A Highly Available Application in the Transis Environment

Ofir Amir, Yair Amir and Danny Dolev
Institute of Computer Science
The Hebrew University of Jerusalem,Israel
E-mail: ofiramir, yairamir, dolev @cs.huji.ac.il

Abstract. This paper presents a typical replicated application in a distributed system. The application was developed on top of Transis, a reliable and efficient transport layer protocol. The basic properties of the protocol and the advantages of using Transis as the transport layer are discussed. The algorithms used in this application can lead to better solutions in the area of distributed transaction systems and replicated databases.

1 Introduction

The reliability and availability of loosely coupled distributed systems is becoming a requirement for many computer systems. One of the main infrastructure services is a distributed database. This paper addresses some aspects of information dissemination within such a service in a dynamic loosely coupled environment.

We chose to tackle this complex problem by studying a particular application. The application we focus on is a Replicated Mail Service. In our design of this service, the database of messages is replicated among several mail servers. Each client can send mail, read it or delete it only when it can connect to one of the servers. A distributed transaction mechanism is used to ensure the eventual consistency of the replicated service.

The Transis project, currently under development at the Hebrew University of Jerusalem, was designed to supply reliable and efficient transport layer services for distributed systems. Transis provides fast and reliable message multicast between all currently connected processors, (despite a possible unreliability of the network). Transis offers several service levels for message ordering, which is a key primitive in managing concurrency in distributed computing. Transis achieves good performance by using the network's basic unreliable broadcast service, it recovers messages that were lost due to omission faults, and it handles processor crashes and dynamic partitions of the network. The Replicated Mail Service presented in this paper utilizes the ability of Transis to keep track of the current membership of connected processors and to efficiently disseminate messages among those processors.

One can approach the consistency problem using the traditional two-phase-commit method. Unfortunately, this will introduce difficulties whenever a connection is unavailable. In many application domains this method is not feasible, since blocking or transaction aborting is not acceptable as a legitimate solution in those domains. Other traditional methods (listed later) impose rigid constraints on the solution.

The main property of our solution is the guarantee that if an eventual path between the source and the target exists, the information will reach the target. By eventual path we don't mean a continuous connection, but rather that pairs of processors along the path were connected at successive periods of time. For each of the processors, the algorithm implicitly builds its knowledge about the status of each message at other processors. This knowledge enables us to efficiently handle all the mail services.

We suggest that the methods used to build this mail service can lead to better solutions in the area of distributed transaction systems and replicated databases.

Related Work

A lot of work has been done in the area of distributed and replicated databases. Two-phase-commit-like protocols [12] are the main tool for providing a consistent view in a distributed database system over an unreliable network. In a typical protocol [14] of this kind, the transaction coordinator sends a request to prepare to commit to all the participant processors. Each processor answers by a "ready to commit" or an "abort" message. If any processor suggests to abort, all processors abort. The transaction coordinator collects all responses and informs the processors to either commit or abort. Between the two phases, each processor keeps the local database locked, waiting for the final word from the transaction coordinator. The main drawback of such a protocol is that when the transaction coordinator fails, all processors may be left blocked and can't resolve the last transaction. Moreover, when a partition occurs, the lack of dynamic replication forces many distributed transactions to abort.

There are many protocols that optimize specific cases [19]. For example, in a fully replicated database, achieving a quorum is enough to resolve the transaction [21]. Some solutions limit the transactional model to commutative transactions. Others give special weight to a specific processor or to a specific transaction [20]. Explicit use of timestamps enables others [4] to overcome the difficulty.

More advanced solutions define the quorum adaptively. When a partition occurs, if a majority of the previous quorum is connected, a new and smaller quorum is established and updates can be performed within this partition [10, 11]. The advantage of those methods is that in many cases, when faults occur and even if the network splits into two, the larger partition can perform updates. The drawback is that there can be situations where almost all of the processors are connected, but cannot perform updates, because of the potential existence of a previous surviving quorum among the processors that are currently down.

Communication mechanisms are central to any distributed transaction system. Today, distributed and replicated database systems are built using point-to-point communication mechanisms, such as TCP/IP, DECNET, ISO and SNA. These mechanisms provide reliable point-to-point message passing between live and connected processors. These mechanisms do not make efficient use of the hardware capabilities. In particular, the broadcast or multicast mechanisms are not used. A distributed or replicated database system is a classic candidate for using these mechanisms, since each update message is sent to multiple destinations.

In addition, a replicated database system requires a global order on the update transactions. When point-to-point communication mechanisms are used, the message ordering must be determined above the communication layer. Group communication services that utilize broadcast or multicast mechanisms can lead to simpler and more efficient solutions.

One of the leading systems in the area of group communication is the ISIS system [5]. The novelty of ISIS is in the formal and rigorous definition of the service interface. Moreover, ISIS utilizes algorithms that are formally proven to have certain needed properties [6] such as virtual synchrony.

The Trans and Totem protocols [3, 17] for ordered multicast utilize the broadcast capability of the network. These protocols spare the need of separate send operations and acknowledgment management for each of the destinations. Similar approaches can be found in the Psync protocol [18], in the Amoeba system [15], in Delta-4 [22], and in the Chang and Maxemchuk protocol [7].

The above systems and protocols are leading the way for better usage of the hardware capabilities, where certain data replication is needed in an environment which is built as a collection of several local area networks.

Membership maintenance is necessary for developing a broadcast or a multicast protocol for group ordered communication. When a point-to-point communication mechanism is used between two parties, the management of ordering messages and acknowledgments is easy. Each processor knows which messages it received, acknowledged, or sent, and for which messages it received an acknowledgment from the other party. When a broadcast communication mechanism is used, a second level of knowledge is needed to determine which messages were received by which processors, which processors are currently connected, and what is the global message order. A good membership algorithm provides this knowledge, even in the presence of processor crashes, processors recovery, network partitions and merges, and booting of new processors.

It is a known fact [13, 8] that the membership problem in a dynamic asynchronous environment, when faults may exist, is unsolvable. The problem is the inability to distinguish between a slow machine and one that has crashed. A number of approaches to bypassing this obstruction exist [1]. In practical asynchronous systems it is often preferable to give up on a slow machine, rather than get stuck in waiting.

The rest of the paper is organized as follows: The next section describes the Transis environment. The third section describes the Replicated Mail Service application. This section specifies the requirements from the replication protocol and the interface presented to the client. It also describes the conceptual solution. The detailed algorithm of the Replicated Mail Server can be found in Section four. Section five compares the designed solution with other strategies, with and without Transis. Section six concludes the paper.

2 The Transis Environment

The Transis System [1, 2] is currently under development at the Hebrew University of Jerusalem. The Transis domain consists of a set of processors that can communicate via multicast messages.

Processes in the Transis environment use group communication mechanisms in order to send and receive ordered messages. The three service levels for group communication in Transis are:

The *Causal* multicast service disseminates messages among the process group such that *causal order* of delivery is preserved. Motivated by Lamport's definition of order of events in a distributed system ([16]), the causal order of message delivery is defined as the transitive closure of:
1. $m \stackrel{cause}{\longrightarrow} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$
2. $m \stackrel{cause}{\longrightarrow} m'$ if $\text{send}_q(m) \rightarrow \text{send}_q(m')$

The *Agreed* multicast service disseminates messages among the process group such that *total order* of delivery is achieved. This order is consistent with the causal order. Moreover, it is consistent over overlapping groups. Transis utilizes two different algorithms to implement this service efficiently [3, 9].

The *Safe* multicast service delivers the messages in the same order as the Agreed multicast service. A message is delivered only after it was received by all the processors in the currently installed membership.

Transis actually guarantees a stronger property for Safe multicast: a Safe message sent by processor p will be delivered at q that doesn't fail, if at the time of delivery of the message at p, q is in p's membership.

The environment is dynamic, processors can crash and restart, and the network can partition and reconnect. The membership protocol of Transis automatically maintains the membership of connected processors, which is necessary for constructing fault-tolerant distributed applications.

The membership protocol guarantees virtual synchrony. It is symmetric and does not disturb the regular flow of messages. The protocol does not allow indefinite blocking of processors, but may rarely remove from the membership live (but inactive) processors unjustly. This is the price that must be paid for maintaining the membership in consensus among all the active processors in an asynchronous environment without blocking. In addition, if a processor is inadvertently taken out of the membership, it can rejoin immediately.

The greatest challenge Transis handles is partitioning and merging. They can be handled because of the symmetric design. All previous membership algorithms deal with the joining of single processors only. Moreover, most systems do not continue execution during network partitioning. However, partitions do occur, especially when the network includes bridging elements. The communication layer must continue to provide service to the application, as well as accurate information about the status of the system, even when the network configuration dynamically changes. The complete merging of partitioned histories is application dependent and therefore is not handled by Transis.

3 A Replicated Mail Service

In this Replicated Mail Service application, we make use of the information supplied by the safe messages and the configuration change messages provided by Transis, in order to complete the merging of partitioned histories for our specific needs.

We rely on Transis to efficiently disseminate messages among the currently connected processors and to manage the group membership. Therefore, we do not need to explicitly handle the sending of acknowledgments, message omissions, and processor faults.

3.1 The Requirement from the Replication Protocol

An **Eventual Path** exists between servers S_0 and S_n within the time frame $[t..t']$ if there is a sequence of servers S_0, S_1, ..., S_n and a sequence of time indicators t_0, t_1, ..., t_n such that

- $t \leq t_0$ and $t_n \leq t'$.
- for each $0 \leq i < n$, $t_i < t_{i+1}$.
- for each time t_i, a possible message sent by S_i to S_{i+1} at time t_i can be delivered at S_{i+1} by time t_{i+1}.

The requirement for the replication protocol is the following: A high priority message sent by a server S_0 at time t will reach the server S_n by time t' if there exists an Eventual Path between S_0 and S_n within the time frame $[t..t']$. Note that the definition is carefully stated for a specific message, whereas if the connection duration allows, all pending messages will be delivered.

3.2 An Interface Specification of the Client

The specifications below define the interface between a mail client and a mail server.

Connect-to-server request must be invoked by the client before it can perform any other request. The client must specify one of the working servers in its partition to which it wants to be connected. The client can also invoke this request when it wants to change to another server, if the previous one did not respond.

Send-mail request is invoked when the client wants to send a message.

Query-mail request is invoked when the client wants to obtain its list of messages. The list also indicates whether each message was already read by that client.

Read-mail request is invoked when the client wants to read one of its messages.

Delete-mail request is invoked when the client wants to delete one of its messages.

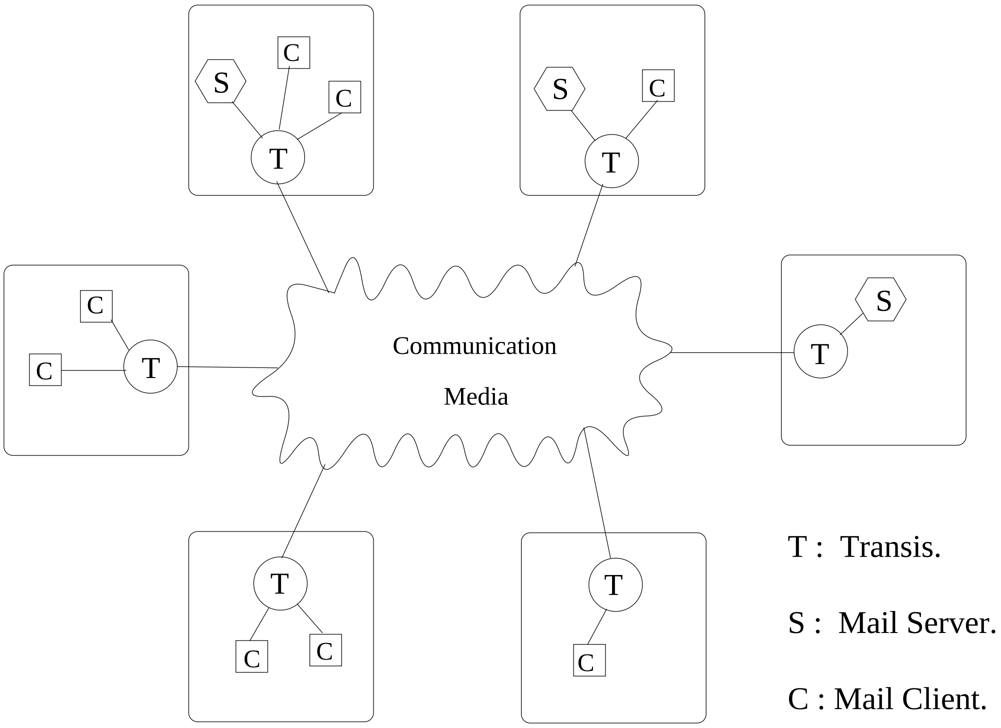

Fig. 1. *The Replicated Mail Service architecture*

3.3 A Conceptual Solution

The architecture of the Replicated Mail Service is presented in Figure 1. Each processor runs the Transis demon as a process. In addition, each processor that is part of the Replicated Mail Service runs the mail server as a process. All the client processes (i.e mail users) and all the server processes are connected to the local Transis demon. Each client process can logically connect to one of the mail servers currently running in its partition. The basic concept behind the solution is to have a process group containing all the mail servers.

Each request from a client process to one of the mail servers that changes the message database creates an update message (i.e a transaction). This update message is then multicast via Transis to all the currently connected servers. Since Transis provides a reliable communication service in the sense that omission faults are recovered, there is no need to manage acknowledgments inside the currently connected group of servers. There is a need, however, to transmit those update messages within a configuration after partitioned or failed processors that do not have those messages join the configuration.

It is important to emphasis that after reaching a second level of knowledge about the state of the message database among the servers in the current configuration, knowledge from servers which are not part of the current configuration can be introduced only after a merge occurs.

4 The Algorithm

The basic idea is to order update messages (i.e transactions) according to a Lamport timestamp [16] which is stamped by the transaction initiator (i.e one of the Replicated Mail Servers) at the time of creation. When two transactions are stamped with the same Lamport timestamp, they are ordered according to the initiator's server id. The protocol completes this order to a global total order that is consistent among the servers. This order preserves causality.

Each server tags all the received transactions. Each transaction can be marked with one of three colors:

- *white:* A white transaction is stable and ordered. A transaction T is *stable* at server p if p knows that all of its targets received it and put it on disk. A transaction T is *ordered* at p if p knows that no transaction that will be received by p in the future will be ordered prior to T. The above order is consistent among all servers. Note that, since T is stable at p, p does not need to explicitly keep T because no other server will ask for it. If T is already invoked at p, p can discard it.
- *green:* A green transaction is ordered but is not locally stable.
- *red:* A red transaction is neither ordered nor stable (yet).

All the white transactions precede the green transactions and the red transactions in the order and define the white zone. All the green transactions precede the red transactions in the order and define the green zone. Similarly, the red transactions define the red zone. The protocol places the separating lines among the transactions according to their colors consistently. A transaction can be tagged by different colors at different servers.

The stable database contains all the transactions in the white zone and the green zone. The red zone is the zone of uncertainty. All the transactions in the red zone are relatively ordered among themselves. Moreover, there are no causal holes in the red zone. However, there is a possibility that a new transaction, invoked at some other processors, will have to be placed anywhere in this zone. All the transactions in the green zone and in the red zone have to be retained in case of a need for retransmission.

Each transaction contains piggybacked information that helps determine the order and the stability of previous transactions, to the best of the transaction initiator's knowledge. Based on this knowledge, transactions can change colors (i.e from red to green and from green to white). The structure of the three transaction zones is presented in Figure 2.

New transactions are disseminated throughout the currently connected servers (i.e. the membership). Transis assures that all the connected servers will receive those transactions. In case of a membership change in the configuration of connected servers, Transis assures that the Virtual Synchrony property ([6, 1]) will hold. Using this property, the servers of the new membership exchange messages

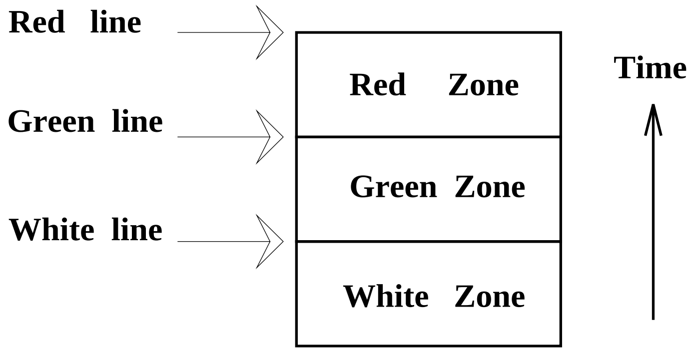

Fig. 2. *The Three Transactions Zones*

containing information about transactions in their local zones. If they remain connected, each processor knows exactly which of the processors have received which transactions. They can now deterministically determine which processor will retransmit which transaction (in case retransmissions are needed).

4.1 The Data Structure

Each Replicated Mail Server maintains the following data structures:

Server_id is the unique id of this instance of the Replicated Mail Server.

Servers is the data structure which contains the current membership of connected servers in the partition of this instance of the Replicated Mail Server.

List is the data structure in which all the mail messages and other update messages are stored. The messages are stored according to the structure of the three transaction zones. The *List* has an up-to-date backup on disk.

Last_LTS is a vector which contains, for each server, the highest Lamport timestamp of all received messages initiated by that server. The *Last_LTS* vector has an up-to-date backup on disk.

Last_ARU is a vector which contains for each of the servers, the highest Lamport timestamp of a message in that server's green zone (i.e the green line in that server). The *Last_ARU* vector has an up-to-date backup on disk. ARU is an acronym meaning "All Received Upto", i.e. this instance knows that that server has, at least, all the messages that are stamped with a Lamport timestamp no greater than the corresponding Last_ARU entry.
Note that the particular entry *Last_ARU*[*Server_id*] corresponds to the green line in this instance of the Replicated Mail Server.

LTS is the highest Lamport timestamp received or initiated by this instance of the Replicated Mail Server. It is also the red line of this instance.

ARU is the Lowest entry of *Last_ARU*. This instance of the Replicated Mail Server knows that all the servers have all the messages that are stamped with a Lamport time stamp no greater than *ARU*. This is also the white line of this instance.

Min_LTS, Max_LTS are vectors that are used after merging a partition to determine which messages require retransmission.
All the messages in the semiopen range $(Min\_LTS..Max\_LTS]$[1] for each server should be retransmitted.

4.2 The Message Formats

The main message formats are presented in Figure 3.

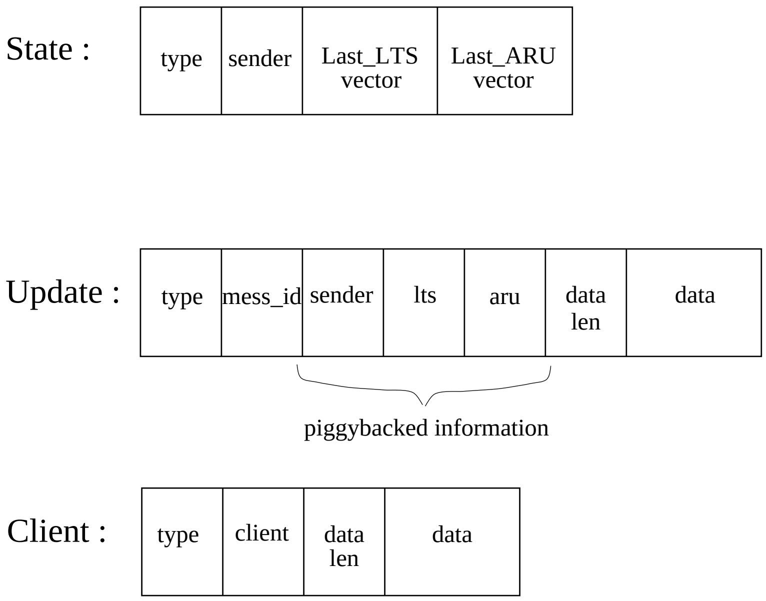

Fig. 3. *The Message Formats*

4.3 Event Handling

There are seven events that the mail server accepts. The events correspond to the reception of messages of the following seven types : Membership change message from Transis, State message from another server, Update message from another server, Send-mail, Read-mail, Delete-mail and Query-mail messages from a client. The mail server is an event-driven program that invokes an event handling routine.

[1] Max_LTS is included but Min_LTS is not

Membership change message from Transis

The mail server identifies the other mail servers in its current configuration. If new members joined the servers' group membership, a recovery process is required. If so, the server transmits a State message containing its database state, and sets the recovery vectors as follows:

$$\forall i \in Servers, \quad Min\_LTS[i] = Last\_LTS[i]$$

$$\forall i \in Servers, \quad Max\_LTS[i] = Last\_LTS[i]$$

State message from another server

The mail server updates its $Last\_ARU$ status vector and its recovery vectors according to the corresponding vectors in the received State message:

$$\forall i \in Servers, \quad Last\_ARU[i] = \max(Last\_ARU[i], message.Last\_ARU[i])$$

$$\forall i \in Servers, \quad Min\_LTS[i] = \min(Min\_LTS[i], message.Last\_LTS[i])$$

$$\forall i \in Servers, \quad Max\_LTS[i] = \max(Max\_LTS[i], message.Last\_LTS[i])$$

When updating the $Last\_ARU$ vector, the mail server updates its knowledge about the last message each server has received from every other server. When updating the Min_LTS recovery vector, the mail server updates its knowledge about the maximal message from each server, all the servers in the **current** configuration have reported. When updating the Max_LTS recovery vector, the mail server updates its knowledge about the message with the highest ordinal from each server that at least one server in the **current** configuration holds.

After receiving a State message from each of the mail servers in the current configuration, the server computes which messages need to be retransmitted. Clearly, those messages are the messages in the range $(Min\_LTS..Max\_LTS]$ For each server.

Since all the servers receive exactly the same messages if they remain connected, they will reach the same conclusion. Based on the fact that they all know the state of each of them prior to the recovery, they can determine, without additional messages, which server will retransmit which messages, and when this retransmission will take place.

The algorithm guarantees that when mail server p receives a message which was initiated by mail server q, then p already has all q's previous messages.

Update message from a server

An update message is triggered by Send-mail, Read-mail and Delete-mail requests, invoked by clients. Upon receiving an Update message from a mail server, the server updates the sender server's entry in the $Last\_LTS$ and $Last\_ARU$ vectors as follows:

$$\begin{array}{ll}\textbf{if} & Last\_LTS[message.sender] < message.lts \\ \textbf{then} & Last\_LTS[message.sender] = message.lts;\end{array}$$

$$\begin{array}{ll}\textbf{if} & Last\_ARU[message.sender] < message.aru \\ \textbf{then} & Last\_ARU[message.sender] = message.aru;\end{array}$$

The server also tries to advance the red, green and white lines.

- The red line is advanced if the *lts* field on this message is higher than the current LTS variable. In that case

$$LTS = message.lts.$$

- The green line is advanced if the sending server's entry in $Last\_LTS$ was lower than any other entry in $Last\_LTS$, and was increased. In that case

$$Last\_ARU[Server\_ID] = \min_{i \in Servers} (Last\_LTS[i]).$$

- The white line is advanced if the sending server's entry in $Last\_ARU$ was lower than any other entry in $Last\_ARU$, and was increased. In that case

$$ARU = \min_{i \in Servers} (Last\_ARU[i]).$$

  After the white line is advanced, all the update messages in the *List* which are below the white line can be discarded. This can be done because the server knows that all other servers have those update messages and they will never be requested.

The update message is then inserted into the *List* in memory and on disk according to its message id. If it is an update about read-mail or delete-mail requests, the target message's state is also updated in memory and on disk. For example, if an update message is received for a delete-mail request, then that update message is inserted to the *List*. In addition, the mail that was deleted is marked as deleted in the *List*.

Send-mail, Read-mail and Delete-mail messages from a client

The server checks the validity and permissions of the request. Only non-deleted messages can be read. If it is the first time a message is read or the first time it is deleted, the server creates an update message which corresponds to the request and transmits it to the servers via Transis. If the request was a valid Read-mail request, it also retrieves the mail message and sends it to the client.

It is important to emphasize that those events do not change the state of the database. Only the corresponding update message will cause a change when invoked.

Query-mail message from a client
The server retrieves the identifiers of messages that were sent to the client and reside in the database of this server. The server then sends an answer message containing these identifiers and a field for each message to indicate whether this message is a new message or whether this message was already read by this client.

5 A Comparison with Other Methods

We now present a comparison of our solution with other possible strategies. First, we discuss strategies that do not use Transis, and instead use a standard transport layer such as TCP/IP. Next, we discuss several solutions that utilize Transis as a transport layer, with different levels of knowledge about the membership.

Centralized Server
This is the usual solution in which one server manages one copy of the mail database. The advantage is that there is no overhead to maintain consistency, the solution is simple and straight forward.

The solution has several drawbacks. If the mail server is not working, or if the computer which hosts the mail server does not function properly or is not connected to the client's computer, then the client can neither send new mail nor read its mail. In some systems this may lead to storing the message in a secondary mail system, where the client is not aware of it.

One Server With Several NFS Copies
A way to overcome some of the problems in the centralized solution, is to have a centralized server which manages several copies of the mail database in different computers over the network, using NFS. Here also there is no communication overhead to maintain consistency. The problem is that the server needs to maintain tedious bookkeeping of separate copies, when network partitions may occur. Furthermore, two inconsistent copies can be found in one partition when the server is in another partition or is not working.

In this solution, the client can send mail only if it is connected to the server. The client can read mail if it has a connection to one of the NFS replica.

Replicated Servers
A natural way to replicate the service is to use two phase commit as in distributed databases. This is completely unacceptable in a mail application due to the blocking characteristic of two phase commit. Therefore, a nonblocking strategy, similar to the solution presented in this paper may be preferable. The problem is, that without a Transis-like transport layer, upper level protocols need to support three major properties: multicasting, message ordering and membership management. When utilizing a point-to-point transport layer such as TCP/IP, it is difficult to synchronize the membership changes with ordered message delivery

and to ensure that the configuration view is consistent among all the members of the configuration. It is even more resource demanding when done by upper-level protocols.

A way to solve the problem when using point-to-point transport protocols is to implement stable queues between pairs of servers. This scheme is efficient if the responsibility of each server is limited to disseminating its own messages. Thus, a message initiated at server p will reach server q, only if server p was connected to server q. When the requirements are enhanced (as in this paper), managing stable queues efficiently is even more complicated.

Using Transis Transis provides an efficient and reliable multicast service for ordered messages, combined and synchronized with membership services, in an asynchronous environment. By using Transis, we can base our solution on the agreement reached among the processors about the ordering of messages and membership events. The application layer (the mail server in this case) obtains the needed information about membership and message events without going down to the data-link or network levels. The server layer has only to maintain state consistency and to recover histories.

We have investigated several methods of maintaining state consistency and recovering history. The methods differ in the level of knowledge about the membership events. The first method assumes no knowledge about membership events, the second assumes obtaining the membership events without history tracking of membership configurations, and the third method presented in this paper assumes full knowledge about membership events in the system (i.e. each server knows which servers participated in its previous membership and which participate in its current membership).

When there is no information about membership changes, missing messages may be tracked as follows. Messages include piggybacked information about preceding messages. Upon receiving messages that follow missing messages, the server discovers that certain messages are missing, and requests for retransmission. This is similar to other lower-level algorithms such as in the Trans algorithm [17], the Psync algorithm [18],,, and in Transis [2]. Recovery of messages, after partitions are remerged, occurs only if new messages are generated. This is a major drawback because it creates the necessity to initiate new messages from time to time. Long timeouts lead to inconsistent views among servers in the same partition. Short timeouts lead to network flooding.

When there is no history tracking of membership configurations, the algorithm presented in this paper will work with the exception that the server must initiate the recovery process each time the membership is changed. The method fully exploits the available membership information received from Transis. Thus, the recovery process takes place only when new members join the membership.

6 Conclusions

The paper presents the development of a highly available and reliable distributed application in the Transis environment. There is an ongoing debate about the usefulness of a reliable membership and message ordering service. The application we chose is a typical application in a distributed system although many other applications exist. The experience we have gained is that having a reliable transport layer drastically simplifies the development process of a replicated application.

The application presented above was assigned as the final project in a Distributed Algorithms course, taught in Fall 1992, at the Hebrew University of Jerusalem. The various solutions presented above cover some of the projects developed during the course. The solution described in the paper was based on a more general replicated transaction tool developed by the authors.

In this solution, Transis takes care of the transport and lower layers responsibilities, such as message dissemination, sending acknowledgments, flow control and membership changes detection. When using Transis, rather than point-to-point mechanisms, it is easy to synchronize the membership changes with ordered message delivery. The only part left to the upper layer protocol is the recovery of the history of events after a join or a processor recovery, and the global ordering of those events. This part is not an integral part of the transport layer and may vary for different applications.

Studying the various alternative solutions to the problem we realized that different teams chose to make different use of the services offered by the lower layer. Some have, unintentionaly, duplicated part of the service offered by the lower layer. We discovered that projects that make better use of the transport layer services tend to be simpler and smaller.

Acknowledgments

We acknowledge the contribution of Ahmad Khalaila, Rimon Orni and others from the High Availability Lab at the Hebrew University. We thank them and other students for many hours of fruitful discussions, and for developing alternate solutions to the problem.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, number 6th, pages 292–312, November 1992.
2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Annual International Symposium on Fault-Tolerant Computing*, number 22, pages 76–84, July 1992.
3. Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *International Conference on Distributed Computing Systems*, number 13th, pages 551–560, May 1993.

4. P. Bernstein, D. Shipman, and J. Rothnie, J.B. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. on Database Systems*, 5(1):18–51, March 1980.
5. K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual.* Dept of Computer Science, Cornell University, Sep 90.
6. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Ann. Symp. Operating Systems Principles*, number 11, pages 123–138. ACM, Nov 87.
7. J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer systems*, 2(3):251–273, August 1984.
8. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.
9. D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered broadcast in asynchronous environments. In *23rd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, June 1993.
10. A. El Abbadi and N. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, (6):319–332, 1991.
11. A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *ACM SIGACT-SIGMOD Symp. on Principles of Database systems*, number 5, pages 240–251, Cambridge, MA, March 1986.
12. K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
13. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.
14. J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course,Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.
15. M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 78.
17. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.
18. L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.
19. C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *ACM SIGMOD Symp. on Management of Data*, May 1991.
20. M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Trans. on Software Engineering*, 3(3):188–194, May 1979.
21. R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.
22. P. Verissimo, L. Rodrigues, and J. Rufino. The Atomic Multicast Protocol (AMp). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, pages 267–294. Springer-Verlag, 1991.