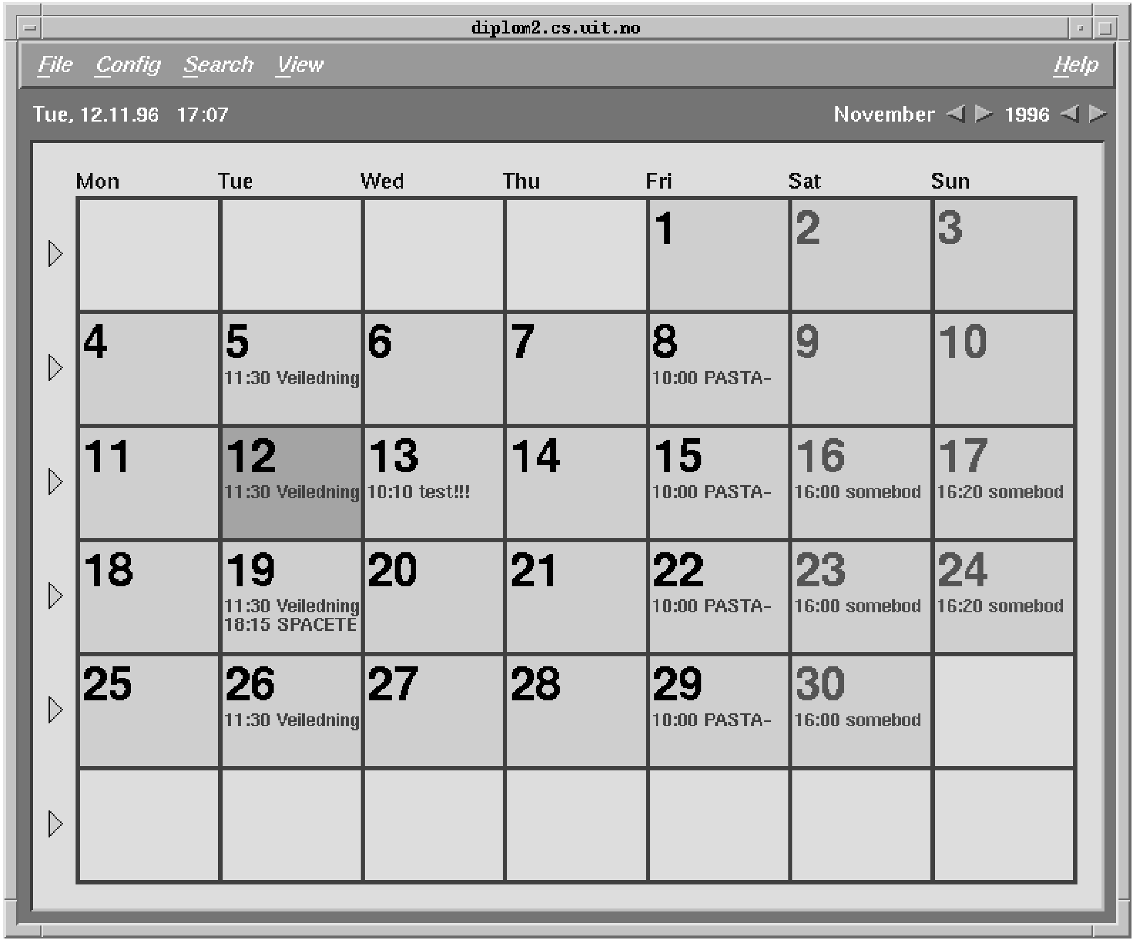
The figure of the `Plan` GUI clearly shows one great problem we have when dealing with mobile machines. The figure is smaller in size than the actual GUI. It will never fit a screen of a PDA. Another problem with this GUI is that in order to select an entry one has to point on it and push the mouse button. This means that with this interface it's impossible to have a "pseudo delete" function, without adding functionality in the client.

4.8 Possible Improvements

In order to further improve the behavior of the system, by utilizing the resources in a better way, one could use some of the mechanisms used in networking theory, like piggy-backing information in reply messages or by using compression to reduce the network load.

One could also use some batching mechanisms to better utilize the bandwidth whenever possible. One could for example batch together add orders with a re-synchronize order, instead of sending them as two separate messages.

FR is a general purpose system that could be optimized for a system like a distributed diary. One could also bind the FR closer to the diary server for performance reasons. This could be done by integrating the FR front end with the Diary Server.



Mon	Tue	Wed	Thu	Fri	Sat	Sun
				1	2	3
4	5 11:30 Veiledning	6	7	8 10:00 PASTA-	9	10
11	12 11:30 Veiledning	13 10:10 test!!!	14	15 10:00 PASTA-	16 16:00 somebod	17 16:20 somebod
18	19 11:30 Veiledning 18:15 SPACETE	20	21	22 10:00 PASTA-	23 16:00 somebod	24 16:20 somebod
25	26 11:30 Veiledning	27	28	29 10:00 PASTA-	30 16:00 somebod	

Figure 4.3: Screen-shot of `Plan`

4.9 Design of the Tests

We will in this section design the tests based on the demands we presented in the Requirement Specification chapter. We will do this by designing test cases. We will start by looking at the most simple cases and move toward a more real setting.

The tests we are going to perform is a *validation* test, where the goal is to validate the application's specification [Pressman, 1994]. Since this black box test is used to demonstrate conformity with the requirements, the test cases should be designed to show this. We use this scheme since only two possible outcomes are possible, either the application conform to the specification and is accepted, or a deviation from the specification is found.

4.9.1 One user

We start by looking at one user. We will have four different test cases for one user.

Test case 1

Here we have one user with one client. The user adds and removes entries from the diary while he has good communication with the diary-server. (The client is hooked to the LAN.)

Test case 2

In this case we still have one user and one client, but this time we simulate a communication situation with lost messages. The user should still add and remove entries but *eventually* the client and the server should be consistent. The amount of messages which are lost can be modified. Whenever an entry is uncertain, this will be shown by a red color.

Test case 3

In this test we hold on to our one user, but increases the number of clients to three. The point with this test is to show that actions on one machine eventually will reach the other. The different clients will add and remove entries and we will show that this does not destroy any consistency demands. The clients are on the same LAN so we will not lose packages.

Test case 4

This is the worst test case possible for an one user system. The user still has three machines, as in test case 3, but this time we have possibilities of loosing messages. The clients will add and remove entries and will be warned about uncertain entries. We will show that the system *eventually* will be consistent.

4.9.2 Many users

We now advance to more than one user. We will show that it's possible to have interaction between users. We will show this in two test cases. These tests involve the FR much more than the previous four tests.

Test case 5

We start by the easiest possible setting, just two users on a LAN. These two users will first get and then write entries in each other's diary. We will show that these entries will show up in the diary to the other person.

Test case 6

This is the most real world test. We will have many users, each with more than one machine, sending entries both locally between their own clients and globally between the users. The clients will run on bad communication links and will lose packages. We will show that the system eventually become consistent, and that the user is warned by a red color of entries which are not consistent.

For all the test cases consistency means syntactical consistency meaning that the checksum of the client entry database is identical with the server entry database.

Chapter 5

Implementation

> *FRANKLIN'S RULE:*
>
> *Blessed is the end user who expects nothing, for he/she will not be disappointed.*

In this chapter we look at the implementation of the distributed diary. First we look at the implementation of the diary server, then on the communication system and then on the implementation of the client.

5.1 Implementation of the Diary Server

The server is implemented as one process. The server starts by initializing its main data structures with the name of the machines the server is responsible for controlling. It then contacts the FR to get the last agreed upon version of the entry file. Then it creates an UDP socket which it blocks on, waiting for orders to be received.

The main data structures in the diary server is the client array, which is the array consisting of all clients the user has and the client's lists, which are lists of entries sorted by name and time.

5.1.1 The clientlist

```
typedef struct clients {
  char          *client_name;
  int           number;
  list_t        *A_list;
  list_t        *B_list;
} client_t;
```

The entries in the `client_t` are mostly self explaining. The clientname is the name of the machine, like "diplom0.cs.UiT.No", and the number is the number of clients. The lists are described in detail in the next section.

5.1.2 The list

The list has information about how many entries there are in the server. This information is stored in the `se_entries`. A first-pointer to the first entry is also necessary. The `se_entries` is used when comparing views with the client for determining inconsistencies.

```
typedef struct mainlist_list {
  int          se_entries;
  entry_l_t    *first;
} list_t;
```

The list consists of entries. The actual entry is based on `Plan`'s entry since `Plan` is used as the GUI for our application. This struct is the main place where there must be changes when changing client software. Most of the variables are of little or none interest for the server, but in order to send an entry from one client to another all client specific information are needed.

```
typedef struct entr{                      /* one day-schedule entry */
  long          nextrep;                  /* next repeating entry in chain or -1*/
  char          user[MAX_SIZE];           /* user name, 0 if from own .dayplan */
  char          message[MAX_SIZE*16];     /* message text, 0 if none */
  char          script[MAX_SIZE*16];      /* shell script, 0 if none */
  char          note[MAX_SIZE];           /* explicit short note, 0 if none */
  time_t        time;                     /* time and date in seconds */
  time_t        length;                   /* length in seconds */
  time_t        early_warn;               /* warn this many seconds in advance */
  time_t        late_warn;                /* and this many seconds, too */
  time_t        except[NEXC];             /* don't trigger on these dates */
  time_t        rep_every;                /* repeat every N days (in seconds) */
  time_t        rep_last;                 /* stop repeating on this date (secs) */
  long          rep_weekdays;             /* weekday bitmap, bit 0=mon..6=sun */
                                          /* bit 8=1st..12=5th, 13=last week */
  long          rep_days;                 /* day bitmap, bit 0=last, 1..31=days */
  BOOL          rep_yearly;               /* repeat every year on the same date */
  BOOL          suspended;                /* not active, green radio button off */
  BOOL          private;                  /* not accessible for others */
  BOOL          noalarm;                  /* no alarm, just warnings if enabled */
  BOOL          notime;                   /* no time is printed in day lists */
  BOOL          hide_in_m;                /* don't show in month view */
  BOOL          hide_in_y;                /* don't show in year view */
  BOOL          hide_in_w;                /* don't show in week view */
  char          acolor;                   /* text col:  0=std, 1..8=HBLACK..  */
  char          triggered;                /* 0=none, 1=early, 2=late, 3=time */
} entry_t;
```

This 2226 bytes entry is part of the `entry_l_t`. This is mainly a list of entries. But there is one important detail there. Since we used an uncertain list of entries to be added or deleted, and this list is an ordinary list of entries we need a way of telling whether or not the entry should be removed or not. Because of this we also have a boolean variable "delete" in the `entry_l_t`.

5.1.3 Processing of the orders

Eventually there will come an order to the server. The server checks that the message came from one of the clients before doing anything with the message.

Then the server issues a `switch` statement depending on which type of order the message represented. Depending on the type the server does one of these things: *resynchronize*, *add*, *delete*, *get_entries* or *get_user*. The order is then processed by the server primitives and an answer is returned to the client.

Since the consistency model is the most important in this thesis we will discuss the re-synchronize order in more detail. To check that a client is synchronized with the server one has to compare the client checksum with the server checksum. Mark that since we only guarantee that the client currently connected to the server could be consistent, we only check the client checksum with the server checksum belonging to that client.

Then the numbers sent by the client is checked with the numbers of the server. There are five possible outcomes. The first is the one where the checksums equals along with the serial numbers. If this is the fact the client and the server is consistent. An OK reply is sent to the client.

The second and third happen when the client has fewer entries than the server. We have two possible outcomes here because the checksums could be equal.

In the fourth it is not possible to have equal checksums. This is the outcome where the client has more entries than the server.

In the fifth possibility the checksums are not equal, but the number of entries in the server and the client equals. This means that one or more of the entries have been modified. This would normally not happen since it does not exists a modify command other than a combination of a delete and add order.

The reply generated from this represent one of these five possibilities. If the reply was an OK message we do know that the client currently in contact with the server is consistent with the diary server. Further, we also know that the client has got all entries that all other clients have given, since the client's uncertain list must be empty, and the B_list in the server also is empty. (Otherwise re-synchronize check would have returned a message saying that the client had too few entries.) Because we know this we can take steps to get the guaranteed consistent entry list into the FR. This is done by writing the entry list to file and making a "insert" order to the FR.

Since the "hinting" scheme described in the design chapter only is an improvement which do not inflict on the correctness or availability of the system, this has not been implemented due to lack of time.

5.2 Communication

As a programming interface toward the UDP/IP the *Berkeley sockets* has been used. This interface provides a set of system calls used when handling sockets. [Stevens, 1990]

At the top of the socket programming interface we have made a library handling much of the complexity with sending and receiving UDP packets to and from the network. This library is a reduced version of Arne Helme's *socklib*, a library which makes the interface to the networking system calls easy to use.

Since the application is supposed to operate on small movable machines with very variable communication this must be simulated when testing the system. To simulate this a variable could be set to represent the chance of loosing a message. A random number generator is used to generate random numbers which are checked against the variable. Then the message is sent or "lost".

5.3 Implementation of the Client

The client uses the same representation of an entry list as the server. This is because the MD5 checksum algorithm needs to have exactly the same list to take the checksum of. This list works like a copy of `Plan`'s way of representing entries.

5.3.1 The Plan graphical interface

In this thesis `Plan` has been used as a graphical interface to the server. `Plan` is a stand alone free software product available on different FTP sites.

To use `Plan` has both been a blessing and a curse. First, by using `Plan` we quickly got an application which could be tested and which worked, but much time was used to find out how things were done in `Plan`. These things have greatly influenced how things have been represented.

The changes done are as low level as possible. This means that most of the changes to the client are in the primitives used by `Plan`. Every time an add_event or delete_event is called in `Plan`, a call to the appropriate library function is called. Nevertheless modifications had to be done in the actual `Plan` files.

One feature we exploited in the `Plan` system is that `Plan` uses a deamon to warn the user whenever something happen. This deamon is re-synchronized whenever there has been an important change in the `Plan`. The resynchronize order is sent every time the client tries to re-synchronize its daemon, or every time "*Resynchronize*" is selected from the menu.

The list of the sorted events, the serverlist, is made global so that as few function arguments should be changed.

There has also been a change in the actual drawing of the calendar because we change color of uncertain events. We use a *red* color to emphasize uncertain entries in the example application.

Chapter 6

Test Results

> *LAW OF CONTINUITY:*
> *Experiments should be reproducible. They should all fail in the same way.*

We will in this chapter, based on the test cases described in the Design chapter test the implemented application. We will first do the tests and then discuss the results.

6.1 Tests

Testing consistency in a distributed system is difficult. Many things can fail. This is the reason for us to set up a set of test cases which handle most of the situations the diary could get into, and thereby validate the requirements. The tests have been designed to mainly validate the consistency demands.

Test case 1

We have the following setting: The diary server and the FR runs on the same machine (diplom0.cs.UiT.No) and the only user client runs on another machine (diplom2.cs.UiT.No). See fig 6.1.

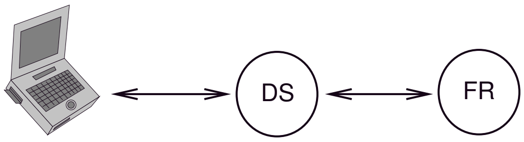


Figure 6.1: Test case 1

When the diary server starts up it tries to contact the FR to get the latest consistent entries. If there are no entries in the FR the server will just begin waiting for new clients to arrive.

When the client starts up it will do so by loading its entry file. For every entry in the file the client issues add orders to add them to the clients main list of entries. This is catched by the library and an add order is sent to the diary server.

When there are no more entries to add, the client will send a re-synchronize order to the server. In test case 1 this will be no problem since both the client and the server is on the LAN.

If we issue an add order the entry will be sent to the diary server which inserts it in its entry-list. The client will send another re-synchronize order following the add order to check that everything was OK. Since we are operating on a LAN everything did go all right and thus the re-synchronize order was a success, and the entry-list will be inserted in the Fr.

The same happen if we issue a delete order.

Test case 2

The setting is the same as in Test case 1. The difference is that we simulate packet loss by using a random function, and show that the system *eventually* will be consistent. The random function gives a certain percentage that a message is lost. See fig 6.2

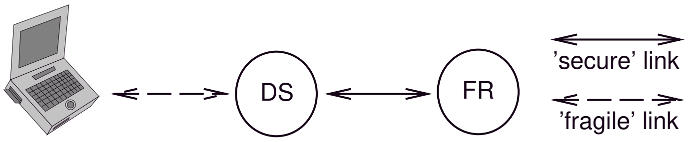


Figure 6.2: Test case 2

The client issues add and delete orders as in Test case 1. If the client do not get an answer it will warn the user by changing the color of the entry. A loss means either that the message was lost on the way from the client to the server or from the the server to the client. Mark that total disconnection is not especially interesting since all entries are lost. Whenever an entry is assumed lost the client will be put in the uncertain list. Those entries will be sent the next time the client succeeds in issuing a re-synchronize order to the server.

Test case 3

We have the following setting: The diary server and the Fr is set up the same way as in Test case 1 and 2. The clients are run on the machines diplom0.cs.-UiT.No, diplom1.cs.UiT.No and diplom2.cs.UiT.No. We will show that if any of the clients adds or removes an appointment all the other clients will see this change and change their databases accordingly. See fig 6.3

After the diary server has been started, the different clients are started. These will contact the diary server and re-synchronize themselves with it. There might be inconsistencies during this process but it will be solved in by getting or sending the inconsistent entries.

Since we in this situation have got one user with many machines one can assume that two orders are not issued at the same time. The server will on the

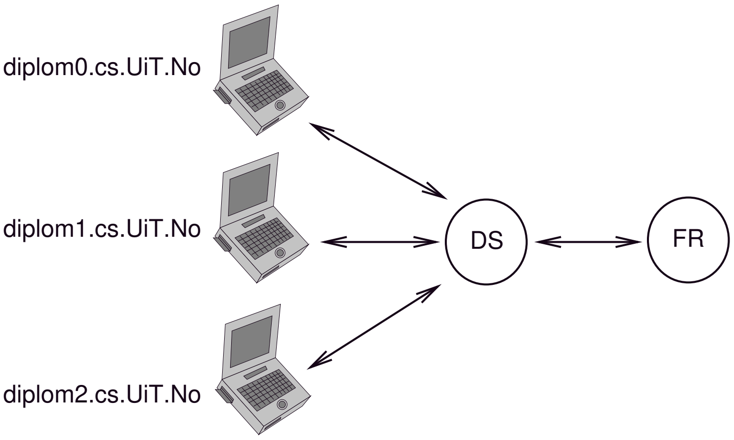


Figure 6.3: Test case 3

other hand serialize the orders.

Whenever one of the clients issue an add or delete order the server will add this order to all the other clients B_lists. When one of those clients issue a resynchronize order it will see that it has too few entries and will ask the server for these entries. The server will then send the entries to that client. This will eventually lead to a consistent state.

Test case 4

In this Test case we have the same setting as in the previous one. We simulate packet loss with the same random generator as in Test case 2. See fig 6.4.

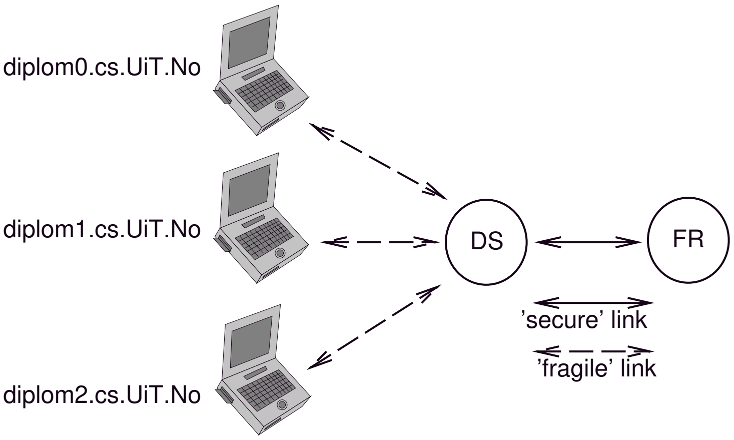


Figure 6.4: Test case 2

When a client issue an order that order has a chance of getting lost. As in Test case 2 the user will be warned by changing the color. The entry is put in the uncertain list. As in Test case 3 the server will put entries from an add or delete order in all other client's B_lists. In this way all the clients will get all the entries. The time this will take is greatly dependent by the chance of loosing messages. If the chance for loosing a message is p the chance for a successful transaction (from a clients view) is $(1-p)^2$.

Mark that for Test case 1 trough 4 the Fr do not play an important role. This is because we are just dealing with one person.

Another observation is the fact that if an off-line client make the system inconsistent by issuing an add or delete order with the other clients not aware of this, the human factor of the system can override this by ignoring the warnings.

Test case 5

In Test case 5 and 6 we move from an one-user environment to a multiuser environment. This demands use of the Fr. We will only describe the situation with two users each with one client for the sake of simplicity. See fig 6.5.

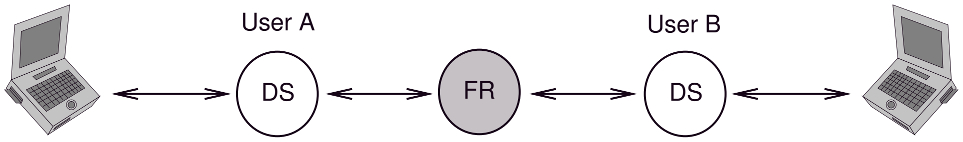


Figure 6.5: Test case 5

To add an entry in another user's diary, one first has to get all the entries for that user. This is done by the diary server which contacts the Fr and fetch the list of entries for that user. These entries are put in a special list, the other-list. After doing this the user can add entries. This work much the same way as having too few entries in the client.

When the diary server gets an entry for an other user, the diary server sees this and add the entry to the other-list, which is then written back into the file repository.

On the other hand the diary server for the other user will see this update next time it receives a re-synchronize order from one of the user's clients. Since the re-synchronize order gets the latest version of the entry file from the Fr it will see that another user has written in the file.

Mark that the communication between the diary server and the Fr is considered safe since they are located on the same LAN.

Test case 6

This is the most realistic test case. We use a number of users (3) each with a number of clients. The clients all have a chance of loosing messages with probability greater that 0. See fig 6.6.

As seen in Test case 4 the application handles the intra user consistency. We also saw in Test case 5 that the inter user consistency also is handled. By running this setting in practice it shows that the system works.

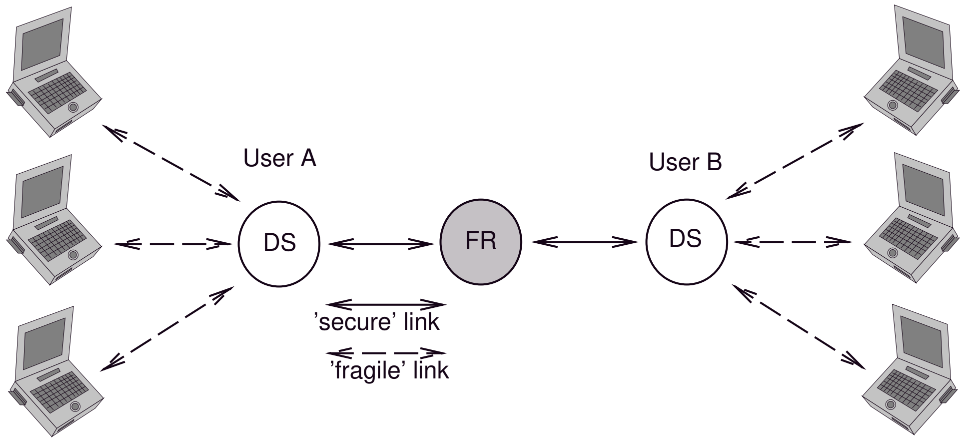


Figure 6.6: Test case 6 - only two users are showed

6.2 Results

The application is hard to measure in terms of performance or other numerical values. The only deadline the client has is the deadline for assuming that the server is unreachable. This will therefore be the maximum length of any transaction.

Further, the system has two possible states, correct or incorrect. All the tests so far have eventually left the system in a correct state. Being a distributed application, it is very difficult to test. This is because all the uncertainness the environment creates. Nothing is guaranteed. The hardware, like machines, network and clocks, the software, like operating system, transport protocols all introduce possible bugs, thus making the whole testing process a very uncertain process.

Another thing to discuss is the probability to lose messages. In the application this has been set to a probability p. One could argue that in the real world, the actual situation for a mobile machine like the "Pocket Companion" is much more like a boolean situation, either it is connected or it is not. This represent the situations where the PDA is either on or off, or in or out of a communication cell. We have chosen to test the application with a probability p to lose messages because this is harder for the system to manage than the boolean situation, but could still happen, and is therefore useful when one need to argue that it will work in an actual situation.

Chapter 7

Discussion

> *CARLSON'S CONSOLATION:*
>
> *Nothing is ever a complete failure; it can always serve as a bad example.*

We will in this chapter discuss the results and solutions presented in this thesis. Based on the test results from Chapter 6 we will evaluate the application against the current state of theory and existing systems from the Theory Chapter 2. A discussion about the success of reaching the goals presented in the Problem Definition follows then. (See section 1.2.) We will also try to present those things that characterize this thesis.

7.1 The Distributed Diary Compared to the Theory

In the Theory chapter we discussed two main problems, mobility problems and consistency problems. Many of the mobile problems discussed in the Theory chapter have not influenced this thesis due to different reasons like lack of hardware to test on or time to implement. This has made the system harder to test in a real environment, and further to a more theoretical view on those issues.

This is especially true for resource management. To measure and develop resource efficient applications one need the actual hardware to test the application on. It has not been possible to check an actual machine for the consequences of our design. Those machines used for testing have far too many resources than an actual mobile machine.

Human/computer interaction is also not possible to test, due to lack of actual hardware. It should however be easy to change the GUI of the application, which has been discussed theoretically in the thesis.

Some of the other issues discussed in the Theory chapter are too important to not take into consideration and is examined further in our research group. This is especially true for the security and privacy part. A study of how to do replication to increase the availability in the FR is also being done.

The lack of some of the important elements have on the other hand not been too bad for the thesis, because the main focus hasn't been on the mobile problems, but rather on problems concerning consistency. It has further lead to a simplified view of the environment, and we have thus had the ability to focus more on the basic problems.

The consistency part of the thesis got the main focus. As the test cases showed in the testing chapter, the optimistic consistency method showed to be good for our mobile and distributed application. The application handles off-line operation well, by warning the user about uncertain entries, and by getting into consistency by using store and forward when the mobile client gets back on-line.

The optimistic method is also based on another major observation, the user involvement. We can use very weak consistency in the set of clients of one user since the user has state information about the system. This was also documented in the tests.

Because the user is part of the decision loop we were also forced to have a much more pessimistic solution when dealing with consistency between different users. Since the user has less information about the global situation we were forced to choose pessimistic consistency control, because of the correctness criteria.

The testing also showed that the pessimistic approach to these problems worked. The pessimistic approaches was implemented in the FR which solved most of our distribution problems between different users. Because FR is (soon) replicated on different sites one could easily have a global system where people can contact each other using the FR. However, this does not solve the client's problem, which want to roam freely around the world without needing to do anything manually to tell others where they are.

Even though optimistic approaches have been used both in database and file systems, our approach goes a step further. Since we include the user's knowledge in the consistency process we have a system where the clients only need to be as consistent as the user decides. This is one of the main characteristics of our solution.

7.2 Design Weaknesses

There is, however, some weakness in our design. The assumption that the FR and the diary server is located on the same network is not optimal. This is not a problem when the clients move in reasonable distance from the diary server. The performance of the system may drop dramatically if the client move a long way ago, like to another country or continent. Even if the replication scheme in FR brings the data closer to you, the diary server is located back in the local network back home.

Another problem, which is rather close to the previous, is the problem of consistency between more than one clients that all are off-line, or by other means is not able to contact their diary server. These clients could be able to talk with each other, but because of the diary server, these machines will not be able to do anything to be more consistent. A solution to this problem could be to let clients both operate as client and servers. This is the approach used in the Bayou project which was described in section 2.4.11.

7.3 The Distributed Diary Against Existing Systems

As shown in the Theory chapter, most existing system with multiuser support are either stationary systems or stationary systems with portable support. They also need great resources in terms of communication bandwidth and internal resources. These things are not good enough for a PDA with networking possibilities. Mobile computers need simple, yet functional, applications, which do not need large amounts of resources.

Although our solution lacks much of the group-ware possibilities the systems described offers, our application represent a framework to build such features on. Our application also have the advantage that it has been built from scratch with mobility in mind, and not as an extra feature like most of the the systems described. Based on the two most important features a distributed diary must have, availability and consistency, we have built a very adaptable application which support mobile machines by offering off-line operation and eventual consistency. Since mobile machines belong to one extreme side of computer science, we have made a rather extreme solution to get as much availability as possible without loosing too much of the consistency part. This we have done by the user involvement.

While other applications tries to hide as much as possible of the consistency part in the application, we tries to utilize the state information the user has to further increase the availability. It follows from this that the system never has a notion of global consistency, leaving this global information to the user.

Most users of existing portable systems have to tell their system when they are about to move, so that the application can route their data to the new place. We have not taken this approach, assuming that the user is moving all the time. This has lead to a solution that doesn't assume anything about where the user are located, or about what type of machine the user are currently using.

7.4 Discussion of the Problem Definition

In Chapter 1 we presented the problem definition which presented the goal of this thesis. Our main goal was to build a diary application in a heterogeneous machine environment, and see that even though communication between these machines had varying quality it was possible to build such an application.

A consequence of the problem definition is that because of the varying quality of the communication, the application need to support off-line operations. This question has been discussed in detail and lead to further questions. First the support for off-line operation lead to questions about consistency. Secondly it leads to questions about the level of trust the users of this application could have in a system not guaranteed consistent. This has also been discussed in depth and we have concluded with that the best way of raising this trust is to involve the user in the decision loop, and letting the user get some of the responsibility for the possible lack of consistency.

The issue of consistency has been discussed in depth in this thesis and we concluded with an optimistic view. This is a natural consequence of letting the user do off-line operations.

Chapter 8

Concluding Remarks and Further Work

> *CLARK'S THIRD LAW:*
> *Any sufficiently advanced technology is indistinguishable from magic.*

We have in this thesis designed, implemented, installed and tested a file repository based distributed diary. The main research issue has been the consistency problems that have occurred when more than one client owned by the same user interacts with the user's private calendar server and when users tries to add entries to other persons' diaries.

8.1 Concluding Remarks

The diary application presented in this thesis focus on many of the problems one come across when one want to make a distributed application in a heterogeneous environment consisting of mobile and stationary machines. Many of the experiences described in this thesis could be of use for other people wanting to write application which includes heterogeneous machines. Our experiences with human involvement could also be of some use. Further, the eventual consistency model used in this thesis was also good choice in this environment.

Further, the science of mobile computing is a relative new field in the computer science area. This has lead to renewed interest in some of the fields of research that has been considered "solved", like consistency in database and file systems. Hopefully this thesis has been of some interest in this renewed fields of research.

One of the most important results from this thesis is to show that this is an application which can be used with the "Pocket Companion". Even though the application has not been tested on mobile machines, we have designed the application in a way that it would fit in a machine like the "Pocket Companion". We will hopefully see an application like our distributed diary in a solution like that.

Since most of the mobile machines we see today are stand alone machines without possibility to communicate wireless with the rest of the world, there exist little software designed specially for these computers. Our distributed diary should hopefully be an element in the process of developing applications for mobile computers.

Many people see mobile computing as a new paradigm in computer science. We will hopefully see that in the years to come these small machines will enter the communication world which we all are part of, and give us locality dependent information and applications which are very personal, like The Distributed Diary. These machines will maybe mean as much as today's PCs to travelers and business people, and will possibly be used by more people than uses PCs today.

8.2 Further work

Although much work has been laid into the distributed diary, we consider it only a framework for later work. Many of the basic elements of such an application has been studied, but little work has been laid in diary functionality.

Many of the commercial available system have strong group-ware support, ranging from resource allocation to group meetings functionality.

However, there are many basic things that must be looked at before one could start implementing the group-ware functionality. As described in the Theory chapter (chapter 2), work has to be done of different fields of research, like security and packed routing. We won't have real mobile computing until an infrastructure is developed that support mobility.

Because of this, work must be done to make today's networking protocols better for mobile computing. This is however too big an issue to give to application programmers and assumes a joint concentration of force by the major players in the network society. As described in the Theory chapter, work are already being done in this field.

When we look at the possibilities to make a better application some studies about power consumption could be done. The application should be optimized for using as little power as possible.

Further, since the application has been developed in an environment with large resources on user/machine I/O like "large" laptop and workstation screens and good keyboards and mice, much work has to be done in this field. Since `Plan` is implemented using MOTIF, it demands a window system from the machine which it runs on. Window systems are not much used in the PDA world.

This could however change. Microsoft, with their introduction of Windows CE, an operating system for small machines like mobile machines, will use a windowing system. This will greatly influence the market for small machines and the next generation of mobile machines will probably all use this system. It will probably make the application designers task easier since this OS probably will be the common platform for these small machines, and thus raise the interest for application development for small and mobile machines.

This means that one should examine more deeply the user/computer interface, both considering how to present the data on a small screen and how to input data.

We described some of the weaknesses with our system in the discussion section. These are obvious candidates for further work. As described in the Existing System section of the Theory chapter, the Bayou project at Xerox Parc uses clients that also could be servers for other machines. This could also be done with our distributed diary.

We can also argument for this from the fact that we have tried to write our client software as a client library. The clients should run on machines with greatly diverging resources and different graphical user environments should be supported. This would lead to interesting clients like Java language Web client. There are some commercial initiatives in this area. Lotus `Organizer 97` have got a Web add-on to access and share calendar information over the Web[1].

We also described how mobile machines tend to be more personal in the Theory chapter. This means that also the distributed diary should have this in mind. As described before, users can look at other diaries and write directly into their diaries. there are no kind of authentication or security whatever. This must be studied further.

---

[1] See http://www.lotus.com/organizer/2156.htm for details.

Bibliography

[Badrinath et al., 1993] Badrinath, B. R., Bakre, A., Imielinski, T., and Marantz, R. (1993). Handling mobile clients: A case for indirect interaction. *Fourth Workshop on Workstation Operating Systems.*

[CE Software Inc, 1995] CE Software Inc. (1995). Time vision network scheduler 3.5.
URL: http://www.cesoft.com/info/tvnetworkschedulerinfo.html.

[CommonTime Ltd, 1996] CommonTime Ltd. (1996). Commontime.
URL: http://www.trendpoint.co.uk/tp/commontime.html.

[Davidson et al., 1985] Davidson, S. B., Garcia-Molina, H., and Skeen, D. (1985). Consistency in partitioned networks. In *Computing Surveys*, volume 17, 3. ACM.

[Demers et al., 1995] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Welch, B. (1995). The Bayou Architecture: Support for Data Sharing among Mobile Users

[Denning et al., 1989] Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989). Computing as a discipline. *Communications of the ACM*, 32(1):9 – 23.

[Downing et al., 1990] Downing, A. R., Greenberg, I. B., and Peha, J. M. (1990). Oscar: A system for weak-consistency replication. In Cabera, L.-F. and Paris, J.-F., editors, *Proceedings of the Workshop on Management of Replicated Data*, Huston, Texas. IEEE.

[Driemeyer, 1996] Driemeyer, T. (1996). Plan - calendar and day planner.
URL: http://www.IN-Berlin.DE/User/bitrot/plan.html.

[Elmasri and Navathe, 1994] Elmasri, R. and Navathe, S. B. (1994). *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc, second edition.

[Fallmyr, 1996] Fallmyr, T. (1996). API towards the PASTA File Repository. Technical Report 96-26, Department of Computer Science, University of Tromsø.

[Fallmyr et al., 1995] Fallmyr, T., Hartvigsen, G., and Stabell-Kulø, T. (1995). Supporting mobile users in a variable connected distributed system: the pasta approach. *NIK'95*.

[Forman and Zahorjan, 1993] Forman, G. H. and Zahorjan, J. (1993). The challenges of mobile computing. Technical Report 93-11-03, Computer Science & Engineering, University of Washington.

[Garcia and Weiderhold, 1982] Garcia, H. and Weiderhold, G. (1982). Read-only transactions in a distributed system. In *ACM Transaction on Database Systems*, volume 7, 2, pages 209–234. ACM.

[Garg and Wilkes, 1996] Garg, V. K. and Wilkes, J. E. (1996). *Wireless and Personal Communications Systems*. Prentice Hall, Upper Saddle River, New Jersey.

[Ghemawat, 1993] Ghemawat, S. (1993). Ical.
URL: http://www.pmg.lcs.mit.edu:80/˜sanjay/ical/doc.html.

[Gifford, 1979] Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the 7th Synopsium on Operating Systems Principles*, pages 155–162, Pacific Grove, Calif. ACM.

[Golding, 1992] Golding, R. A. (1992). *Week-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc.

[Herlihy, 1984] Herlihy, M. P. (1984). General quorum consensus: A replication method for abstract data types. Technical report, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

[Kernigham and Ritchie, 1988] Kernigham, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall.

[Kistler and Satyanarayanan, 1992] Kistler, J. and Satyanarayanan, M. (1992). Disconnected operation in the coda file system. *TOCS*, 10(1):3–25.

[Koch et al., 1993] Koch, H., Krombholz, L., and Theel, O. (1993). A breif introduction into the world of “mobile computing”. Technical Report THD-BS-1993-03, Department of Computer Science, University of Darmstadt.

[Kopetz and Veríssimo, 1993] Kopetz, H. and Veríssimo, P. (1993). *Real Time and Dependability Concepts*, chapter 16, pages 411–446. ACM Press Frontier Series. Addidon-Wesley Publishing Company.

[Louts Corporation, 1996] Lotus Development Corporation. (1996). Calendaring & scheduling.
URL: http://www.lotus.com/calendar/.

[Microsoft, 1996] Microsoft (1996). Microsoft schedule+.
URL: http://www.microsoft.com/msscheduleplus/.

[Olsen, 1996] Olsen, D. F. (1996). *Construction of a Distributed Diary.* Semester thesis, University of Tromsø, Spring 1996

[Parker et al., 1983] Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kieser, S., and Cline, C. (1983). Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw.*, Eng. 9(3).

[Perkins et al., 1994] Perkins, C., Johnson, D. B., and Myles, A. (1994). Mobility support in ipv6. Technical report, Network Working Group. Internet Draf.

[Pitoura and Bhargava, 1995] Pitoura, E. and Bhargava, B. (1995). Maintaining consistency of data in mobile distributed environments. In *Proceedings of the 15th International Conference on Distributing Computing Systems.*

[Pressman, 1994] Pressman, R. S. (1994) *Software Engineering: Practitioner's Approach.* McGrawy-Hill Book Company Europe, European third edition

[Rivest, 1992] Rivest, R. (1992). The md5 message-digest algorithm. RFC 1321.

[Satyanarayanan et al., 1993] Satyanarayanan, M., Kistler, J. J., Mummert, L. B., Ebling, M. R., Kumar, P., and Lu, Q. (1993). Experience with disconnected operation in a mobile comuting environment. In *Proceedings of the 1993 USENIX Symopsium on Mobile and Location-Independent Computing.*

[Stabell-Kulø, 1995] Stabell-Kulø, T. (1995). File Repository Transfer Protocol. Verson 1 Computer Science Technical Report no. 95-21.

[Stevens, 1990] Stevens, W. R. (1990). *UNIX network programming.* Prentice-Hall International Inc.

[Svobodova, 1984] Svobodova, L. (1984). *File Servers For Network-based Distributed Systems*, volume 16 of *Computer Surveys*, chapter 4, pages 353–398.

[Tanebaum, 1989] Tanebaum, A. S. (1989). *Computer Networks.* Prentice-Hall International Inc.

[Verimation AB, 1995] Verimation AB (1995). Memo - product overview.
URL: http://www.verimation.se/memo_po.html.

[Veríssimo, 1993] Veríssimo, P. (1993). *Real-Time Communication*, chapter 17, pages 447–490. ACM Press Frontier Series. Addidon-Wesley Publishing Company.

[Vogel et al., 1995] Vogel, A., Kerhervé, B., von Bochmann, G., and Gecsei, J. (1995). Distributed multimedia and qos: A survey.

Appendix A

System Scenarios

In this chapter we describe the three main situations the diary application could encounter. The first two describe an off-line situation between the client and the server, with the difference being who make the system inconsistent. The third is the most "real world" situation where every component may inflict an inconsistency.

A.1 Client Inconsistency

In this scenario we will look at what could happen when the client operates in an off-line mode. This means that the effect of operations in the client makes it inconsistent with the server. Mark that the inconsistent state is only between the client and the server, where the client is right. The changes will be saved on stable storage until the client can resume the interaction with the server, and knows that it is consistent with the server. See figure A.1

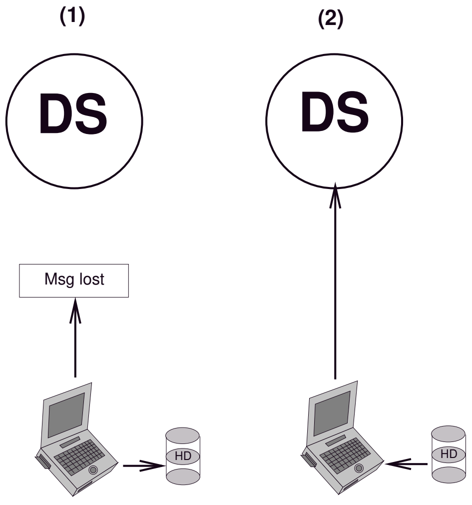


Figure A.1: Client inconsistency

When interaction is resumed it is up to the client to get the state consistent.

Mark that it is very easy to get the system back into a consistent state. This is because the client backs up the entries it knows the server doesn't have. The client sends these entries to the server when it comes back on line.

A.2 Server inconsistency

This is primary the same situation as the client inconsistency scenario. The only difference is that the roles have changed, some other person wants to write an appointment into the user's diary. To do this we insert an entry in the users Fr. Since the diary server polls the Fr it will get this change. This time it is the server who has the newest appointment, with the client not knowing of it. The client will discover this the next time it tries to re-synchronize itself with the server since the server will tell the client that it has to few entries. See figure A.2

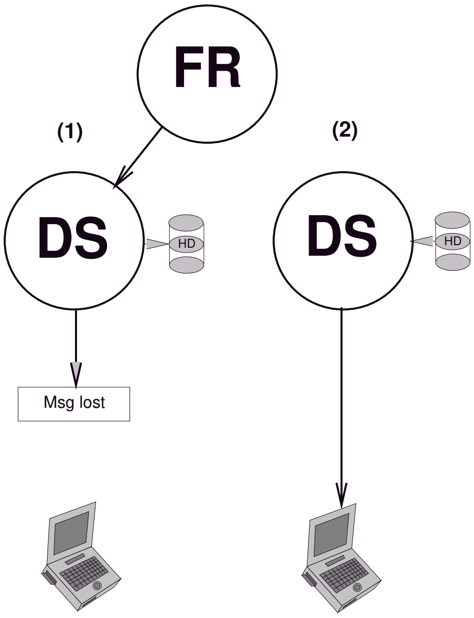


Figure A.2: Server inconsistency

Since the diary server manages more than one client, we have to be sure that the entry is kept until all clients know about it.

A.3 Client/Server inconsistency

This is a combination of the two situations described above, and has a bad worst-case namely the situation where one get into an unsolvable consistency conflict.

An example of this is the user making an appointment while other users makes appointments with the user. This may lead to "I thought that you thought that we had an appointment" inconsistencies. This type of inconsistencies are due to the fact that the users have far less state information about the global diary system, than they have about their own system. See figure A.3.

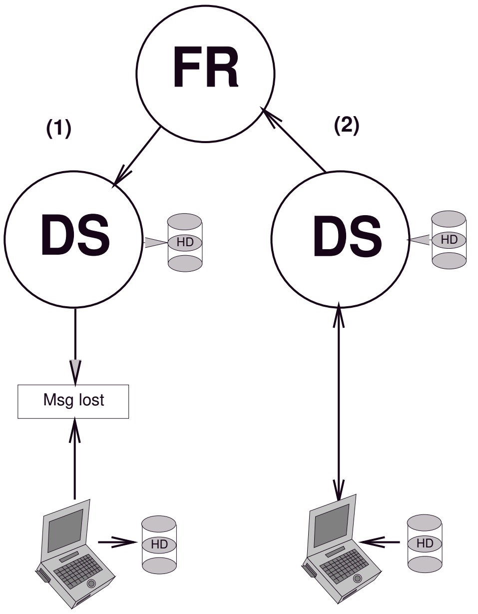


Figure A.3: Client/Server inconsistency, the real world problem

Our system doesn't try to solve this problem, since it is a semantical problem, and can't be solved *good* without user involvement. Syntactically it's not a problem since we make sure that the client is consistent with the diary server which in turn is consistent with the Fr.

Appendix B

Header files

B.1 server.h

```
/* -*-C-*-
*****************************************************************
*
* File:         server.h
* RCS:          $Header: /usr/local/PASTA/cvs-rep/PASTA/
*                           src/diary/source/Diplom/include/server.h
*                           v 1.2 1996/10/30 17:22:36 dago Exp $
* Description:
* Author:       Dag-Frode Olsen
* Created:      Tue Aug  20 22:02:35 1996
* Modified:
* Language:     C
* Package:      N/A
* Status:       Experimental (Do Not Distribute)
*
* (c) Copyright 1996, Dag-Frode Olsen, all rights reserved.
*
*****************************************************************
*/

/*
*****************************************************************
*
*       server.h
*
*****************************************************************
* Description
*       Here is all the structs used by both the client and the
*       server.
*       These are:
*         * entry_t   - a representation of an entry without any
*                       pointers.
*                       See client code for further info.
*         * mesg_t    - a mesg struct.
*         * entry_l_t - a list of entries.
*         * list_t    - struct with pointer to the first entry int
*                       the entrylist and two integers representing
*                       the number of entries known.
```

```
*        * certain_t - a list of uncertain entries.
*        * client_t  - a struct with include info about a client.
*
*/

#ifndef SERVER_H
#define SERVER_H

#define OK        0
#define ERROR     -1
#define TIMEOUT     -10

#define ERR_CHANCE  1.0

#define MAX_CLIENTS 5

typedef struct entr {
  long          nextrep;
  char          user[MAX_SIZE];
  char          message[MAX_SIZE*16];
  char          script[MAX_SIZE*16];
  char          note[MAX_SIZE];
  time_t        time;
  time_t        length;
  time_t        early_warn;
  time_t        late_warn;
  time_t        except[NEXC];
  time_t        rep_every;
  time_t        rep_last;
  long          rep_weekdays;
  long          rep_days;
  BOOL          rep_yearly;
  BOOL          suspended;
  BOOL          private;
  BOOL          noalarm;
  BOOL          notime;
  BOOL          hide_in_m;
  BOOL          hide_in_y;
  BOOL          hide_in_w;
  char          acolor;
  char          triggered;
} entry_t;

typedef struct mesg {
  int        type;
  unsigned   char    checksum[16];
  int        cli_n;
  int        ser_n;
  entry_t    entr;
} mesg_t;

typedef struct entry_list {
  entry_t    entr;
  BOOL       delete;
  struct     entry_list  *next;
} entry_l_t;
```

```
typedef struct mainlist_list {
  int         cl_entries;
  int         se_entries;
  char        name[10];
  entry_l_t   *first;
} list_t;

typedef struct certain_list {
  char        note[MAX_SIZE];
  time_t      time;
  BOOL        overlap;
  struct certain_list *next;
} certain_t;

typedef struct clients {
  char        *client_name;
  int         number;
  list_t      *A_list;
  list_t      *B_list;
} client_t;


#endif /* SERVER_H */

```

B.2 socklib.h

```
/* -*-C-*-
****************************************************************
*
* File:        socklib.h
* RCS:         $Header: /usr/local/PASTA/cvs-rep/PASTA/
*                       src/diary/source/Diplom/include/socklib.h
*                       v 1.2 1996/12/09 18:39:16 dago Exp $
* Author:      Dag-Frode Olsen
* Created:     Tue Aug  20 22:02:35 1996
* Language:    C
* Modified     Tue Aug  20 22:02:35 1996 (Dag-Frode Olsen)
* Status:      Experimental (Do Not Distribute)
*
* Orginally written by Arne Helme
*
****************************************************************
*/

/*
****************************************************************
*
*       server.h
*
****************************************************************
* Description
*   All network primitives are defined here.
*
*/
#ifndef _SOCKLIB_INCLUDED
#define _SOCKLIB_INCLUDED

/* Error values and their meanings */
#define SL_ESOCKET      0 /* socket system call error */
#define SL_EBIND        1 /* bind system call error */
#define SL_EHOSTNAME    3 /* Invalid host name format */

#ifndef _SOCKLIB_LIBSOURCE
#ifdef VMS
#include <in.h>                 /* for sockaddr_in */
#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 64
#endif
#else
#include <netinet/in.h>         /* for sockaddr_in */
#endif
extern int
    sl_errno,
    sl_timeout_s,
    sl_timeout_us,
    sl_default_retries,
    sl_broadcast_enabled;
extern struct sockaddr_in
    sl_dgram_lastaddr;

#endif /* _SOCKLIB_LIBSOURCE */
```

```
#ifdef __cplusplus
#ifndef __STDC__
#define __STDC__    1
#endif
#endif

#ifdef __STDC__
extern int    CreateDgramSocket(int);
extern int    DgramSend(int, char *, int, char *, int);
extern int    DgramReceiveAny(int, char *, int);
extern int    DgramReply(int, char *, int);
extern char   *DgramLastname(void);
#else /* __STDC__ */
extern int    CreateDgramSocket();
extern int    DgramSend();
extern int    DgramReceiveAny();
extern int    DgramReply();
extern char   *DgramLastname();
#endif /* __STDC__ */

#if !defined(select) && defined(__linux__)
#define select(N, R, W, E, T) select((N),            \
        (fd_set*)(R), (fd_set*)(W), (fd_set*)(E), (T))
#endif

#endif /* _SOCKLIB_INCLUDED */
```

Appendix C

Client files

C.1 network.c

```
/* -*-C-*-
*****************************************************************
*
* File:         network.c
* RCS:          $Header: /usr/local/PASTA/cvs-rep/PASTA/
*                   src/diary/source/include/server.h
*                   v 1.10 1996/06/04 19:40:07 dago Exp $
* Author:       Dag-Frode Olsen
* Created:      Tue Aug  20 22:02:35 1996
* Language:     C
* Status:       Experimental (Do Not Distribute)
*
* (c) Copyright 1996, Dag-Frode Olsen, all rights reserved.
*
*****************************************************************
*/

/*
*****************************************************************
*
*       network.c
*
*****************************************************************
* This is the file dealing with the client functions
*   Functions:
*     randint - generate random numbers.
*     make_socket - generate a socket and bind it.
*     mashall_info - marshalls an entry an puts it in a message.
*     unmashall - unmashalls a massage and puts it in an entry.
*     md5_list - make a md5 checksum of a list.
*     backup_entry - saves en entry on stable storage and marks it
*                    uncertian.
*     get_backup - reads the backupfile and tries to redo the order.
*     send_resync - tries to resyncronize the client with the server.
*     send_entry - sends a entry to the server.
*     entry_add - adds an entry to the list.
*     entries_add - add all entries in plan's mainlist to clients
*                   mainlist.
```

```
*       get_user - contact FR to get all the users entries.
*       get_server_entries - get all unknown entries from server.
*       find_entry - locates an entry in the mainlist.
*       send_delete_entry - sends an delete entry order to the server.
*       entry_delete - ready the delete mesg.
*       del_entry - deletes an entry from a list
*
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <time.h>
#include "conf.h"
#include "socklib.h"
#include "md5.h"
#include "server.h"

int                     sport   = PORT;

int                     socket;
const double los_prob = ERR_CHANCE;/* Probability of lost message*/

struct backup {
  int order;
  entry_t entry;
};

extern char             *username;
extern struct list      *mainlist; /* list of all schedule entries*/
extern certain_t        *certain;  /* list of uncertain entries */
extern char             *server;

#define TIMEOUT_SEC 9       /* Number of sec's to wait */
#define TIMEOUT_USEC 0      /* Number of usec's to wait */

#define RESYNCHRONIZE 1     /* Resync order */
#define ADD          2      /* Add order */
#define DELETE       3      /* Delete order */
#define GET          4      /* Get order */
#define GET_USER        5         /* Load a user from FR */
```

```
/*
 **************************************************************
 *
 *      static int randint(int lo, int hi)
 *
 **************************************************************
 * Description
 *      Generate a random number beteween lo and hi.
 *
 * Input Parameters
 *      int lo - lower int
 *      int hi - higher int
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns a random number
 *
 * Called By
 *      network functions.
 */
static int randint(int lo, int hi)
{
  static int first_call = TRUE;     /* Detects first call */

  /* Seed */
  if (first_call) {
    first_call = FALSE;
    /* We want different seeds for different
       executions of the program */
    srand48((long)time(NULL));
  }

  return lo + (int)lrand48() % (hi − lo + 1);
}
```

```
/*
 ************************************************************
 *
 *      int make_socket()
 *
 ************************************************************
 * Description
 *      Make a socket and bind it
 *
 * Input Parameters
 *      None
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns -1 if any errors occured.
 *
 * Called By
 *      Main program.
 */
int make_socket()
{
  /* Make socket and bind it */
  if ((socket = CreateDgramSocket(0)) == -1)
    return -1;
  else
    return 0;
}
```

```
/*
 ****************************************************************
 *
 *      void mashall_info(int order, struct entry *entry,
 *                        mesg_t *tmp)
 *
 ****************************************************************
 * Description
 *      Fill in header information and copy data from an entry into
 *      a message.
 *
 * Input Parameters
 *      order - the type of order.
 *      entry - the entry to copy from.
 *
 * Output Parameters
 *      tmp - the mashalled message.
 *
 * Return Value
 *      None
 *
 * Called By
 *      network functions.
 */
void mashall_info(int order, struct entry *entry, mesg_t *tmp)
{
  tmp->type = order;
  strcpy(tmp->checksum,'\0');

  tmp->entr.nextrep = entry->nextrep;

  if (entry->user == NULL)
    strcpy(tmp->entr.user, username);
  else
    strcpy(tmp->entr.user, entry->user);

  if (entry->message != NULL)
    strcpy(tmp->entr.message, entry->message);
  else
    strcpy(tmp->entr.message, "");

  if (entry->script != NULL)
    strcpy(tmp->entr.script, entry->script);
  else
    strcpy(tmp->entr.script, "");

  if (entry->note != NULL)
    strcpy(tmp->entr.note, entry->note);
  else
    strcpy(tmp->entr.note, "");

  tmp->entr.time = entry->time;
  tmp->entr.length = entry->length;
  tmp->entr.early_warn = entry->early_warn;
  tmp->entr.late_warn = entry->late_warn;
  tmp->entr.except[NEXC] = entry->early_warn;
  tmp->entr.rep_every = entry->rep_every;
```

```
  tmp->entr.rep_last = entry->rep_last;
  tmp->entr.rep_weekdays = entry->rep_weekdays;
  tmp->entr.rep_days = entry->rep_days;
  tmp->entr.rep_yearly = entry->rep_yearly;
  tmp->entr.suspended = entry->suspended;
  tmp->entr.private = entry->private;
  tmp->entr.noalarm = entry->noalarm;
  tmp->entr.notime = entry->notime;
  tmp->entr.hide_in_m = entry->hide_in_m;
  tmp->entr.hide_in_y = entry->hide_in_y;
  tmp->entr.hide_in_w = entry->hide_in_w;
  tmp->entr.acolor = entry->acolor;
  tmp->entr.triggered = entry->triggered;
}
```

```
/*
 ***************************************************************
 *
 *      void unmashall(entry_t *entr, struct entry *entry)
 *
 ***************************************************************
 * Description
 *      Make an ordinary entry from a message.
 *
 * Input Parameters
 *      entr - the entry to copy from.
 *
 * Output Parameters
 *      entry - an ordinary "plan" entry.
 *
 * Return Value
 *      None
 *
 * Called By
 *      network functions.
 */
void unmashall(entry_t entr, struct entry *entry)
{
  entry->nextrep = entr.nextrep;

  entry->user = (char *)malloc(sizeof(char)*MAX_SIZE);
  if (strcmp(entr.user, username) == 0)
    entry->user = (char *)0;
  else
    strcpy(entry->user, entr.user);

  entry->message = (char *)malloc(sizeof(char)*MAX_SIZE*16);
  if (strlen(entr.message) != 0)
    strcpy(entry->message, entr.message);
  else
    entry->message = (char *)0;

  entry->script = (char *)malloc(sizeof(char)*MAX_SIZE*16);
  if (strlen(entr.script) != 0)
    strcpy(entry->script, entr.script);
  else
    entry->script = (char *)0;

  entry->note = (char *)malloc(sizeof(char)*MAX_SIZE);
  if (strlen(entr.note) != 0)
    strcpy(entry->note, entr.note);
  else
    entry->note = (char *)0;

  entry->time = entr.time;
  entry->length = entr.length;
  entry->early_warn = entr.early_warn;
  entry->late_warn = entr.late_warn;
  entry->except[NEXC] = entr.early_warn;
  entry->rep_every = entr.rep_every;
  entry->rep_last = entr.rep_last;
  entry->rep_weekdays = entr.rep_weekdays;
```

```
  entry->rep_days = entr.rep_days;
  entry->rep_yearly = entr.rep_yearly;
  entry->suspended = entr.suspended;
  entry->private = entr.private;
  entry->noalarm = entr.noalarm;
  entry->notime = entr.notime;
  entry->hide_in_m = entr.hide_in_m;
  entry->hide_in_y = entr.hide_in_y;
  entry->hide_in_w = entr.hide_in_w;
  entry->acolor = entr.acolor;
  entry->triggered = entr.triggered;
}
```

```
/*
 ****************************************************************
 *
 *      int md5_list(list_t *list, unsigned char digest[])
 *
 ****************************************************************
 * Description
 *      Make a md5 checksum of the mainlist. Remove any old entries
 *      from the list. Checks must be done to check for repeating
 *      entries.
 *
 * Input Parameters
 *      *list - pointer to the mainlist.
 *
 * Output Parameters
 *      digest[] - the checksum digest.
 *
 * Return Value
 *      none
 *
 * Called By
 *      network functions.
 */
int md5_list(list_t *list, unsigned char digest[])
{
  MD5_CTX    context;
  entry_l_t *ep, *ep2;
  time_t     now;

  now = time(0);

  MD5Init (&context);

  ep = list->first;
  if (!(ep->entr.time > now || ep->entr.rep_last > now ||
      (ep->entr.rep_every > 0 && ep->entr.rep_last == 0) ||
      (ep->entr.rep_weekdays > 0 && ep->entr.rep_last == 0) ||
      ep->entr.rep_yearly == TRUE)) {
    list->first = ep->next;
    free(ep);
  }
  else {
    ep2 = list->first;
    ep = ep2->next;
    while (ep != NULL)
      if (!(ep->entr.time > now || ep->entr.rep_last > now ||
            (ep->entr.rep_every > 0 && ep->entr.rep_last == 0) ||
            (ep->entr.rep_weekdays > 0 &&
             ep->entr.rep_last == 0) ||
            ep->entr.rep_yearly == TRUE)) {
        ep2->next = ep->next;
        free(ep);
      }
      else {
        ep2 = ep;
        ep = ep->next;
      }
```

```
  }

  ep = list->first;

  while (ep != NULL) {
    MD5Update (&context, (unsigned char *)&ep->entr.user,
               strlen(ep->entr.user));
    MD5Update (&context, (unsigned char *)&ep->entr.message,
               strlen(ep->entr.message));
    MD5Update (&context, (unsigned char *)&ep->entr.script,
               strlen(ep->entr.script));
    MD5Update (&context, (unsigned char *)&ep->entr.note,
               strlen(ep->entr.note));
    MD5Update (&context, (unsigned char *)&ep->entr.time,
               sizeof(ep->entr.time));
    MD5Update (&context, (unsigned char *)&ep->entr.length,
               sizeof(ep->entr.length));
    ep = ep->next;
  }
  MD5Final(digest, &context);

}
```

```
/*
 ****************************************************************
 *
 *      void backup_entry(entry_t entry, int order)
 *
 ****************************************************************
 * Description
 *      Backing up an entry. The entry is not confired to be in the
 *      server. Also back up the type of order issued. An add order
 *      must be handled different than a delete order.
 *
 * Input Parameters
 *      entry - entry to be backed up.
 *      order - type of order to back up.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      None
 *
 * Called By
 *      network functions
 */
void backup_entry(entry_t entry, int order)
{
  FILE        *fp;
  char        *path;
  certain_t   *cer, *trav;
  BOOL        found;
  struct backup backup;

  /* check if entry is uncertain */
  found = FALSE;
  cer = certain;
  while (cer != NULL && found == FALSE) {
    if (strcmp(cer->note, entry.note) == 0)
      found = TRUE;
    cer = cer->next;
  }

  if (found == FALSE) {
    fprintf(stderr, "BACKING up entry %s. \n", entry.note);

    /* empty */
    if (certain == NULL) {
      cer = (certain_t *)malloc(sizeof(certain_t));
      strcpy(cer->note, entry.note);
      cer->time = entry.time;
      cer->overlap = FALSE;
      cer->next = (certain_t *)0;
      certain = cer;
    }
    else if (certain->next == NULL) {
      cer = (certain_t *)malloc(sizeof(certain_t));
      strcpy(cer->note, entry.note);
      cer->time = entry.time;
```

```
      cer->overlap = FALSE;
      cer->next = (certain_t *)0;
      certain->next = cer;
    }
    else {
      cer = (certain_t *)malloc(sizeof(certain_t));
      strcpy(cer->note, entry.note);
      cer->time = entry.time;
      cer->overlap = FALSE;
      cer->next = (certain_t *)0;
      trav = certain;
      while (trav->next != NULL)
        trav = trav->next;
      trav->next = cer;
    }

    /* write to file */
    path = (char *)malloc(sizeof(char *)*100);
    sprintf(path, "%s.%s", BACKUP_PATH, username);
    if ((fp = fopen(path, "a")) == NULL)
      exit(-1);
    backup.order = order;
    backup.entry = entry;
    if (fwrite((void *) &backup, sizeof(struct backup), 1, fp) < 1)
      fprintf(stderr, "fwrite error");
    fclose(fp);
  }
  else
    fprintf(stderr, "Entry %s has already been backed up!\n",
            entry.note);
}/*
 ***************************************************************
 *
 *     void get_backup(list_t *list)
 *
 ***************************************************************
 * Description
 *     Read the backupfile. Call send_entry or send_delete_entry
 *     to reissue the order. If that is successfull delete
 *     that entry from the backupfile and the uncertainlist, else
 *     do nothing.
 *
 * Input Parameters
 *     list - the serverlist;
 *
 * Output Parameters
 *     None
 *
 * Return Value
 *
 * Called By
 *     main program
 */
BOOL get_backup(list_t *list)
{
  FILE       *fp, *fp2;      /* open file */
  char       *path;          /* the actual file to write to */
```

```
BOOL        sync = TRUE;
int         error;
mesg_t      mesg;
certain_t   *cer, *del;
entry_t     entry;
struct backup backup;
int         order;
entry_l_t   *ep;

path = (char *)malloc(sizeof(char *)*128);
sprintf(path, "%s.%s", BACKUP_PATH, username);

if ((fp = fopen(path, "r")) == NULL){   /* re readdatabase */
  fprintf(stderr, "No backupfile. Adding mainlist\n");
  entries_add(list);
  return(OK);
}

sprintf(path, "%s.%s", BACKUP_PATH, "tmp");
if ((fp2 = fopen(path, "w")) == NULL)
  exit(-1);

while (!feof(fp) && fread((void *) &backup,
                          sizeof(struct backup), 1, fp)) {
  order = backup.order;
  entry = backup.entry;

  fprintf(stderr, "Got entry %s form backupfile\n", entry.note);

  mesg.type = order;
  mesg.checksum[0] = '\0';
  mesg.cli_n = 0;
  mesg.ser_n = 0;
  mesg.entr = entry;

  if (order == ADD)
    error = send_entry(mesg, list, TRUE);
  else
    error = send_delete_entry(mesg, TRUE);

  if (error != OK) {
    sync = FALSE;
    if (order == ADD)
      if (error == TIMEOUT)
        list->se_entries--;
    else
      if (error == TIMEOUT)
        list->se_entries--;
  }
  else
    if (order == ADD)
      list->se_entries++;
    else
      list->se_entries--;

  if (error == OK || error == -12) {
    /* remove entry from the uncertainlist */
```

```
    /* delete first */
    del = certain;

    if ((certain->time == entry.time) &&
        (strcmp(certain->note, entry.note) == 0)){
      certain = del->next;
      free(del);
    }
    else {
      del = certain;
      cer = del->next;
      while (cer != NULL)
        if ((cer->time == entry.time) &&
            (strcmp(cer->note, entry.note) == 0)){
          del->next = cer->next;
          free(cer);
          cer = cer->next;
        }
        else {
          del = cer;
          cer = cer->next;
        }
    }
    fprintf(stderr, "Retransmission of entry %s successfull\n",
            entry.note);
  }
  else {
    /* save entry on temporary backupfile */
    if (fwrite((void *) &backup, sizeof(struct backup),
                1, fp2) < 1)
      fprintf(stderr, "fwrite error");
  }
}
fclose(fp);
fclose(fp2);

path = (char *)malloc(sizeof(char *)*128);
sprintf(path, "%s.%s", BACKUP_PATH, username);

if (sync) {
  if ((fp = fopen(path, "w")) == NULL)
    return (-1);
  fprintf(stderr, "Destroying backupfile\n");
  fclose(fp);

  /* Destroy uncertainlist */
  cer = certain;
  while (cer != NULL){
    del = cer;
    cer = cer->next;
    del->next = (certain_t *)0;
    free(del);
  }
  certain = (certain_t *)malloc(sizeof(certain_t));
  certain = (certain_t *)0;
}
```

```
  else {
    if ((fp = fopen(path, "w")) == NULL)
      return (−1);
    sprintf(path, "%s.%s", BACKUP_PATH, "tmp");
    if ((fp2 = fopen(path, "r")) == NULL)
      return (−1);
    while (!feof(fp) && fread((void ∗) &backup,
                              sizeof(struct backup), 1, fp2))
      if (fwrite((void ∗) &backup, sizeof(struct backup),
                 1, fp) < 1)
        fprintf(stderr, "fwrite error");
  }
  fclose(fp);
  fclose(fp2);
}
```

```
/*
 ***************************************************************
 *
 *      int send_resync(struct list *list, int type)
 *
 ***************************************************************
 * Description
 *      Sends a checksum of the database to the server along with
 *      the two serialnumbers.
 *
 * Input Parameters
 *      list - Pointer to the mainlist of all the entries.
 *      type - It's a resynchronize order.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns -1 if any errors occured.
 *
 * Called By
 *      main program.
 */
int send_resync(list_t *list, int type)
{
  mesg_t    mesg;
  int       reply;
  int       n;                 /* return value for select */
  struct    timeval   tv;      /* time to block with select */
  fd_set    rfds;
  unsigned  char    digest[16];/* Checksum digest */

  /* zero socket */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec = 0;
  tv.tv_usec = 0;

  while ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv))
        > 0)
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));

  mesg.type = type;
  strcpy(mesg.entr.user, username);

  if (list->cl_entries > 0) {
    md5_list(list, digest);
    strcpy(mesg.checksum, digest);
    mesg.cli_n = list->cl_entries;
    mesg.ser_n = list->se_entries;
  }
  else {
    mesg.cli_n = 1;
    mesg.ser_n = 0;
  }
```

```
  DgramSend(socket, server, sport,(void *) &mesg, sizeof(mesg));

  /* Wait for reply */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec  = TIMEOUT_SEC; /* Deadline is 1 sec */
  tv.tv_usec = TIMEOUT_USEC;

  if ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv)) == -1)
    fprintf(stderr, "Select failed\n");
  else if (n == 0 )
    fprintf(stderr, "Select timeout - no respone for server\n");
  else if (n > 0) {
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));
    fprintf(stderr, "reply: %d\n", reply);
    switch (reply) {
    case -1:
      fprintf(stderr, "Checksum match OK\n");
      return(OK);
    case -2:
      fprintf(stderr,
             "Checksum error. One or more entries are modified\n");
      return(-2);
    case -3:
      fprintf(stderr,
              "Checksum error. Too many entries in client\n");
      return(-3);
    case -4:
      fprintf(stderr,
              "Checksum error. Too few entries in client\n");
      return(-4);
    default:
      fprintf(stderr, "ERROR(%d) in send_resync()\n", reply);
      return(ERROR);
    }

  }
}
```

```
/*
 **************************************************************
 *
 *      int send_entry(mesg_t *mesg, BOOL backup)
 *
 **************************************************************
 * Description
 *      Send a entry to the server. If the server does not respond
 *      call the backup_entry function which saves the day (if the
 *      entry sent was not already a backup entry.
 *
 * Input Parameters
 *      mesg - the mesg to be sent.
 *      backup - is this a backup entry?
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns -1 if any errors occured.
 *
 * Called By
 *      main program and network functions.
 */
int send_entry(mesg_t mesg, BOOL backup)
{
  int       n;                  /* return value for select */
  struct    timeval   tv;       /* time to block with select */
  fd_set    rfds;
  int       reply;

  fprintf(stderr, "Sending entry: %s\n", mesg.entr.note);

  /* zero socket */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec = 0;
  tv.tv_usec = 0;

  while ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv))
        > 0)
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));

  if (((double)randint(0, 1000) / 1000.0) <= los_prob)
    /* Send order to server */
    DgramSend(socket, server, sport, (void *) &mesg, sizeof(mesg));
  else {
    fprintf(stderr,
          "Select timeout - no respone for server (fake)\n");
    backup_entry(mesg.entr, ADD);
    return(TIMEOUT);
  }

  /* Wait for reply */
  FD_ZERO(&rfds);
```

```
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec = TIMEOUT_SEC;
  tv.tv_usec = TIMEOUT_USEC;

  if ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv)) == -1)
    fprintf(stderr, "Select failed\n");
  else if (n == 0 ) {
    fprintf(stderr, "Select timeout - no respone for server\n");
    if (!backup)
      backup_entry(mesg.entr, ADD);
    return(TIMEOUT);
  }
  else if (n > 0) {
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));
    switch (reply) {
    case -11:
      fprintf(stderr, "Add order processed at server\n\n");
      return(OK);
    case -12:
      fprintf(stderr, "Entry allready at server\n\n");
      return(-12);
    default:
      fprintf(stderr, "ERROR(%d) in send_entry()\n", reply);
      return(ERROR);
    }
  }
}
```

```
/*
 ***************************************************************
 *
 *      entry_add(struct entry *entry, list_t *list)
 *
 ***************************************************************
 * Description
 *      Make a list over the entries that is sure to be in the
 *      client. Only update the certain server entries when an
 *      send_endry is successfull.
 *
 * Input Parameters
 *      entry - the entry to be sent.
 *      list - the serverlist to update
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns ERROR if any errors occured.
 *
 * Called By
 *      main program and network functions.
 */
void entry_add(struct entry *entry, list_t *list)
{
  entry_l_t *ep, *ep2;
  mesg_t     mesg;
  int        error;
  time_t     now;

  mashall_info(ADD, entry, &mesg);

  /* DEBUG */
  if (strcmp(mesg.entr.user, username) != 0) {
    fprintf(stderr, "Other user! (%s)\n", mesg.entr.user);
    fprintf(stderr, "  -note:%s\n", mesg.entr.note);
  }
  else {
    fprintf(stderr, "Own user! (%s)\n", mesg.entr.user);
    fprintf(stderr, "  -note:%s\n", mesg.entr.note);
  }

  ep = (entry_l_t *)malloc(sizeof(entry_l_t));
  ep->entr = mesg.entr;
  ep->next = (entry_l_t *)0;

  /* only link entries belonging to 'username' in the serverlist! */
  if (strcmp(mesg.entr.user, username) == 0) {
    /* don't send out of date entries */
    now = time(0);

    if (entry->time > now || entry->rep_last > now ||
        (entry->rep_every > 0 && entry->rep_last == 0) ||
        (entry->rep_weekdays > 0 && entry->rep_last == 0) ||
        entry->rep_yearly == TRUE) {
      /* future entry */
```

```
      /* empty */
      if (list->first == NULL)
        list->first = ep;
      /* first? */
      else if (strcmp(entry->note, list->first->entr.note) < 0 ||
               (strcmp(entry->note, list->first->entr.note) == 0 &&
                entry->time < list->first->entr.time)) {
        ep->next = list->first;
        list->first = ep;
      }
      /* search for right place */
      else {
        ep2 = list->first;

        while (ep2->next != NULL &&
               ((strcmp(entry->note, ep2->next->entr.note) > 0) ||
                (strcmp(entry->note, ep2->next->entr.note) == 0 &&
                 entry->time > ep2->next->entr.time)))
          ep2 = ep2->next;
        if (ep2->next == NULL)
          ep2->next = ep;
        else {
          ep->next = ep2->next;
          ep2->next = ep;
        }
      }
    }
  }
  error = send_entry(mesg, FALSE);

  switch (error) {
  case OK:
    list->cl_entries++;
    list->se_entries++;
    break;
  case -12:
    list->cl_entries++;
    list->se_entries++;
    break;
  case TIMEOUT:
    list->cl_entries++;
    break;
  default:
    list->cl_entries++;
    break;
  }
}
```

```
/*
 ******************************************************************
 *
 *      int entries_add(list_t *list)
 *
 ******************************************************************
 * Description
 *      Send all entries in a list to the server.
 *
 * Input Parameters
 *      list - Pointer to the mainlist of all the entries.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      None
 *
 * Called By
 *      main program.
 */
int entries_add(list_t *list)
{
  int       i;

  for (i = 0; i < mainlist->nentries; i++)
    if (mainlist->entry[i].user == NULL)
      entry_add(&mainlist->entry[i], list);
}
```

```
/*
 ******************************************************************
 *
 *      int get_user_entries(char *user)
 *
 ******************************************************************
 * Description
 *      Get all entries for an user from FR
 *
 * Input Parameters
 *      user - The name of the user.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      Return -1 if any error occured.
 *
 * Called By
 *      main program.
 */

int get_user_entries(char *user)
{
  int       n;                  /* return value for select */
  struct    timeval   tv;       /* time to block with select */
  fd_set    rfds;
  int       reply;
  mesg_t    mesg;

  mesg.type = GET_USER;
  strcpy(mesg.entr.user, user);

  fprintf(stderr, "get_user_entries: %s\n", user);

  /* zero socket */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec = 0;
  tv.tv_usec = 0;

  while ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv)) > 0)
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));

  if (((double)randint(0, 1000) / 1000.0) <= los_prob)
    /* Send order to server */
    DgramSend(socket, server, sport, (void *) &mesg, sizeof(mesg));
  else {
    fprintf(stderr, "Select timeout - no respone for server (fake)\n");
    return(TIMEOUT);
  }

  /* Wait for reply */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
```

```
  FD_SET(0, &rfds);

  tv.tv_sec = TIMEOUT_SEC;
  tv.tv_usec = TIMEOUT_USEC;

  if ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv)) == -1)
    fprintf(stderr, "Select failed\n");
  else if (n == 0 ) {
    fprintf(stderr, "Select timeout - no respone for server\n");
    return(TIMEOUT);
  }
  else if (n > 0) {
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));
    switch (reply) {
    case -31:
      fprintf(stderr, "Get_user order processed at server\n\n");
      return(-31);
    case -32:
      fprintf(stderr, "No user\n\n");
      return(-32);
    default:
      fprintf(stderr, "ERROR(%d) in get_user_entries()\n", reply);
      return(ERROR);
    }
  }
}
```

```
/*
 ****************************************************************
 *
 *      int get_server_entries(list_t *slist, char *user)
 *
 ****************************************************************
 * Description
 *      Get all entries in the server not heard of.
 *
 * Input Parameters
 *      slist - Pointer to the serverlist.
 *      user  - The user to get all entries for.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns -1 if any errors occured.
 *
 * Called By
 *      main program.
 */
int get_server_entries(list_t *slist, char *user)
{
  int      n;                 /* return value for select */
  struct   timeval   tv;      /* time to block with select */
  fd_set   rfds;
  mesg_t   mesg;
  struct entry *entry;
  struct lookup lookup;

  fprintf(stderr, "Getting entries for user %s.\n", user);

  while(TRUE) {
    /* zero socket */
    FD_ZERO(&rfds);
    FD_SET(socket, &rfds);
    FD_SET(0, &rfds);

    tv.tv_sec = 0;
    tv.tv_usec = 0;

    while ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv))
          > 0)
      DgramReceiveAny(socket, (void *) &mesg, sizeof(mesg));

    mesg.type = GET;
    strcpy(mesg.entr.user, user);

    DgramSend(socket, server, sport, (void *) &mesg, sizeof(mesg));

    /* Wait for reply */
    FD_ZERO(&rfds);
    FD_SET(socket, &rfds);
    FD_SET(0, &rfds);

    tv.tv_sec = TIMEOUT_SEC;
```

```
    tv.tv_usec = TIMEOUT_USEC;

    if ((n = select(socket+1, (int *) &rfds, NULL, NULL,
                    &tv)) == -1)
      fprintf(stderr, "Select failed\n");
    else if (n == 0 ) {
      fprintf(stderr, "Select timeout - no respone for server\n");
      return(TIMEOUT);
    }
    else if (n > 0) {
      DgramReceiveAny(socket, (void *) &mesg, sizeof(mesg));
      switch (mesg.type) {
      case -1:
        fprintf(stderr, "No more orders in server\n");
        return(OK);
      case 1:
        mainlist->locked++;
        fprintf(stderr, "Got entry from server\n");
        entry = (struct entry *)malloc(sizeof(struct entry));
        unmashall(mesg.entr, entry);
        if (find_entry(mesg.entr, slist) == FALSE){
          (void)add_entry(&mainlist, entry);
          sleep(TIMEOUT_SEC);
          rebuild_repeat_chain(mainlist);
          mainlist->locked--;
        }
        mainlist->modified = TRUE;
        break;
      case 2:
        mainlist->locked++;
        fprintf(stderr, "Got delete entry from server\n");
        entry = (struct entry *)malloc(sizeof(struct entry));
        unmashall(mesg.entr, entry);
        lookup_entry(&lookup, mainlist, entry->time, TRUE,
                     FALSE, '*');
        delete_entry(mainlist, lookup.index);
        rebuild_repeat_chain(mainlist);
        mainlist->locked--;
        mainlist->modified = TRUE;
        break;
      default:
        fprintf(stderr, "ERROR %d\n", mesg.type);
        return(ERROR);
      }
    }
  }
}
```

```
/************************************************************
 *
 *      BOOL find_entry(entry_t entry, list_t *list)
 *
 ************************************************************
 * Description
 *      Find an entry in the mainlist.
 *
 * Input Parameters
 *      entry - the entry to be found
 *      *list - pointer to the main list.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns TRUE if entry was found.
 *      otherwise FALSE.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
BOOL find_entry(entry_t entry, list_t *list)
{
  entry_l_t  *ep;

  ep = list->first;

  while (ep != NULL) {
    if (ep->entr.time == entry.time &&
        strcmp(ep->entr.note, entry.note) == 0) {
      return(TRUE);
    }
    else
      ep = ep->next;
  }
  return(FALSE);
}
```

```
/*
 ***************************************************************
 *
 *      int send_delete_entry(struct entry *entry, BOOL backup)
 *
 ***************************************************************
 * Description
 *      Send a entry to be deleted. Only back up entry if
 *      it is not already backed up.
 *
 * Input Parameters
 *      entry - entry to be deleted.
 *      backup - is the entry a backed up entry?
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns -1 if any errors occured.
 *
 * Globals Referenced
 *
 * Called By
 *      network function.
 */
int send_delete_entry(mesg_t mesg, BOOL backup)
{
  int       n;                    /* return value for select */
  struct    timeval   tv;         /* time to block with select */
  fd_set    rfds;
  int       reply;

  /* zero socket */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
  FD_SET(0, &rfds);

  tv.tv_sec = 0;
  tv.tv_usec = 0;

  while ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv))
         > 0)
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));

  if (((double)randint(0, 1000) / 1000.0) <= los_prob)
    /* Send order to server */
    DgramSend(socket, server, sport, (void *) &mesg, sizeof(mesg));
  else {
    fprintf(stderr,
            "Select timeout - no respone for server (fake)\n");
    backup_entry(mesg.entr, DELETE);
    return(TIMEOUT);
  }

  /* Wait for reply */
  FD_ZERO(&rfds);
  FD_SET(socket, &rfds);
```

```
  FD_SET(0, &rfds);

  tv.tv_sec = TIMEOUT_SEC;
  tv.tv_usec = TIMEOUT_USEC;

  if ((n = select(socket+1, (int *) &rfds, NULL, NULL, &tv)) == -1)
    fprintf(stderr, "Select failed\n");
  else if (n == 0 ) {
    fprintf(stderr, "Select timeout - no respone for server\n");
    if (!backup)
      backup_entry(mesg.entr, DELETE);
    return(TIMEOUT);
  }
  else if (n > 0) {
    DgramReceiveAny(socket, (void *) &reply, sizeof(reply));
    switch (reply) {
    case -21:
      fprintf(stderr, "Delete order processed in server\n");
      return(OK);
    case -22:
      fprintf(stderr, "Entry not in server\n\n");
      return(-22);
    default:
      fprintf(stderr, "ERROR\n");
      return(ERROR);
    }
  }
}
```

```
/*
 **************************************************************
 *
 *      void entry_delete(struct entry *entry, list_t *list)
 *
 **************************************************************
 * Description
 *      Send a entry to be deleted.
 *
 * Input Parameters
 *      entry - entry to be deleted.
 *      list - the mainlist
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      None
 *
 * Globals Referenced
 *
 * Called By
 *      main program.
 */
void entry_delete(struct entry *entry, list_t *list)
{
  int       error;
  mesg_t    mesg;
  time_t    now;

  fprintf(stderr, "Deleting entry: %s\n", entry->note);

  now = time(0);
  if (entry->time > now) {
    mashall_info(DELETE, entry, &mesg);

    error = send_delete_entry(mesg, FALSE);

    switch (error) {
    case OK:
      list->cl_entries--;
      list->se_entries--;
      break;
    case TIMEOUT:
    case -22:
      list->cl_entries--;
      break;
    default:
      list->cl_entries--;
      break;
    }
  }
  del_entry(mesg.entr, list);
}
```

```
/*
 ****************************************************************
 *
 *      int del_entry(entry_t entry, list_t *list)
 *
 ****************************************************************
 * Description
 *      Delete an entry from the mainlist.
 *
 * Input Parameters
 *      entry - the entry to be deleted
 *      *list - pointer to the mainlist.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns OK if action was succsessfull
 *      ERROR if not.
 *
 * Globals Referenced
 *
 * Called By
 *      network functions.
 */
int del_entry(entry_t entry, list_t *list)
{
  entry_l_t  *ep, *ep2;

  /* delete first */
  ep = list->first;

  if ((ep->entr.time == entry.time) &&
      (strcmp(ep->entr.note, entry.note) == 0)){
    list->first = ep->next;
    free(ep);
  }
  else {
    ep2 = list->first;
    ep = ep2->next;
    while (ep != NULL)
      if ((ep->entr.time == entry.time) &&
          (strcmp(ep->entr.note, entry.note) == 0)){
        ep2->next = ep->next;
        free(ep);
        return(OK);
      }
      else {
        ep2 = ep;
        ep = ep->next;
      }
  }
  return(ERROR);
}
```

Appendix D

Server files

D.1 server.c

```
/* -*-C-*-
******************************************************************
*
* File:        server.c
* RCS:         $Header: /usr/local/PASTA/cvs-rep/PASTA/
*                   src/diary/source/include/server.h
*                   v 1.10 1996/06/04 19:40:07 dago Exp $
* Author:      Dag-Frode Olsen
* Created:     Teu Aug  20 22:02:35 1996
* Language:    C
* Status:      Experimental (Do Not Distribute)
*
* (c) Copyright 1996, Dag-Frode Olsen, all rights reserved.
*
******************************************************************
*/

/*
******************************************************************
*
*       server.c
*
******************************************************************
* Description:
*       This file get orders from clients, checks that the client
*       was a legal one and calls different primitives to handle
*       the request.
*
*       The different orders are:
*       - resynchronize (1)
*       - add           (2)
*       - delete        (3)
*       - get_entries   (4)
*       - get_user      (5)
*
*/
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#include <time.h>
#include "conf.h"

#include "socklib.h"
#include "server.h"
#include "md5.h"

static BOOL         debug; /* print debugging information */

#define RESYNC_OK -1     /* checksum did match */
#define MODIFIED  -2     /* entries ok, but different ckeck */
#define TOO_MANY -3      /* too many entries in client */
#define TOO_FEW   -4     /* too few entries in client */

#define ADD_OK           -11    /* add order sucess */
#define ADD_ALREADY      -12    /* entry already in server */

#define DELETE_OK        -21    /* delete order sucess */
#define NO_ENTRY         -22    /* no entry to delete */

#define GET_USER_OK      -31    /* get user command ok */
#define NO_USER          -32    /* no such user */

#define SVR_ERROR        -99    /* general server failure */

const double los_prob = ERR_CHANCE;
```

```
/*
 *****************************************************************************
 *
 *      static int randint(int lo, int hi)
 *
 *****************************************************************************
 * Description
 *      Generate a random number beteween lo and hi.
 *
 * Input Parameters
 *      int lo - lower int
 *      int hi - higher int
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns a random number
 *
 * Called By
 *      server
 */
static int randint(int lo, int hi)
{
  static int first_call = TRUE;      /* Detects first call */

  /* seed */
  if (first_call) {
    first_call = FALSE;
    /* We want different seeds for
        different executions of the program */
    srand48((long)time(NULL));
  }

  return lo + (int)lrand48() % (hi - lo + 1);
}
```

```
main(int argc, char **argv)
{
  int           reply, type;
  int           csocket;              /* Server's socket */
  int           sport = PORT;         /* Server's port */
  int           i, client_nr;
  int           server_entries;
  mesg_t        mesg;                 /* The incoming message */
  client_t      clients[MAX_CLIENTS+2]; /* Two extra for other users and */
                                      /* own FR list */
  unsigned char digest[16];           /* The checksum digest */
  char          *from;                /* Name of the client machine */
  char          *owner;               /* Owner of the diary server */
  PID_T         pid;                  /* pid in lockfile */
  FILE          *fp;
  entry_l_t     *tmp;


  /* load vital data from file */
  if ((fp = fopen("serverfile", "r")) == NULL) {
    fprintf(stderr, "PANIC, could not read serverfile\n");
    exit(-1);
  }

  fscanf(fp, "USERS#%d\n", (int *) &clients[0].number);

  owner  = (char *)malloc(sizeof(char)*MAX_SIZE);
  fscanf(fp, "OWNER#%s\n", owner);

  for (i = 0; i < clients[0].number; i++) {
    clients[i].client_name = (char *)malloc(sizeof(char) * MAX_SIZE);
    fscanf(fp, "CLIENT#%s\n", clients[i].client_name);
  }

  fprintf(stderr, "*** clients: %d *** owner %s ***\nClients are: \n",
          clients[0].number, owner);
  for (i = 0; i < clients[0].number; i++)
    fprintf(stderr, " - %s\n", clients[i].client_name);
  fprintf(stderr, "\n");

  fclose(fp);


  /* make an extra list for other users */
  clients[MAX_CLIENTS].client_name = (char *)malloc(sizeof(char) * MAX_SIZE);
  strcpy(clients[MAX_CLIENTS].client_name, "otherlist");
  clients[MAX_CLIENTS].A_list = (list_t *)malloc(sizeof(list_t));
  clients[MAX_CLIENTS].A_list->first = (entry_l_t *)0;
  clients[MAX_CLIENTS].B_list = (list_t *)malloc(sizeof(list_t));
  clients[MAX_CLIENTS].B_list->first = (entry_l_t *)0;


  /* make an extra list for FR list */
  clients[MAX_CLIENTS+1].client_name = (char *)malloc(sizeof(char) * MAX_SIZE);
  strcpy(clients[MAX_CLIENTS+1].client_name, "FR_list");
  clients[MAX_CLIENTS+1].A_list = (list_t *)malloc(sizeof(list_t));
  clients[MAX_CLIENTS+1].A_list->first = (entry_l_t *)0;
```

```
clients[MAX_CLIENTS+1].B_list = (list_t *)malloc(sizeof(list_t));
clients[MAX_CLIENTS+1].B_list->first = (entry_l_t *)0;


/* init entry array */
for (i = 0; i < clients[0].number; i++) {
  clients[i].A_list = (list_t *)malloc(sizeof(list_t));
  clients[i].A_list->first = (entry_l_t *)0;
  clients[i].A_list->cl_entries = 0;
  clients[i].A_list->se_entries = 0;
  clients[i].B_list = (list_t *)malloc(sizeof(list_t));
  clients[i].B_list->first = (entry_l_t *)0;
  clients[i].B_list->cl_entries = 0;
  clients[i].B_list->se_entries = 0;
}


/* get the last current version from the FR */
if (extract_FR(owner) > 0) {
  fprintf(stderr, "extraxt_FR(owner) OK\n");
  /* read the server savefile from disk */
  for (i = 0; i < clients[0].number; i++)
    if (load_entry(clients[i].A_list, owner) < 0) {
      fprintf(stderr, "No entryfile.%s\n", owner);
      break;
    }
}
else
  fprintf(stderr, "extraxt_FR(owner) FAILED\n");


for (i = 0; i < clients[0].number; i++) {
  fprintf(stderr, "Client[%d].A_list\n", i);
  status(clients[i].A_list);
  fprintf(stderr, "Client[%d].B_list\n", i);
  status(clients[i].B_list);
}


/* Open an UDP socket */
if ((csocket = CreateDgramSocket(sport)) == -1)
  fprintf(stderr, "Could not create the client listen socket\n");


if (argc > 1 && !strcmp(argv[1], "-d")) {
  argv++;
  argc--;
  debug = TRUE;
}


if (!debug) {
  (void)umask(0077);
  setuid(getuid());
  setgid(getgid());
  if ((pid = fork()) > 0) {
    sleep(1);
```

```
      exit(0);
    }
  }


  from = (char *)malloc(sizeof(char)*MAX_SIZE);


  /* enter infinite loop */
  for (;;) {
    /* Block & waits for a request from a client */
    DgramReceiveAny(csocket, (void *) &mesg, sizeof(mesg));


    from[0] = '\0';
    from = DgramLastname();


    client_nr = −1;
    for (i = 0; i < clients[0].number; i++)
      if (strcmp(from, clients[i].client_name) == 0) {
        type = mesg.type;
        client_nr = i;
        break;
      }
      else
        type = ERROR;

    fprintf(stderr, "\nGot order: %d - from %s(%d)\n", type,
            mesg.entr.user, client_nr);


    /* do the specific task */
    switch (type) {
```

```
case 1:          /* resynchronize */
  /* 1. Get the current file from the server
   * 2. Calculate the md5 checksum of the client's
   *    serverlist.
   * 3. Compare it with the checksum that the client sent.
   *    if the checksum match all is ok
   *    else return a error mesg depending on wheter there
   *         are too few or too many entries in the server
   *         compared to the client.
   */

  /* get the last current version from the FR */
  if (extract_FR(owner) > 0) {
    fprintf(stderr, "extraxt_FR(owner) OK\n");
    /* read the server savefile from disk */
    for (i = 0; i < clients[0].number; i++) {
      if (load_entry(clients[MAX_CLIENTS+1].A_list, owner) < 0) {
        fprintf(stderr, "No entryfile.%s\n", owner);
        break;
      }
      else {
        /* Check if the clients[MAX_CLIENTS+1].A_list have any */
        /* entries not in the current A_list. If yes, put it in the */
        /* B_list */
        tmp = clients[MAX_CLIENTS+1].A_list->first;
        if (find_entry(tmp->entr, clients[client_nr].A_list,
                       FALSE) == FALSE) {
          if (add_entry(FALSE, tmp->entr,
                        clients[client_nr].B_list) == -1 )
            fprintf(stderr, "add_entry(): Error\n");
        }
        else
          fprintf(stderr, "Entry already in A_list - skip it\n");

        /* Delete the entry from the list, need a cleen list next time */
        if (delete_entry(tmp->entr,
                         clients[MAX_CLIENTS+1].A_list) == -1)
          fprintf(stderr, "delete_entry(): Error\n");
      }
    }
  }
  else
    fprintf(stderr, "extraxt_FR(owner) FAILED\n");


  /* check if the checksums matches */
  fprintf(stderr, "%s checksum: ", clients[client_nr].client_name);
  for (i = 0; i < 16; i++)
    fprintf (stderr, "%02x", mesg.checksum[i]);
  fprintf (stderr, " %d (%d)\n", mesg.cli_n, mesg.ser_n);

  md5_list(clients[client_nr].A_list, digest);

  fprintf(stderr, "Server chesksum:        ");
  for (i = 0; i < 16; i++)
    fprintf (stderr, "%02x", digest[i]);
```

```
server_entries = 0;
clients[client_nr].A_list->se_entries = 0;
clients[client_nr].B_list->se_entries = 0;

tmp = clients[client_nr].A_list->first;
while (tmp != NULL) {
  clients[client_nr].A_list->se_entries++;
  tmp = tmp->next;
}

tmp = clients[client_nr].B_list->first;
while (tmp != NULL) {
  clients[client_nr].B_list->se_entries++;
  tmp = tmp->next;
}

server_entries = clients[client_nr].A_list->se_entries +
  clients[client_nr].B_list->se_entries;

fprintf (stderr, " %d (%d)\n", server_entries,
         clients[client_nr].A_list->se_entries);


if (strncmp(mesg.checksum, digest, 16) == 0) {
  reply = RESYNC_OK;

  /* Resync ok so save entryfile and and put entryfile into FR */
  if (save_entry(clients[client_nr].A_list) < 0)
    fprintf(stderr, "ERROR: save_entry() failed\n");
  else if (insert_FR(clients[client_nr].A_list->first->entr.user) < 0)
    fprintf(stderr, "ERROR: insert_FR() failed\n");

  if (mesg.cli_n < server_entries)
    reply = TOO_FEW;
}
else if (mesg.cli_n == server_entries)
  reply = MODIFIED;
else if (mesg.cli_n > server_entries)
  reply = TOO_MANY;
else if (mesg.cli_n < server_entries)
  reply = TOO_FEW;
else
  reply = SVR_ERROR;


fprintf(stderr, "\n********RESYNCH*********\n");
for (i = 0; i < clients[0].number; i++) {
  fprintf(stderr, "Client[%d].A_list\n", i);
  status(clients[i].A_list);
  fprintf(stderr, "Client[%d].B_list\n", i);
  status(clients[i].B_list);
}
break;
```

```
case 2:                  /* add */
  /* 1. Check if the entry exists as a delete entry in the
   *    B_list.
   * If yes
   *    delete entry in all the B_lists and reply
   * if no
   *    check if the entry exists in the A_list.
   *    if yes
   *       reply that the entry alredy exists.
   *    if no
   *       insert the entry in the A_list and in all the
   *       other B_lists if the entry does not exists in
   *       that client's A_list. If the entry exists in
   *       the B_list, remove it from the B_list. Insert
   *       in the FR.
   */


  /* check if owner is calling */
  if (strcmp(mesg.entr.user, owner) == 0) {
    fprintf(stderr, "Own user\n");

    if (find_entry(mesg.entr, clients[client_nr].B_list,
                   TRUE) == TRUE ) {
      fprintf(stderr, "This entry is due to be removed\n");
      /* remove the entry from all B_lists */
      for (i = 0; i < clients[0].number; i++)
        if (delete_entry(mesg.entr, clients[i].B_list) == -1) {
          fprintf(stderr, "delete_entry(): Error\n");
          reply = SVR_ERROR;
        }
      reply = ADD_OK;
    }


    /* check if the entry exists in the A_list */
    else if (find_entry(mesg.entr, clients[client_nr].A_list,
                        FALSE) == TRUE ) {
      reply = ADD_ALREADY;
      fprintf(stderr, "Entry allready in server - skip it\n");
    }
    else {
      fprintf(stderr, "! found %s save it in %s\n",
              mesg.entr.note, clients[client_nr].client_name);
      if (add_entry(FALSE, mesg.entr, clients[client_nr].A_list) == -1 ) {
        fprintf(stderr, "add_entry(): Error\n");
        reply = SVR_ERROR;
      }
      else {
        reply = ADD_OK;


        /* delete entry from B_list if it's there */
        if (find_entry(mesg.entr, clients[client_nr].B_list,
                       FALSE) == TRUE)
          if (delete_entry(mesg.entr,
```

```
                         clients[client_nr].B_list) == -1) {
          fprintf(stderr, "delete_entry(): Error\n");
          reply = SVR_ERROR;
        }


      /* Then check if the other clients has it, or add it */
      for (i = 0; i < clients[0].number; i++)
        if (i != client_nr)
          if (find_entry(mesg.entr, clients[i].A_list,
                         FALSE) == FALSE ) {
            fprintf(stderr, "! found %s save it in %s\n",
                    mesg.entr.note, clients[i].client_name);
            if (find_entry(mesg.entr, clients[i].B_list,
                           FALSE) == FALSE ) {
              if (add_entry(FALSE, mesg.entr, clients[i].B_list) == -1) {
                fprintf(stderr, "add_entry(): Error\n");
                reply = SVR_ERROR;
              }
              else
                reply = ADD_OK;
            }
            else
              reply = ADD_OK;
          }
    }
  }
}
/*          add order from other user
 * first check if the entry is in the entrylist
 * - if yes
 *   ok
 * - if not
 *   add it to the otherlist and write the otherlist
 *   back into the FR
 */
else {
  fprintf(stderr, "Other user %s\n", mesg.entr.user);
  if (find_entry(mesg.entr, clients[MAX_CLIENTS].A_list,
                 FALSE) == TRUE ) {
    reply = ADD_OK;
    fprintf(stderr, "Entry (%s) is in A_list\n", mesg.entr.note);
  }
  else {
    fprintf(stderr, "Entry is NOT! in A_list\n");
    if (add_entry(FALSE, mesg.entr, clients[MAX_CLIENTS].A_list) == -1) {
      fprintf(stderr, "add_entry(): Error\n");
      reply = SVR_ERROR;
    }
    else {
      /* save and insert entries in FR */
      if (save_entry(clients[MAX_CLIENTS].A_list) < 0) {
        fprintf(stderr, "save_entry(): Error\n");
        reply = SVR_ERROR;
      }
      else if (insert_FR(mesg.entr.user) < 0) {
        fprintf(stderr, "insert_FR(): Error\n");
```

```
        reply = SVR_ERROR;
      }
      else
        reply = ADD_OK;
    }
  }
}


fprintf(stderr, "\n**********ADD***********\n");
for (i = 0; i < clients[0].number; i++) {
  fprintf(stderr, "Client[%d].A_list\n", i);
  status(clients[i].A_list);
  fprintf(stderr, "Client[%d].B_list\n", i);
  status(clients[i].B_list);
}
break;
```

```
case 3:              /* delete */
  /* First check if the entry exists in the A_list.
   * If yes
   *   delete entry in the A_lists and reply
   *   add a delete entry to all other client's B_lists.
   *   if the entry exists as an add entry in the B_list
   *   delete it.
   *   Insert A_list in FR.
   * if no
   *   reply error (not supported by plan)
   */


  /* check if the entry exists in the A_list */
  if (find_entry(mesg.entr, clients[client_nr].A_list,
                 FALSE) == FALSE) {
    fprintf(stderr, "! found - not deleted\n");
    reply=NO_ENTRY;
  }
  else if (delete_entry(mesg.entr,
                        clients[client_nr].A_list) == -1) {
    fprintf(stderr, "delete_entry: Error\n");
    reply = SVR_ERROR;
  }
  else {
    fprintf(stderr, "removed entry(): %s\n", mesg.entr.note);
    reply = DELETE_OK;


    /* delete ok so save entryfile and and put entryfile into FR */
    if (save_entry(clients[client_nr].A_list) < 0)
      fprintf(stderr, "ERROR: save_entry() failed\n");
    else if (insert_FR(clients[client_nr].A_list->first->entr.user) < 0)
        fprintf(stderr, "ERROR: insert_FR() failed\n");


    /* add a delete entry to all other client's B_lists */
    for (i = 0; i < clients[0].number; i++)
      if (i != client_nr)
        /* if the entry exists as an add entry, remove it */
        if (find_entry(mesg.entr, clients[i].B_list, FALSE) == TRUE) {
          if (delete_entry(mesg.entr, clients[i].B_list) == -1) {
            fprintf(stderr, "delete_entry: Error\n");
            reply = SVR_ERROR;
          }
        }
    /* if the entry already exists as a delete entry, do nothing. */
        else if (find_entry(mesg.entr, clients[i].B_list, TRUE) == FALSE)
          if (add_entry(TRUE, mesg.entr, clients[i].B_list) == -1) {
            fprintf(stderr, "add_entry(): Error\n");
            reply = SVR_ERROR;
          }
    fprintf(stderr, "finished deleting\n");
  }


  fprintf(stderr, "\n********DELETE*********\n");
```

```
for (i = 0; i < clients[0].number; i++) {
  fprintf(stderr, "Client[%d].A_list\n", i);
  status(clients[i].A_list);
  fprintf(stderr, "Client[%d].B_list\n", i);
  status(clients[i].B_list);
}
break;
```

```
case 4:          /* get entries */
  /* If there exists entries in the B_list
   *   send first entry
   *   remove it from B_list
   */

  /* mesg.entr.user decides which list to use */
  fprintf(stderr, "Get_entries order got for user %s\n", mesg.entr.user);
  if (strcmp(mesg.entr.user, owner) == 0)
    /* own user */
    if (clients[client_nr].B_list->first == NULL) {
      mesg.type = -1;
      fprintf(stderr, "No entry to send. (B_list empty)\n");
      DgramReply(csocket, (void *) &mesg, sizeof(mesg));
    }
    else {
      if (clients[client_nr].B_list->first->delete == FALSE)
        mesg.type = 1;
      else
        mesg.type = 2;


      mesg.entr = clients[client_nr].B_list->first->entr;


      fprintf(stderr, "Sending entry %s to client\n", mesg.entr.note);
      DgramReply(csocket, (void *) &mesg, sizeof(mesg));


      if (delete_entry(mesg.entr, clients[client_nr].B_list) == -1)
        fprintf(stderr, "delete_entry(): Error\n");
    }
  else
    /* other user */
    if (clients[MAX_CLIENTS].B_list->first == NULL) {
      mesg.type = -1;
      fprintf(stderr, "No entry to send. (otherlist empty)\n");

      DgramReply(csocket, (void *) &mesg, sizeof(mesg));
    }
    else {
      mesg.type = 1;
      mesg.entr = clients[MAX_CLIENTS].B_list->first->entr;


      fprintf(stderr, "Sending entry %s to client\n", mesg.entr.note);
      DgramReply(csocket, (void *) &mesg, sizeof(mesg));


      if (delete_entry(mesg.entr, clients[MAX_CLIENTS].B_list) == -1)
        fprintf(stderr, "delete_entry(): Entry\n");
    }


  fprintf(stderr, "\n*******GET_ENTRIES******\n");
  for (i = 0; i < clients[0].number; i++) {
```

```
    fprintf(stderr, "Client[%d].A_list\n", i);
    status(clients[i].A_list);
    fprintf(stderr, "Client[%d].B_list\n", i);
    status(clients[i].B_list);
  }
  break;
```

```
    case 5:          /* get user */
      /* Get the entryfile belonging to the user from the FR
       * and put them in the otherlist.
       */


      fprintf(stderr, "Get_user order\n");
      if (extract_FR(mesg.entr.user) > 0) {
        fprintf(stderr, "extraxt_FR(%s) OK\n", mesg.entr.user);
        if ((load_entry(clients[MAX_CLIENTS].A_list, mesg.entr.user)) ||
            (load_entry(clients[MAX_CLIENTS].B_list, mesg.entr.user)) < 0) {
          fprintf(stderr, "No server entryfile for user '%s',\n",
                  mesg.entr.user);
          reply = NO_USER;
        }
        else
          reply = GET_USER_OK;
      }
      else {
        fprintf(stderr, "extraxt_FR(): Error\n");
        reply = NO_USER;
      }


      fprintf(stderr, "\n******GET_USER*******\n");
      for (i = 0; i < clients[0].number; i++) {
        fprintf(stderr, "Client[%d].A_list\n", i);
        status(clients[i].A_list);
        fprintf(stderr, "Client[%d].B_list\n", i);
        status(clients[i].B_list);
      }
      break;


    default:
      fprintf(stderr, "ERROR IN MESG TYPE!\n");
      break;
    }


    /* Send reply to client */
    if (type != 4)
      if (((double)randint(0, 1000) / 1000.0) <= los_prob)
        /* Send reply to client */
        DgramReply(csocket, (void *) &reply, sizeof(reply));
      else
        fprintf(stderr, "No reply sent to client (fake)\n");
  }
}
```

D.2 primitives.c

```
/* -*-C-*-
****************************************************************
*
* File:         primitives.c
* RCS:          $Header: /usr/local/PASTA/cvs-rep/PASTA/
*                   src/diary/source/include/server.h
*                   v 1.10 1996/06/04 19:40:07 dago Exp $
* Author:       Dag-Frode Olsen
* Created:      Tue Aug  20 22:02:35 1996
* Language:     C
* Status:       Experimental (Do Not Distribute)
*
* (c) Copyright 1996, Dag-Frode Olsen, all rights reserved.
*
****************************************************************
*/

/*
****************************************************************
*
*       server.c
*
****************************************************************
* Description:
*       This file has all the functions the server uses to do
*       orders from the clients.
*
*       They are:
*       - add_entry - adds an entry to the right list.
*       - find_entry - finds an entry in a list.
*       - delete_entry - deletes an entry from a list.
*       - md5_list - make the md5 checksum of a list.
*/
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include "conf.h"
#include "server.h"
#include "version.h"
#include "md5.h"
#include "fr.h"
#include "frlib.h"

extern char          *progname;    /* argv[0] */


char    owner[MAX_SIZE]    = "dago";     /* owner of the diary server */
char    filename[MAX_SIZE] = "entryfile";

void print_command(int c, FILE *fd)
{
  switch (c)  {
```

```
  case INSERT:
    fprintf(fd, "INSERT ");
    break;
  case INSERT_A:
    fprintf(fd, "INSERT_A ");
    break;
  case EXTRACT_P:
    fprintf(fd, "EXTRACT_P ");
    break;
  case EXTRACT_O:
    fprintf(fd, "EXTRACT_O ");
    break;
  case EXTRACT_S:
    fprintf(fd, "EXTRACT_S ");
    break;
  case DELETE:
    fprintf(fd, "DELETE ");
    break;
  case FLUSH:
    fprintf(fd, "FLUSH ");
    break;
  case REFRESH:
    fprintf(fd, "REFRESH ");
    break;
  case RELEASE:
    fprintf(fd, "RELEASE ");
    break;
  case STAT:
    fprintf(fd, "STAT ");
    break;
  case STAT_LOCAL:
    fprintf(fd, "STAT_LOCAL ");
    break;
  default:
    fprintf(fd, "illegal command ");
  }
}

void check_owner(struct FR *FR)
{
    char *o;
    o = getenv("USER");
    sprintf(FR->owner, "%s", o);

/* fprintf(stderr, “check_owner(): FR->owner = %s, o = %s\n”, FR->owner, o); */
}

void print_meta_data(FILE *f, struct FR *FR)
{
    fprintf(f, "Accumulated meta data\n");
    fprintf(f, "\ttimestamp: %s", FR->timestamp);
    fprintf(f, "\tserver_filename: %s\n", FR->server_filename);
    fprintf(f, "\tlocal_filename: %s\n", FR->local_filename);
    fprintf(f, "\towner: %s", FR->owner);
    fprintf(f, "\tsize: %d\n", FR->size);
    fprintf(f, "\tquality: 0x%x\n", FR->quality);
    fprintf(f, "\tlast_cico: ");  print_command(FR->last_cico, f);
```

```
    fprintf(f, "\n");
    fprintf(f, "\tlast_cmd: ");  print_command(FR->last_cmd, f);
    fprintf(f, "\n");
    fprintf(f, "\tlocked: %d\n", FR->locked);
    fprintf(f, "\tserver_time: %ld\n", FR->server_time);
    fprintf(f, "\tmeta_string: %s\n", FR->meta_string);
}
```

```
/*
 ***************************************************************
 *
 *      int load_entry(list_t *list, list_t *list2, char *user)
 *
 ***************************************************************
 * Description
 *      Load the file into the mainlist. Only add entry if reference
 *      list do not have the entry.
 *
 * Input Parameters
 *      None
 *
 * Output Parameters
 *      list  - The mainlist
 *      list2 - A reference list.
 *      user  - name of the user to load.
 *
 * Return Value
 *      returns ERROR if any errors occured.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int load_entry(list_t *list, char *user)
{
  FILE *fp;
  entry_t entry;
  char file[MAX_SIZE];

  /* make the filename */
  strcpy(file, filename);
  strcat(file, ".");
  if (strcmp(user, owner) == 0)
    strcat(file, owner);
  else
    strcat(file, user);

  if ((fp = fopen(file, "r")) == NULL)
    return(ERROR);

  while (!feof(fp) && fread((void *) &entry, sizeof(entry_t), 1, fp))
      add_entry(FALSE, entry, list);

  fprintf(stderr, "File loaded %s\n", file);

  fclose(fp);
  return(OK);
}
```

```
/*
 ***************************************************************
 *
 *      int save_entry(list_t *list)
 *
 ***************************************************************
 * Description
 *      Save a list to a file.
 *
 * Input Parameters
 *      list - A pointer to the list to save.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns ERROR if any errors occured.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int save_entry(list_t *list)
{
  FILE *fp;
  entry_l_t *ep;
  char file[MAX_SIZE];

  strcpy(file, filename);
  strcat(file, ".");
  strcat(file, list->first->entr.user);

  fprintf(stderr, "Saving file %s\n", file);

  if ((fp = fopen(file, "w")) == NULL)
    return(ERROR);

  /* Shoud not be empty! */
  if (list->first == NULL) {
    fprintf(stderr, "save_entry(): list empty\n");
    fclose(fp);
    return(ERROR);
  }

  /* write all entries */
  ep = list->first;
  while (ep != NULL) {
    fprintf(stderr, "writing entry %s\n", ep->entr.note);
    if (fwrite((void *) &ep->entr, sizeof(entry_t), 1, fp) < 1)
      fprintf(stderr, "fwrite error");

    ep = ep->next;
  }

  fclose(fp);
```

```
        }
```

```
/*
 ****************************************************************
 *
 *      int add_entry(BOOL delete, entry_t entry, list_t *list)
 *
 ****************************************************************
 * Description
 *      Add an entry to the mainlist, and to the server file.
 *
 * Input Parameters
 *      delete - the entry is to be deleted.
 *      entry - the entry to be saved.
 *      list - the mainlist.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns ERROR if any errors occured.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int add_entry(BOOL delete, entry_t entry, list_t *list)
{
  entry_l_t *ep, *ep2;

  ep = (entry_l_t *)malloc(sizeof(entry_l_t));
  ep->entr = entry;
  ep->next = (entry_l_t *)0;
  ep->delete = delete;

  /* empty */
  if (list->first == NULL)
    list->first = ep;
  /* first? */
  else if (strcmp(entry.note, list->first->entr.note) < 0 ||
           (strcmp(entry.note, list->first->entr.note) == 0 &&
            entry.time < list->first->entr.time)) {
    ep->next = list->first;
    list->first = ep;
  }
  /* search for right place */
  else {
    ep2 = list->first;
    while (ep2->next != NULL &&
           ((strcmp(entry.note, ep2->next->entr.note) > 0) ||
            (strcmp(entry.note, ep2->next->entr.note) == 0 &&
             entry.time > ep2->next->entr.time)))
      ep2 = ep2->next;
    if (ep2->next == NULL)
      ep2->next = ep;
    else {
      ep->next = ep2->next;
      ep2->next = ep;
```

```
    }
  }
  /* list->se_entries++; */

  return(OK);
}
```

```
/*
 *************************************************************
 *
 *      int insert_FR(char *user)
 *
 *************************************************************
 * Description
 *      Write file entryfile to the *user directory in the FR
 *      First delete the file in FR because of paranoia!
 *
 * Input Parameters
 *      user  - The actual user
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns OK if everything was OK
 *      otherwise ERROR.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int insert_FR(char *user)
{
  struct FR *FR;
  char localfile[MAX_SIZE], serverfile[MAX_SIZE];
  int i, j;

  strcpy(localfile, filename);
  strcat(localfile, ".");
  strcat(localfile, user);
  strcpy(serverfile, "/");
  strcat(serverfile, user);
  strcat(serverfile, "/");
  strcat(serverfile, filename);

  fprintf(stderr, "localfile: %s, serverfile:%s\n", localfile, serverfile);

  FR = FR_open(localfile, serverfile, INSERT);
  if (FR == NULL)  {
    fprintf(stderr, "INSERT of '%s' failed. Client error!\n", localfile);
    return(ERROR);
  }

  check_owner(FR);

  /* Insert the file */
  if ((i = FR_insert_file(FR, FALSE)) < 0) {
    fprintf(stderr, "Failed INSERT-ing '%s'\n", localfile);
    fprintf(stderr, "FR->frtp_status %d\n", FR->frtp_status);
    if (FR->frtp_status == 500)
      fprintf(stderr, "UNKNOWN Command. Socket fuckup!\n");
  }
  else {
```

```
    fprintf(stderr, "Inserting of file in user %s's repository OK\n", user);
    (void) FR_update(FR);
  }

  (void) FR_close(FR, FALSE);

  if (FR->locked) {
    fprintf(stderr, "File is still LOCKED!\n");

    FR = FR_open(NULL, serverfile, RELEASE);
    if (FR == NULL)  {
      fprintf(stderr, "RELEASE on '%s' failed. Client error!\n", serverfile);
      return(-1);
    }

    /* Release the lock if any*/
    j = FR_lock_release_file(FR, FALSE);
    if (j < 0)
      fprintf(stderr, "Error RELEASE-ing lock on '%s'\n", serverfile);
    else
      (void) FR_update(FR);

    (void) FR_close(FR, FALSE);
  }

  return(i);
}
```

```
/*
 ***************************************************************
 *
 *      int extract_FR(char *user)
 *
 ***************************************************************
 * Description
 *      Get the last appointmentfile for user.
 *
 * Input Parameters
 *      user  - The actual user
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns OK if everything was OK
 *      otherwise ERROR.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int extract_FR(char *user)
{
  struct FR *FR;
  char       *localfile;
  char       *serverfile;
  int        i, j;

  localfile = (char *)malloc(sizeof(char)*MAX_SIZE);
  serverfile = (char *)malloc(sizeof(char)*MAX_SIZE);

  strcpy(localfile, filename);
  strcat(localfile, ".");
  strcat(localfile, user);
  strcpy(serverfile, "/");
  strcat(serverfile, user);
  strcat(serverfile, "/");
  strcat(serverfile, filename);

  fprintf(stderr, "local: %s, remote: %s\n", localfile, serverfile);

/*   if (meta_dir_init() < 0)  */
/*      fprintf(stderr, “failure of meta data\n”); */

  FR = FR_open(localfile, serverfile, EXTRACT_P);
  if (FR == NULL) {
    fprintf(stderr, "EXTRACT of '%s' failed. Client error!\n", serverfile);
    return(-1);
  }

  strcpy(FR->local_filename, localfile);

  if ((i = FR_extract_file(FR, FALSE)) < 0)
    fprintf(stderr, "Error EXTRACT-ing '%s'\n", serverfile);
```

```
  else {
    fprintf(stderr, "Extract of file in user %s's repository OK\n", user);
    (void) FR_update(FR);
  }

  (void) FR_close(FR, FALSE);

  if (FR->locked) {
    fprintf(stderr, "File is still LOCKED!\n");

    FR = FR_open(NULL, serverfile, RELEASE);
    if (FR == NULL)  {
      fprintf(stderr, "RELEASE on '%s' failed. Client error!\n", serverfile);
      return(-1);
    }

    /* Release the lock */
    j = FR_lock_release_file(FR, FALSE);
    if (j < 0)  {
      fprintf(stderr, "Error RELEASE-ing lock on '%s'\n", serverfile);
    } else
      (void) FR_update(FR);

    (void) FR_close(FR, FALSE);
  }

  return(i);
}
```

```
/*
 ****************************************************************
 *
 *      BOOL find_entry(entry_t entry, list_t *list, BOOL delete)
 *
 ****************************************************************
 * Description
 *      Find an entry in the mainlist.
 *
 * Input Parameters
 *      entry  - the entry to be found
 *      *list  - pointer to the main list.
 *      delete - is it a delete entry?
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns TRUE if entry was found.
 *      otherwise FALSE.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
BOOL find_entry(entry_t entry, list_t *list, BOOL delete)
{
  entry_l_t  *ep;

  ep = list->first;

  while (ep != NULL) {
    if (ep->entr.time == entry.time &&
        strcmp(ep->entr.note, entry.note) == 0
        && ep->delete == delete) {
      return(TRUE);
    }
    else
      ep = ep->next;
  }
  return(FALSE);
}
```

```
/*
 ****************************************************************
 *
 *      int delete_entry(entry_t entry, list_t *list)
 *
 ****************************************************************
 * Description
 *      Delete an entry from a list.
 *
 * Input Parameters
 *      entry - the entry to be deleted
 *      *list - pointer to the mainlist.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      returns OK if action was succsessfull
 *      ERROR if not.
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int delete_entry(entry_t entry, list_t *list)
{
  entry_l_t  *ep, *ep2;

  /* delete first */
  ep = list->first;

  if ((ep->entr.time == entry.time) &&
      (strcmp(ep->entr.note, entry.note) == 0)){
    list->first = ep->next;
    free(ep);
/*      list->se_entries–; */
    return(OK);
  }
  else {
    ep2 = list->first;
    ep = ep2->next;
    while (ep != NULL)
      if ((ep->entr.time == entry.time) &&
          (strcmp(ep->entr.note, entry.note) == 0)){
        ep2->next = ep->next;
        free(ep);
/*      list->se_entries–; */
        return(OK);
      }
      else {
        ep2 = ep;
        ep = ep->next;
      }
  }
  return(ERROR);
}
```

```
/*
 *****************************************************************
 *
 *      int md5_list(list_t *list, unsigned char digest[])
 *
 *****************************************************************
 * Description
 *      Make a md5 checksum of the mainlist. Remove any old
 *      entries from the list.
 *
 * Input Parameters
 *      *list - pointer to the mainlist.
 *      digest[] - the checksum digest.
 *
 * Output Parameters
 *      None
 *
 * Return Value
 *      none
 *
 * Globals Referenced
 *
 * Called By
 *      Main program.
 */
int md5_list(list_t *list, unsigned char digest[])
{
  MD5_CTX    context;
  entry_l_t *ep, *ep2;
  time_t     now;

  now = time(0);

  MD5Init (&context);

  /* check if entry is out of time */
  ep = list->first;
  if (!(ep->entr.time > now || ep->entr.rep_last > now ||
      (ep->entr.rep_every > 0 && ep->entr.rep_last == 0) ||
      (ep->entr.rep_weekdays > 0 && ep->entr.rep_last == 0) ||
      ep->entr.rep_yearly == TRUE)) {
    list->first = ep->next;
    free(ep);
  }
  else {
    ep2 = list->first;
    ep = ep2->next;
    while (ep != NULL)
      if (!(ep->entr.time > now || ep->entr.rep_last > now ||
            (ep->entr.rep_every > 0 && ep->entr.rep_last == 0) ||
            (ep->entr.rep_weekdays > 0 &&
             ep->entr.rep_last == 0) ||
            ep->entr.rep_yearly == TRUE)) {
        ep2->next = ep->next;
        free(ep);
      }
      else {
```

```
        ep2 = ep;
        ep = ep->next;
      }
  }


  ep = list->first;
  while (ep != NULL) {
    MD5Update (&context, (unsigned char *)&ep->entr.user,
               strlen(ep->entr.user));
    MD5Update (&context, (unsigned char *)&ep->entr.message,
               strlen(ep->entr.message));
    MD5Update (&context, (unsigned char *)&ep->entr.script,
               strlen(ep->entr.script));
    MD5Update (&context, (unsigned char *)&ep->entr.note,
               strlen(ep->entr.note));
    MD5Update (&context, (unsigned char *)&ep->entr.time,
               sizeof(ep->entr.time));
    MD5Update (&context, (unsigned char *)&ep->entr.length,
               sizeof(ep->entr.length));
    ep = ep->next;
  }


  MD5Final(digest, &context);
}


void status(list_t *list)
{
  entry_l_t *tmp;
  int        i = 0;

  tmp = list->first;

  fprintf(stderr, "ENTRY#    NOTE:    DELETE:\n");
  while (tmp != NULL) {
    fprintf(stderr, "   %d   ", i);
    i++;
    fprintf(stderr, "%s", tmp->entr.note);
    fprintf(stderr, "%s", (tmp->delete) ? " TRUE\n" : " FALSE\n");

    tmp = tmp->next;
  }
  fprintf(stderr, "\n");
}
```