A Fault-tolerant Distributed Mail Service

{David Cheng (dscheng), David Dickinson (davidrd), Subhasish Bhattacharya (sbhatt)} @mit.edu

6.824: Distributed Computer Systems
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Electronic mail has become an important mode of communication for applications ranging from simple conversation to delivery of important data. Most of the current mail systems do not provide guarantees on the availability of email to the client. This paper describes the implementation of a Fault-Tolerant Distributed Mail Service, or FDMS. It provides highly available and reliable access to mail for multiple users on multiple servers, and maintains consistent state replicas across the mail servers. The system will guarantee access to email in the event of failures in up to half of the servers.

1. Introduction

People use electronic mail in a variety of applications. Often, very important information is delivered through email. A system that allows the client to access their mail from multiple servers and provides tolerance against failure of multiple servers would be an important service for users.

1.1 System Requirements

- **Fault-tolerance**: In any large system, node failure is unavoidable. Our system ideally would provide a consistent and available view of the mail state to all clients at all times. Certain failure modes are inevitable, but as our server recovery mechanism will be built into the system design for normal operation, the overhead will not be substantial.
- **Scalability**: Addition and removal of new servers to FDMS should not disrupt the operation of the other servers already operating in the system.
- **Performance**: The performance of FDMS should be comparable to, or not significantly worse than, a nonreplicated system. System performance should scale linearly with the number of servers.
- **Correctness**: All actions are performed by all servers in exactly the same order.

The goal of this project is to implement a highly available fault-tolerant distributed mail service. The system is made highly available by introducing redundancy in the form of replicas of the mail service. A primary server is elected to maintain the consistency of the replicas.

We expect to analyze the benefits and problems with using state-replication in providing a fault-tolerant data service. Each server will contain a data file and a log file. Read and append operations to the data file will be used to simulate email operations to read, write, and delete messages for multiple users. The log file will be used to recover requests from the clients that were not yet processed by the primary in the event of failure. A variant on the Paxos consensus algorithm is used in conjunction with the primary copy method to attain agreement across all replicas on the ordering of mail operations, and continued functionality in the face of failures of up to half of the replicas.

Section 2 provides an overview of the system design. Section 3 provides details on our implementation of the system. Section 4 describes the analyses done on the system. Section 5 discusses related works that were used in developing this system. Section 6 summarizes the results and provides conclusions on the system.

2. System Design

2.1 Overview

The fault-tolerant distributed mail service (FDMS) provides high availability and fault-tolerant storage of mail messages. Clients may interact with the mail system from any server in the network, and access their mail in a durable and consistent state. Fault tolerance is provided via strict data consistency among servers. Servers have identical copies of data, and servers in the system are not allowed to have divergent data. This incurs a penalty in that each client operation must be disseminated in the system. However, strict consistency was chosen over eventual consistency because it provides stronger guarantees on the availability and durability of user data.

Consistency of actions is managed through the primary server. The primary is the central node for assignment of action ids to new actions in the system. In order to manage the possibility of primary failure, a new primary that is elected in the system must be able to determine the correct action id to use when assigning action ids. This new primary uses the Paxos consensus algorithm [1] to determine the highest action id that other servers have seen. The Paxos consensus algorithm has been chosen since it is the most fault-tolerant way to get consensus without real-time guarantees [2].

Figure 1 illustrates the components of FDMS. A client uses the interface described in Section 2.2 in order to manage current email or deliver new email. All servers in the system are equivalent in design, which is described in Section 2.3. Each server has a unique id, which is the concatenation of its IP address and port number. During operation, there will be one server elected as the leader, the primary server, to ensure strict ordering of operations on the mail file. All servers other than the primary server are slave servers. The primary server also acts as a slave server so a client can directly send it a request.

A server is responsible for servicing client transactions and propagating operations that modify the mail file to the other servers in the system. Propagation of operations is handled among servers through a separate inter-server interface, described in Section 2.4.

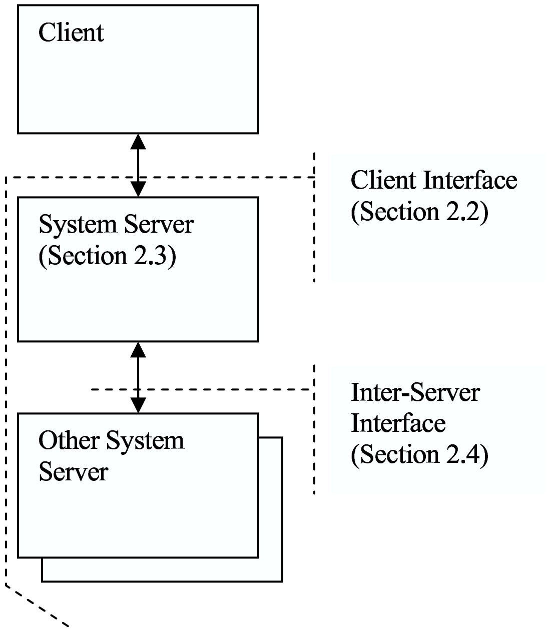
Client
Client Interface (Section 2.2)
System Server (Section 2.3)
Inter-Server Interface (Section 2.4)
Other System Server

Figure 1: System Component Diagram

To differentiate between requests received by a client and those within the system, we use two different terms. Requests from the client are referred to as operations and are assigned local operation ids. Requests which have been processed by the primary are referred to as actions, and are assigned action ids.

2.2 Client interface

FDMS provides a simple interface to clients, allowing a client id to read and delete messages from their own mailbox, as well as to write messages to any mailbox in the system.

Read Request `{userId messageId}`
Delete Request `{userId messageId}`
Write Request `{destinationId message}`

Figure 2: Client requests to server

The client interface is serviced by RPC. Requests for READ, DELETE, and WRITE operations are supported. The results of a DELETE or WRITE operation are not observable to the user until FDMS commits the operation in the system.

Read response `{message status}`
Delete response `Status`
Write response `{messageId status}`

Figure 3: Server responses to client

2.3 Server Design

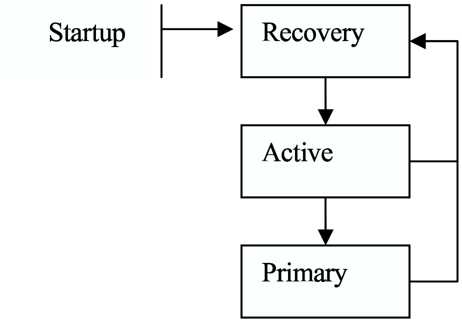

Figure 4: Server states

Server:
ID: unique, (IP, Port) pair
Primary Election Time: start time of an election
Primary: current primary's ID
Peers: list of all peer IDs
Max Action Id: id of latest action committed to this server
Mail File: Disk file containing committed mail
Delete Record File: Disk File containing deleted mail information. Garbage collected and reflected in the mail file during startup
Operation Log: Non-volatile Log of committed and uncommitted operations received by this server
Action Buffer: Volatile storage used to buffer actions, which have been assigned an action id by the primary

Figure 5: Server information

2.3.1 Active

In the Active state, the server processes user requests. Each server uses the Mail File to service client reads and the Operation Log to manage operations while they are uncommitted in FDMS. Each server has an operation log, which contains uncommitted WRITE and DELETE operations for use during recovery of primary or slave server in the event of failure.

When a server receives a WRITE or DELETE operation it assigns an operation Id to the request before forwarding the request to the primary. The operation id is a local index, which is used along with the owner id by the server receiving the request to determine which request can be committed in the operation log once the corresponding action has been performed. The READ operations can be processed immediately and therefore do not need to be stored in the operation log.

An action buffer is present to buffer actions, which have been assigned an action id by the primary until it can be committed at the conclusion of the two-phase commit and until the action with the action id following the current maximum action id has been received.

Each client WRITE operation is handled internally by the receiving server as an append operation to the Mail File while a client DELETE operation is handled as an append operation to the Delete record file. By performing DELETE operations using an append operation, the overhead of removing the record from the mail file is avoided. During the update and read operations the delete record file is checked before the mail file. During startup (section 2.3.3) garbage collection is performed to commit the changes in the deleted record file to the mail file.

Before a commit to the Mail File or Delete Record file is committed, the server must request an action id for the operation from the primary by queuing the requests into the Operation Log as uncommitted operations.

Each slave server sequentially handles uncommitted operations in the queue by sending *primary request* messages to the primary. If there is currently no primary due to FDMS being in leader election mode, the originating server arranges to resent the request after a suitable timeout.

An active slave server, which receives a *Primary Index* message, will reply with the primary id. This ensures that servers who start-up in a system can correctly identify the current primary.

2.3.2 Primary

In order to assure progress in the PAXOS algorithm, a unique primary must be chosen among all servers. For simplicity, the primary in FDMS is the server with the largest ID. We define the ID A_{ID} larger than the ID B_{ID} iff $A_{ID}(Port) > B_{ID}(Port)$ or $A_{ID}(Port) = B_{ID}(Port)$ AND $A_{ID}(IP) > B_{ID}(IP)$.

The first action the primary performs when it becomes the primary is to get a consensus on the highest action id not yet assigned to any of the server's operations. This occurs when the FDMS goes into leader election mode. The system determines the highest action id using the Lamport's Paxos consensus algorithm. As long as a majority of the servers are alive and there is a single server coordinating the consensus, Paxos is guaranteed to succeed. When the primary has a consensus on the highest action id it can begin processing requests after updating its mail file state to be consistent with that of the determined action id. It requests the contents of any actions it does not have locally by sending *update request* messages. In the Primary state, the server continues to process user requests as in the Active state, but also processes Primary Requests from other servers.

2.3.3 Startup

During startup, a server is initialized with the ID of all other servers in the system. Each server assumes all other servers are alive and chooses the leader as described in Section 2.3.3. A server begins in the recovery state to ensure that operations performed by clients are committed to the system. This will be used to sync up the system during the startup.

The system startup is triggered by a Find Primary request from any client to a server. On receiving this message the server triggers a leader election.

2.3.4 Recovery

During recovery, the server first determines the current primary by sending out primary index operations to its peer servers until it receives a response from them including the primary information.

Next the recovering server synchronizes with the primary, retrieving the primary's current maximum action index with a primary action index operation. If its own maximum action id is lower than the primary's, the server will request a copy of each action it is missing. If such an action contains its own ID as the owner, the server may mark the corresponding operation as committed in its Operation Log. After correcting its Mail File, the server queues uncommitted operations to be sent to the primary and then transitions to the Active state.

2.4 Server operation

This section describes the sequence of operations of the different phases of FDMS.

Section 2.4.1 describes the slave server request. This phase occurs after a WRITE or DELETE request is received from the client and the operation has been added to the operation log and assigned an operationId by the slave server.

The primary server response is described in section 2.4.2. This phase is triggered by a request from the slave server.

The primary server election is described next in section 2.4.3. This phase occurs during startup, when a primary fails, when a primary discovers that it is in a minority partition and when the network connection between the slave server and the primary server is disrupted.

Finally, section 2.4.4 describes the server synchronization, which occurs during recovery and after primary election.

2.4.1 Slave Server Request

1. Receive request from client
2. If READ request process immediately from mail file
   Else if WRITE or DELETE add to operation log and assign the operation an operation id
3. Request the primary to assign an action id to the request
4. On receiving a response from the primary add the action to the action buffer
5. On receiving a commit from the primary for an action mark the action as committed in the action buffer.
6. Process requests from the action buffer is possible
7. Un-commit processed actions from the operation log.

Figure 6: Server actions – non-failure case

Primary request

```
{message    userId    operationId}
{messageId  actionType    ownerId}
```

Figure 7: Primary Server request

2.4.2 Primary Server Response

The response from the primary server is implemented using a two-phase commit protocol.

Primary Response

1. Receive request from server
2. Assign an action id to request and broadcast to all slave servers
3. Add operation to action buffer
4. On successfully broadcasting to a majority of servers in step 2 broadcast a commit for the action to all
5. Commit operation in the action buffer and proceed as in the server case.

Figure 8: Primary actions – non-failure case

In response to a primary request from a slave server the primary assigns the request the highest action Id not yet assigned. This action Id will be the highest actionId it knows about incremented by one. The primary then broadcasts a primary response message to all servers.

Primary response

```
{message      userId     messageId}
{operationId actionType actionId
 ownerId}
```

Primary Commit

```
{actionId}
```

Figure 9: Primary Server response

Primary Commit

If the primary response RPC succeeds to a majority of the servers broadcasted to the primary broadcasts a primary commit message to all servers. However, if the request broadcast fails for a majority of the servers the primary triggers a leader election.

This ensures that if the network partition of the primary becomes a minority the primary will perform no updates.

2.4.3 Primary Server Election

The server initiating the Primary Election clears its primary ID field and sends an agent alive message to all other servers to begin the Primary Election phase. This produces a cascade of agent alive messages, one message from every live server to every other server. After a suitable delay to allow the messages to settle, each server sets their primary ID according to the algorithm described above (Section 2.3.2).

There is guaranteed to be a single primary at a time since any server requires a majority to be elected primary. If at anytime the network partition in which the primary exists looses its majority a new leader election round is triggered will be triggered as a part of the two phase commit, which will be guaranteed not to succeed until the partition regains its majority.

Once a primary is elected the primary sends agent status messages to gain consensus over the highest action index in the mail file. When the consensus concludes the primary informs the slave servers the decision reached by broadcasting max action id messages. This phase is followed by the server synchronization phase described in the next section.

Agent Alive `{Id}`
Agent Status `{ID   isPrimary  currentActionID}`
Max Action Id `{Action Id}`

Figure 10: Leader Election

2.4.4 Server Synchronization

A server may request retransmission of the contents of an operation by sending an update request to the primary server.

Update request `actionId`

Update response
`{ownerId    message    userId}`
`{actionType messageId  operationId`
`actionId}`

Update actionId request `{Void}`
Update actionId response `{actionId}`
Primary Index `{void}`
Find Primary `{void}`

Figure 11: Server synchronization requests

It sends a message with the desired action id to the primary server. The receiving primary server replies with an update response.

A server may request the current action id from the primary. The primary will reply with a message of type update actionId response.

2.5 Failure modes

2.5.1 Primary Server Failure

If a server does not get a response from the primary within a predetermined timeout after sending a Primary Request, it may assume that the primary is dead. In this instance, the server begins a Primary Election round.

2.5.2 Slave Server failure

A slave server can fail, i.e. have a stopping failure and then become alive, at any time. Once a slave becomes alive it will proceed through the standard restart procedure to sync with FDMS.

3. Implementation Details

3.1 Code Base

The implementation of FDMS that we developed needs to handle a large amount of asynchronous messages. In order to simplify the task of handling these messages, we decided to use the asynchronous event-handling library libasync to manage the specifics of asynchronous events and messages.

Libasync also provided us with the RPC mechanism to managing delivery of messages.

3.2 Network Assumptions

Our implementation also makes a number of assumptions about network behavior. They are as follows:

1. Reflexivity of network connections. If server A can directly message server B, then B can directly message A.
2. Transitivity of network connections. If server A can directly message server B, and B can directly message C, then A can message C
3. If another server does not respond to an RPC within the expected maximum network roundtrip time, it will never respond to that RPC call.

Assumptions 1 and 2 ensure that any network partition is fully connected so that we don't need to concern ourselves with explicit relaying of messages through intermediate servers.

Assumption 3 ensures that each round of primary election and action indexing is kept isolated. No responses from a previous round will contaminate a future round or initiate another round.

3.3 Code modules

The implementation of the servers was managed through a single C++ class. Within the class, different modules were implemented to manage the client-server requests, the primary server election, and the server-server synchronization of operations on the mail file.

4. Analysis

We experimented on the time required to carry out requests within the system. In each case, we attempted to measure times from a system of only one server, then of systems with multiple servers. This allowed us to get a sense of how response time reacts to system complexity.

4.1 Client request

In the case of a client request, the server servicing the request does not complete any server communication or synchronization before returning a response to the client. Therefore, in all cases, a client request is a measure of the RPC time plus the code path that services the request.

Startup	3.1 – 3.2 ms
Read	2.4 – 3.0 ms

Table 1: Time for client to receive response

4.2 Action synchronization

In order to measure the time required to synchronize actions, we timestamp the operation log entries. When the action is committed, we measure the time difference. This has been done for the write operation as well as the delete operation

	Write	Delete
1 Server	4.0 – 5.0 ms	1.3 – 2.5 ms
2 Servers	5.0 – 7.0 ms	2.8 – 3.7 ms
3 Servers	11 – 14 ms	4.1 – 4.5 ms
4 Servers	11 – 15 ms	4.2 – 4.8 ms
5 Servers	11 – 15 ms	4.2 – 4.7 ms

Table 2: Request to Server

We noted a reduction of approximately one to two ms when requests are sent to the primary. This corresponds to the reduction in the number of RPCs by one.

The system response time levels out after complexity grows to three servers. While we did not extend this to large numbers of servers, the response suggests that the system can scale to the number of servers sufficient to manage mail traffic.

5. Related Work

Grapevine [3] was the prototypical distributed mail service, with similar aims of high reliability. Where Grapevine had the more modest goal that the failure of a single Grapevine server would not render any service unavailable to any client, we have designed FDMS explicitly to handle up to n/2 failures. However, Grapevine focused upon many issues that we have less unaddressed, such as name space management and load balancing. While Grapevine provided a replicated user database, it did not replicate mail messages.

A more recent system, Porcupine [4], had similar goals to ours in the provision of high-availability replicated mail service, and updates could also be initiated at any replica. However, porcupine took a much different approach to the ordering of mail operations, choosing to use loosely synchronized clocks [5]. Porcupine adds redundancy by maintaining affinity between each user and a few preferred servers, a "Fragment list".

Unlike Porcupine FDMS is not dependent on the clocks of the replica servers being synchronized. While Porcupine depends on at least one replica server being alive for each fragment, FDMS just requires a majority of the replica servers to be alive. Each server maintains the current state of the mail server. While this adds an overhead to the processing of write and delete operations since requests have to go through the primary server, the response times for a read operation in FDMS is reduced. Read operations can be serviced directly by the replica receiving the request instead of having to access the mail from a server in the fragment list.

Also, Porcupine requires the soft state for the fragment states to be reconstructed during recovery and coordinated by a server. This recovery process is more complex than FDMS where the mail servers just need to obtain the actions missing due to the failure from the primary. Finally scaling the Porcupine system adds overhead in determining which fragment list new servers should be added for load balancing.

The algorithm used by FDMS to obtain consensus has an overall structure very similar to the version of PAXOS modified so as to allow consensus to be reached on a sequence of values, known as MULTIPAXOS and discussed by De Prisco in [6, 7]. The resulting algorithm lies somewhere in between majority voting and primary copy replication techniques. Its similarity to voting schemes comes from the use of majorities to achieve consistency, and it shares with primary copy techniques the need for a unique leader before termination can be guaranteed.

6. Conclusions

In this project, we have implemented a fault-tolerant mail service in order to measure the benefits and problems in using a strict-consistency model for a distributed mail service.

Our system meets the three initial criteria we set forth for FDMS.

- **Fault-tolerance and Correctness**. As long as a majority of the servers are alive and in the same network partition, our system will continue to make progress and maintain consistency. The Paxos algorithm ensures consensus in the presence of a unique leader and a majority of servers alive. Once consensus is achieved, two-phase commit guarantees that when a network partition containing the primary loses its majority, correctness is preserved.
- **Scalability**. Addition and removal of servers did not disrupt the normal

functioning of FDMS. New servers joining the system synchronized with the primary while allowing the other servers to continue functioning normally. This synchronization was performed using the mechanism built into the leader election sequence. Thus no additional complexity was added to the system design. Removal of servers did not disrupt our system due to the total replication of the mail file on all other servers, as long a majority of servers continued to function.

- **Performance**. System response time rose with initial increases in system complexity, but leveled off at the upper range of number of servers in the system tested. The results obtained for a single server system, were used to model a nonreplicated system. Our results for a multi-server FDMS system were competitive with the nonreplicated system performance.

7. References

[1] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp., Palo Alto, Sep. 1989.

[2] B.Lampson How to build a Highly Available System using Consensus. Distributed Algorithms, ed. Babaoglu and Marzullo, Lecture Notes in Computer Science 1151, Springer, 1996, pp 1-17.

[3] A. Birrell, R. Levin, M. Schroeder, R. Needham. Grapevine: an exercise in distributed computing. Communications of the ACM, April 1982, Vol. 25 Issue 4.

[4] Y. Saito, B. Bershad, H. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. Operating Systems Review 34(5): 1-15, Dec. 1999.

[5] D. Mills. Improved algorithms for synchronizing computer network clocks. In ACM SIGCOMM 1994, pp. 317-327.

[6] R. De Prisco, B. Lampson, N. Lynch. Revisiting the PAXOS algorithm. Theoretical Computer Science 243 (2000) pp. 35-91.

[7] R. De Prisco, Revisiting the Paxos algorithm, M.S. Thesis, Massachusetts Institute of Technology, LCS, Cambridge, MA June 1997. Tech Rep. MIT-LCS-TR-717, Lab. for Computer Science, MIT.