Fault Tolerant Master-Slave Replication and Recovery in Globe

Jeroen Ketema

2nd March 2001

Contents

Chapter 1

Introduction

Currently, the computer-research community puts a lot of effort in the development of distributed systems that are able to function in wide-area environments. Being able to function in such environments requires, among other things, scalability and fault tolerance. Some systems achieve scalability by means of data replication. The systems, for example, replicate websites at many servers. This makes it possible to have many simultaneous accesses to the websites.

Using data replication requires some form of consistency between the replicas. To achieve consistency special protocols are used, called consistency protocols. Combining replication with consistency protocols gives us replication strategies. These strategies are implemented by means of replication protocols.

1.1 Replication Protocols and Their Fault Tolerance

A disadvantage of most replication protocols used for scalability is their lack of fault tolerance. The protocols focus only on performance, not on availability in the face of failures. As a result, when a failure occurs, the replicated data may become inconsistent, or some parts of the data may even get lost. To overcome this situation, we need fault tolerant replication protocols. Unfortunately, literature gives little exact information on what these fault tolerant protocols should look like. Mostly, we find only very general descriptions,[1] and results of measurements.[2]

To overcome the shortcomings of literature, this thesis describes a fault tolerant replication protocol we designed. The protocol has three characteristic properties. First, the protocol implements the master-slave replication strategy, which we briefly describe next. Second, the protocol is usable in situations where replication is used for performance. Third, the protocol is fault tolerant in the face of crash failures. This last characteristic means the protocol is resilient in situations where computers, processes, and threads stop producing output until restarted.[3]

We now give a short description of the master-slave replication strategy. A full description can be found in Chapter 2. Starting at the foundation of the strategy, we must note that it revolves around three types of entities. These are the master, slave, and client entities. The master and slave entities hold the replicated data. Clients do not hold the data. The only function of the clients is to receive requests for operations on the replicated data, and to forward the requests to the slave entities. The number of clients may be arbitrary, as may be the number of slaves. However, there must always be exactly one master. The master is the only entity that may execute requests that change the replicated data. Slaves may not execute such requests. They may only execute requests that read the replicated data. All requests changing the replicated data must be forwarded to the master. After the master has executed a request that changes the data, it must send an update of the replicated data to all slaves.

1.2 Fault Tolerance and Globe

One of the wide-area distributed systems currently under development is Globe.[4] At the heart of Globe are distributed shared objects. These objects may replicate their data, as their name already suggests. To facilitate the replication each object includes its own replication protocol.

A shortcoming of the currently available version of Globe is the absence of fault tolerance. A first step towards fixing this can be the addition of a fault tolerant replication protocol. Unfortunately, simply adding a protocol is probably not enough. Scalability problems may arise, as a fault tolerant protocol is likely to differ from a protocol that is not fault tolerant and that implements the same replication strategy.

To partially perform the proposed first step of making Globe fault tolerant, this thesis shows how to fit into Globe the fault tolerant master-slave protocol we describe. To make the fit thorough we also identify potential scalability problems, and propose solutions to them.

In addition to a first step towards fault tolerance, fitting the protocol into Globe functions as a good example of a possible application of our master-slave protocol.

1.3 Recovery in Globe

Fault tolerant replication protocols are, of course, just one aspect of fault tolerance. Another important aspect is recovery. Recovery comprises both mechanisms and policies. Mechanisms make recovery possible. Policies determine what to recover and in what order to recover.

As a second step towards a fault tolerant Globe, this thesis looks at the recovery of object servers. These servers are a part of every Globe system. They are dedicated to holding the so-called representatives or local objects, which we describe in Chapter 4. With respect to the recovery of object servers, this thesis focuses on the mechanisms needed for local object recovery. In addition, this thesis also focuses on the policies that help to determine which local objects to recover and in what order to recover them.

1.4 Overview

All the above topics are discussed in the following chapters. Fist of all, Chapter 2 discusses our fault tolerant master-slave protocol and Chapter 3 assesses it. Thereafter, Chapter 4 describes how we can add our fault tolerant protocol to Globe. In addition, the chapter identifies a number of potential scalability problems and proposes solutions to them. Finally, Chapter 5 deals with recovery of object servers, and Chapter 6 gives some conclusions. Chapter 6 also gives some ideas about what can be done in the future as a follow-up to this thesis.

Chapter 2

The Fault Tolerant Master-Slave Protocol

In this chapter, we describe the fault tolerant master-slave protocol we designed. The protocol is fault tolerant with respect to crash failures.

Before we describe the fault tolerant master-slave protocol, we make explicit in Section 2.1 the assumed system in which the protocol must be able to function. In the two sections thereafter, we describe the fault tolerant master-slave protocol. The description proceeds in two steps, one per section. In the first step, a non-fault tolerant protocol introduces the mater-slave replication strategy. Thereafter, the second step gives our fault tolerant protocol. After the steps, the final section of this chapter discusses some adaptations to the fault tolerant protocol that improve its efficiency.

2.1 The Assumed System

In essence, the assumed system is simply a set of computers with processes running on them. The processes can communicate by means of a network that connects all computers.

The assumed network is constrained in the following two ways:

1. The network must be synchronous
2. The network connections must be First-In First-Out (FIFO)

The first constraint signifies that every message sent through the network must arrive within a fixed amount of time. Unfortunately, most networks are not consistent with this constraint. They deliver most messages within a fixed amount of time, but not necessarily all of them. We say more on this problem, and the consequences it has for our fault tolerant protocol, in Chapter 3.

The second network constraint says the network must maintain the order in which processes send messages over individual network connections. We can easily satisfy this constraint by associating a sequence number with every message.

Although many different failures exist, our master-slave protocol is fault tolerant only in the face of crash failures. For that reason, we limit the types of failures that can occur in the assumed system to crash failures. In combination with the synchronous network assumption, allowing only crash failures makes it possible to detect crashes. As crash failures cause components to stop producing output, such components can no longer send messages. Accordingly, as the network makes sure all messages arrive within a fixed amount of time, the cause of another component missing a message must be a crash failure.

The non-fault tolerant master-slave protocol given in the next section does not completely adhere to the assumed system. As the protocol is not fault tolerant, it assumes a system completely free of failures.

2.2 The Non-Fault Tolerant Master-Slave Protocol

In Chapter 1, we already gave a brief explanation of the master-slave replication strategy. In this section, we give a more thorough explanation. In addition, this section presents a non-fault tolerant protocol that implements the master-slave replication strategy.

Before we begin our explanation, we must note that the master-slave replication strategy is often referred to as the primary-backup replication strategy. When this alternative is used, the master entity is called the primary and the slave entities are called backups.

2.2.1 The Master-Slave Replication Strategy

As said before, the master-slave replication strategy uses three types of entities, called master, slave, and client. There is always one master. However, numerous slaves and clients may exist. The master and slaves hold the replicated data. The clients, on the other hand, act only as front-ends for processes that use the replicated data. In that way, the processes using the data, or users for short, do not need to know where the replicas of data are.

As implementations often represent entities as processes, we from now on refer to entities simply as processes.

A user can request two types operations on the replicated data: read requests and read/write requests. Read requests only read the replicated data. Read/write requests may also update the replicated data. A user must send both types of requests to a client. When a client receives a request, it must forward the request to a slave. Thereafter, when the slave receives the request, two things can happen. In case of a read request, the slave executes the request, and it sends a reply to the client from which it received the request. Following this, the client returns the reply to the user. In case of a read/write request, the slave forwards the request to the master. The master then executes the request, after which it sends an update of the replicated data to every slave. This makes sure the slaves have an up-to-date version of the replicated data. In addition to sending the updates, the master returns a reply to the slave that forwarded the request. After receiving the reply, the slave in turn returns the reply to the client that sent the request. Finally, the client returns the reply to the user.

There are two requirements to which the handling of read requests and read/write requests must adhere. The first requirement is that the master and the slaves must apply all updates in the same order. This makes sure no inconsistencies arise between replicas due to different update orders. The second requirement is that when a user sends a read/write request and thereafter sends another request, the other request must operate on a version of the replicated data that includes the changes made by the read/write request. This requirement ensures that every request of a user operates on the correct data when it causally relates to an earlier request of that same user.

A requirement of a different kind is that the master can send the updates to every slave only when it knows where every slave is. To provide the master with the locations of the slaves, every new activated slave could send an announcement message to the master. An additional advantage of sending this message is that it makes it possible for the master to send a copy of the replicated data to the activating slave. A problem, unfortunately, is that the slave somehow needs to find out the address of the master. However, we can easily achieve this by using a directory service that holds the required address. Similar to the previous problem, clients need the addresses of slaves. We can solve this problem too by using a directory service.

A consequence of the previous requirements on addresses is that the different types of processes can become active only in a very specific order. First, the master needs to become active, as slaves need its address. Then, before clients can become active, at least one slave needs to become active, as clients require slave addresses. Finally, other slaves and clients can become active in an arbitrary order.

As a final remark, we note must that the slave described above can incorporate the functions a client fulfils, and that the master can incorporate the functions of both clients and slaves. We did not add these functionalities to keep our discussion simple. However, it should be easy to derive the extended master and slaves from the above description.

2.2.2 The Non-Fault Tolerant Master-Slave Protocol

In what follows next, we explain a non-fault tolerant master-slave protocol that functions as an introduction to the fault tolerant protocol. During the explanation, we use the state diagrams of Figures 2.1, 2.2, and 2.3. There is one state diagram for every type of process. Essentially, the diagrams are directed graphs, in which the vertices are states, and in which the directed edges are state transitions. A state transition that does not start in any state denotes a start state. The state named after the process in question, is the state in which the process waits when not handling any event.

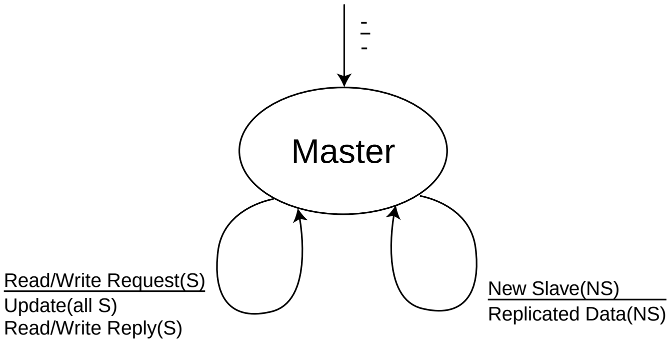


Figure 2.1: State diagram for the master process of the non-fault tolerant protocol

The state diagrams associate a label with every state and every state transition. The label of a state makes it easy to refer to that state. The label associated with a state transition describes two things. First, the label describes the cause of the transition. This can be a message sent by process, or a request submitted by a user. Second, the label also gives the externally visible reaction of the process in question. This reaction may be a message sent to one or more processes, or a reply returned to a user. A horizontal line separates both parts of a transition label. The cause is always above the line, and the response always below it. The labels of the transitions do not show the internal changes caused by state transitions. We mention these changes in the text only.

In the labels associated with the state transitions, both the text of the cause and the text of the reaction may consist of a single dash. In case a cause is a dash, no external event is necessary for the transition to take place. In case of a response, a dash means no reaction is visible externally. Behind every cause and response that is not a dash, we find one or more letters between parentheses. Above the line in a state transition label, the letters tell us who caused an event. Below the line, the letters tell us for whom a reaction is intended. Table 2.1 gives an overview of the possible letter combinations.

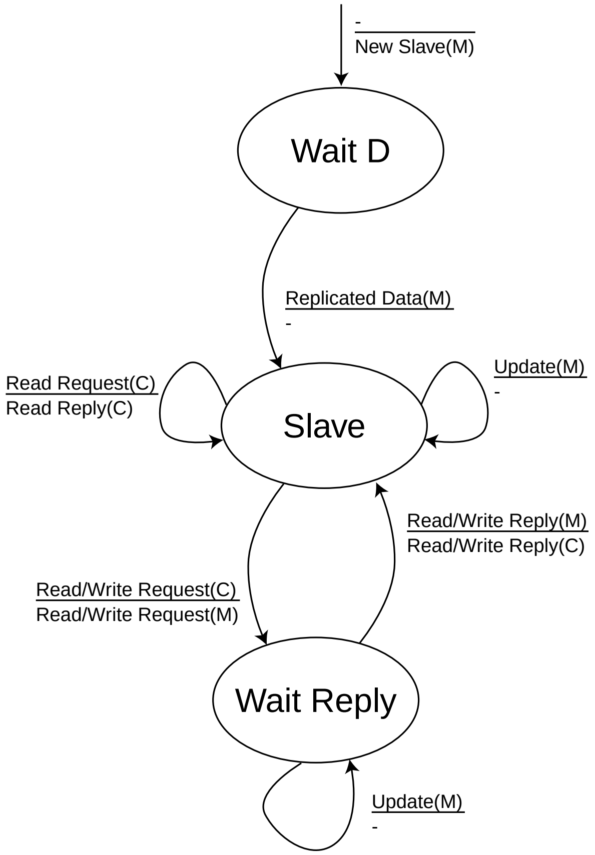


Figure 2.2: State diagram for the slave processes of the non-fault tolerant protocol

We now explain the non-fault tolerant protocol by means of scenarios that refer to the state diagrams. The scenarios revolve around the events given in the general explanation of the master-slave replication strategy. Only two types of events are possible: activation events and requests sent by users. There are three activation events: master activation, slave activation, and client activation. The sending of requests has two possibilities: sending of a read request and sending of a read/write request. While looking at the scenarios, keep in mind the protocol functions within the system described in Section 2.1.

Master Activation

As a first scenario, we consider the activation of the master. When the master becomes active, it enters the only state of Figure 2.1. No externally visible actions take place. Of course, when slaves use a directory service to locate the master, the moment of activation is a suitable one for registration at the directory service.

Slave Activation

Activation of slaves is more complicated than master activation. The first thing a slave needs to do is to find out the address of the master. It can possibly do this by contacting a directory service. Thereafter,

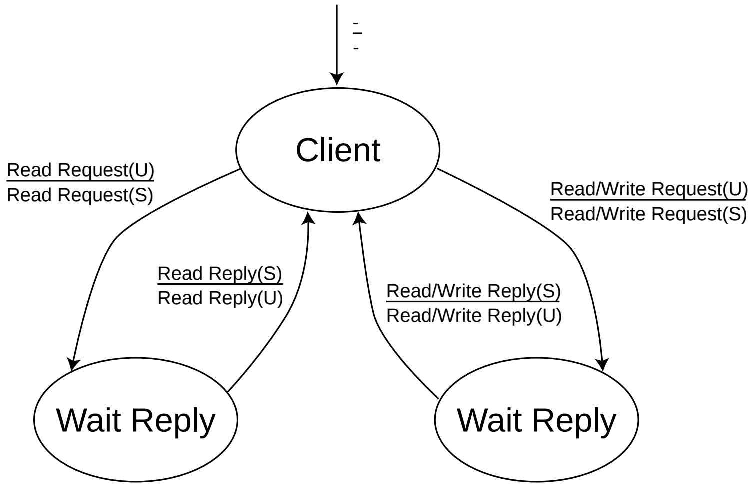


Figure 2.3: State diagram for the client processes of the non-fault tolerant protocol

Table 2.1: Meaning of all possible letter combinations in Figure 2.1, 2.2 and 2.3

Letter combination	Meaning
M	Master
S	Slave
NS	New slave (an activating slave)
All S	All known slaves
C	Client
U	User

the slave needs to let the master know it is there. The slave accomplishes this by letting the slave send a message with address information to the master. This is the New Slave message at the top of Figure 2.2. After sending the message, the slave waits for a reply in the Wait D state. Eventually, the master receives the New Slave message. When this happens the master replies by sending a copy of the replicated data, as the master state diagram shows. The master also saves the information sent by the slave, so it can send updates later on. When the slave receives the replicated data, a transition takes place to the Slave state. This makes the slave fully activated. Like the master, the slave may need to register at a directory service. A good moment for doing this is just after the reception of the replicated data, as the slave is fully usable only from of that moment.

Client Activation

Like the master, a client does not need to execute any externally visible actions during activation. However, a client needs the address of a slave to be able to operate. An appropriate moment for acquiring such an address is the moment of activation. Alternatively, the client can also wait until a user submits a request, as the client needs the address only then. Activation brings a client in the Client state of Figure 2.3.

Sending a Read Request

When a user sends a read request to a client, the client forwards this request to a slave. The client does this only when in its Client state. When it is in any of its other states, it is busy handling a request, and the handling of the newly sent request must wait. Nevertheless, the client eventually forwards the request, and one of the slaves receives it. When this happens, the slave executes the request, and returns a reply to the client. Similar to the client, the slave handles the request only when in its Slave state. After the client forwards the request, it waits for the reply in the Wait Reply state on the left side of Figure 2.3. When the client eventually receives the reply, it returns the reply to the user, and a transition takes place back to the Client state.

Sending a Read/Write Request

Comparable to the case of a read request, a client forwards a sent read/write request to a slave. However, unlike the in case of a read request, the client waits in the right Wait Reply state of its state diagram. In addition, the slave forwards the request to the master. The master executes the request upon reception. After execution, the master sends an update of the replicated data to every slave. The slaves can handle this update in both their Slave state and their Wait Reply state. Besides sending updates, the master also sends a reply to the slave that forwarded the request. This slave waits for the reply in its Wait Reply state. After the slave receives the reply, it returns to the Slave state. The slave also sends the reply to the correct client. That client returns the reply to the user that sent the request.

We have now described all possible scenarios. However, we did not describe how the protocol satisfies the requirements for ordering updates, and for executing requests with an up-to-date version of replicated data. Fortunately, the protocol can satisfy both requirements by making use of the FIFO network assumption stated in Section 2.1, by only allowing one network connection between every pair of processes, by letting every user always use a unique client, and by letting every client always use a unique slave.

Let us first look at how we can guarantee that both master and slaves apply all updates in the same order. What we know is that the master executes all read/write requests, which are the source of every update. Therefore, all slaves must apply all updates they receive in the same order the master executes the corresponding read/write requests in. The master can easily let the slaves know what the order is by sending all updates in the order of the request execution. As the master uses a single network connection per slave to communicate, and as all network connections are FIFO, the network delivers all updates in the order the master sends them in. Consequently, the slaves know the correct order for applying the updates.

We now turn to the requirement that a request sent by a user following a read/write request sent by that same user, must execute with an up-to-date version of the replicated data. We must consider two cases, as there are two types of requests. However, what holds in both cases, is that the request following the read/write request is not handled until the reply to the read/write request returns. This is the consequence of users always using the same client. The client handles only one request at a time, and the FIFO assumption that makes certain the client receives the requests in the correct order.

Looking at the case where both requests are read/write requests, we see that the master must handle them both. As the client sends the second read/write request only after it receives a reply to the first one, the master knows the changes to the replicated data made by the first request. Consequently, it can execute the second request with an up-to-date version of the replicated data.

If we now look at the case in which the second request is a read request, we see that we have a situation in which both the master and a slave must handle a request. What we know is that the master uses only a single connection to communicate with the slave that forwards the read/write request. Therefore, because the master sends the update associated with the request just before it sends the reply associated with the request, the FIFO assumption makes sure the slave knows which update is associated with the request. As a client always uses the same slave, that slave must see both the read/write request and the read request. Consequently, as the client makes sure the slave does not receive the read request until after receiving the reply to the read/write request, the slave knows which changes belong to the read/write request before it receives the read request. As a result, the slave can execute the read request with an up-to-date version of the replicated data.

We must note the above protocol lacks the ability to shutdown processes. However, a shutdown very much resembles a crash failure, as both stop the production of output. As we discuss crash failures only in the next section, we decided to also postpone the inclusion of shutdowns until that section.

2.3 The Fault Tolerant Master-Slave Protocol

In this section, we present our fault tolerant master-slave protocol by extending the non-fault tolerant protocol of the previous section with crash failure resiliency. In addition to crash failure resiliency, we also extend the non-fault tolerant protocol by allowing processes to shutdown.

The basic idea behind our fault tolerant protocol is that every process is expendable. What this means depends on the type of process.

In the case of the master process, being expendable means another process must be able to take over the role of the master. We decided every slave process must be able to take over the role. Slave processes are suitable, as they possess the replicated data. To make the take over completely possible we decided every slave must also possess information on the other slaves present, just like the master.

For slaves, expendability means something different. As we replicate for performance, it is likely there are multiple slaves. This means other slaves can simply take over the clients of a slave that shuts down or crashes. There are two concerns here. First, the master needs to notice the shut down or crashed slave so it can stop sending updates. Second, we may need to activate a new slave to get the best performance. The protocol solves the first concern by letting the master run a crash detection mechanism for the detecting slave crashes, and by letting slaves send a message to the master when they shut down. The second concern can be solved by allowing the protocol to dynamically start new slave. We did not include this in the protocol. However, it should be easy to add this functionality.

In case of clients, expendability means we allow a user to start a new client when a shutdown or a crash occurs.

As can be concluded from the above explanation, we handle shutdowns in about the same way as crashes. We can do this, as shutdowns very much resemble crash failures, for both stop the production of output. As we see later in this section, we do not handle shutdowns and crashes in completely the same way because the crash detection mechanism we propose may be slow. We take some measures to circumvent the crash detection mechanism during shutdowns.

Once again, we use state diagrams and scenarios to describe the protocol. Figures 2.4, 2.5 and 2.6 give the state diagrams, one for every type of process. The diagrams contain the same elements as those in the previous section. However, there are also five new elements. First, some of the state transition labels now include text in italics. These italics briefly describe a condition that must be satisfied before a process may take the transition associated with the label. Second, all state diagrams now include an end state. A process reaches the end state when it shuts down. Vertices with a double border mark the end states. Third, a state transition can now have a number of causes. In the labels associated with state transitions, backslashes separate the various causes. Fourth, some transitions do not have letters between parentheses behind the cause. The letters are missing, because it is impossible to point out a process or user from which the event originates. The scenarios below give more details on this. Fifth, three new letter combinations appear between parentheses: NM, all S - NS, and all C. Table 2.2 gives the meaning of these new letter combinations together with the combinations that already occurred.

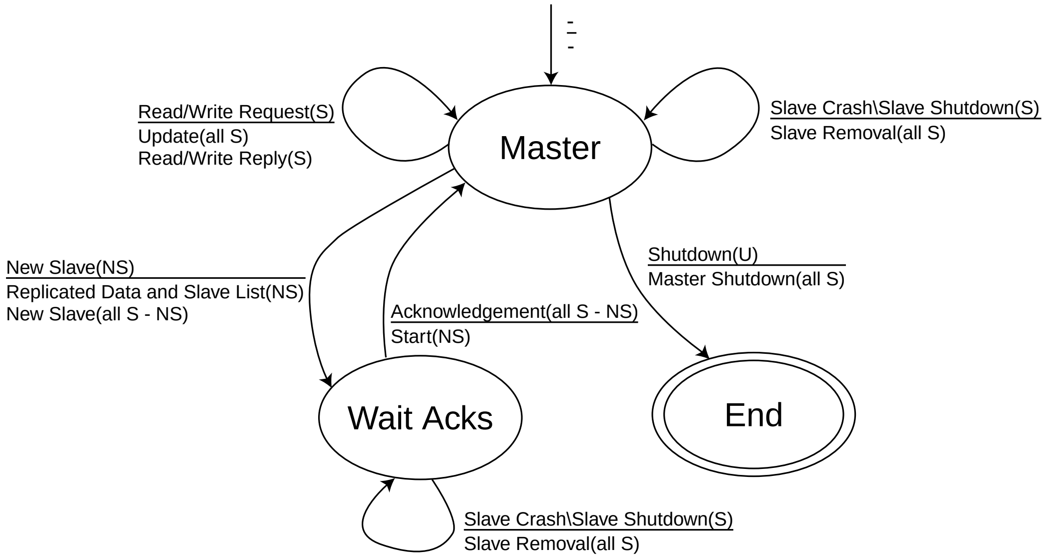


Figure 2.4: State diagram for the master process of the fault tolerant protocol

Table 2.2: Meaning of all possible letter combinations in Figure 2.4, 2.5 and 2.6

Letter combination	Meaning
M	Master
NM	New master after master crash
S	Slave
NS	New slave (an activating slave)
All S	All known slaves
All S - NS	All known slaves except a new slave
C	Client
All C	All clients associated with a specific slave
U	User

What is missing from each of the three state diagrams is a crash state. We left these states out on purpose. They would only obscure the state diagrams, for a transition to a crash state is possible from every state except the end states.

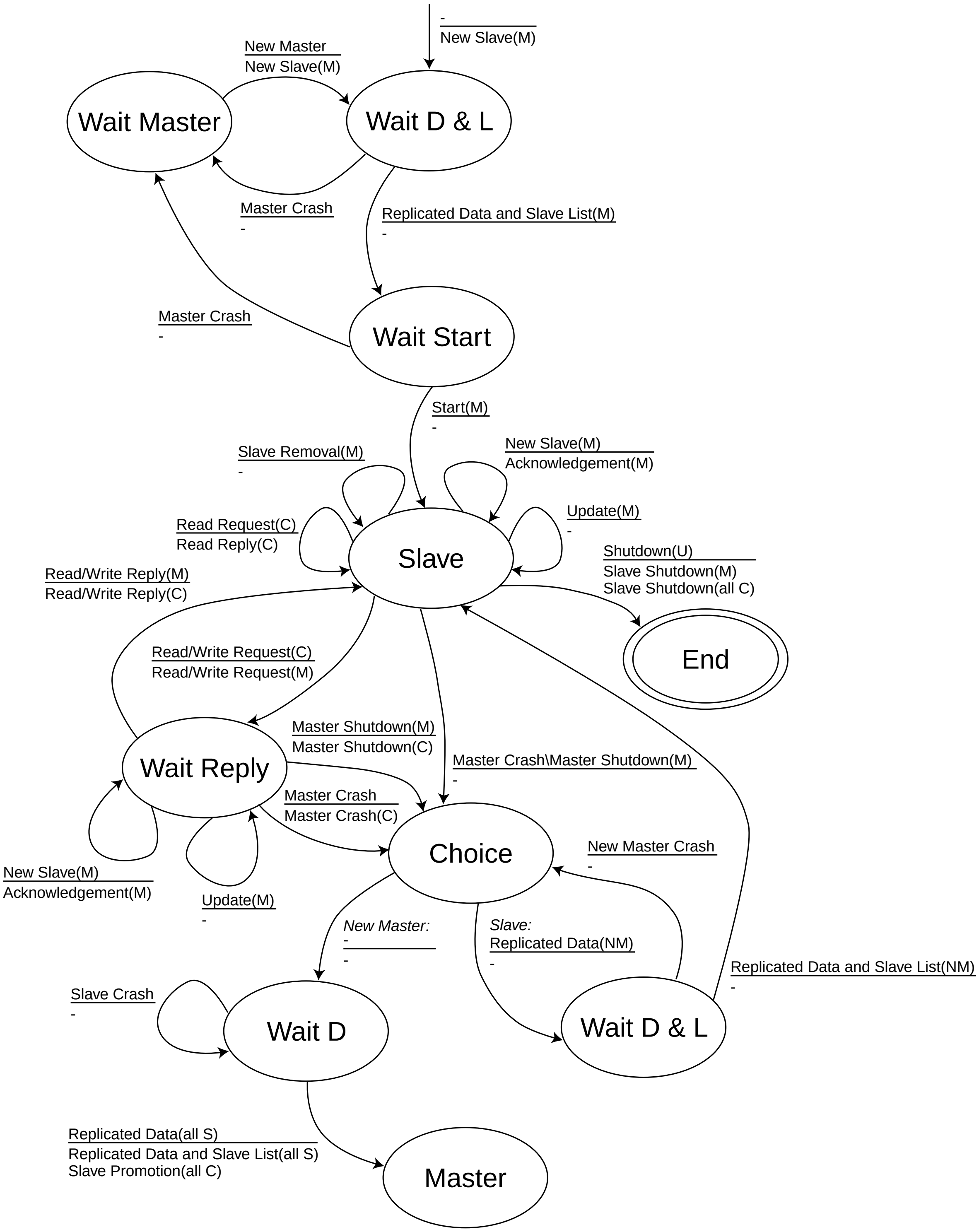


Figure 2.5: State diagram for the slave processes of the fault tolerant protocol

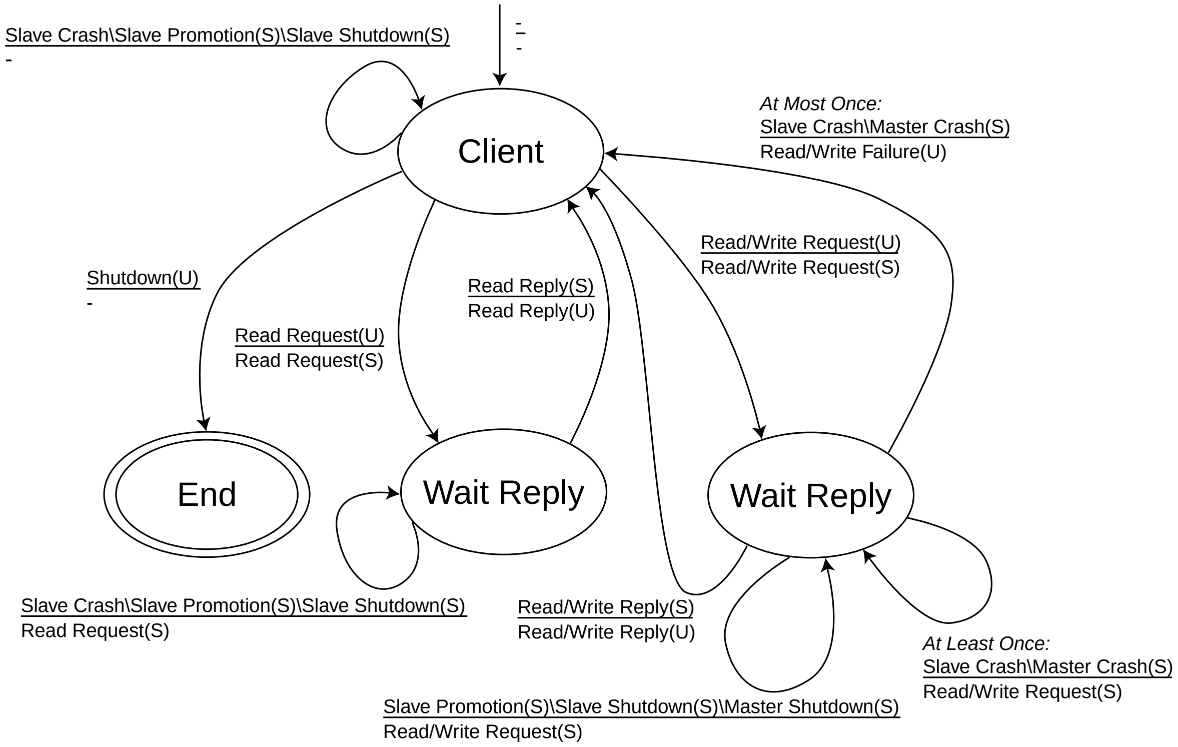


Figure 2.6: State diagram for the client processes of the fault tolerant protocol

We next give the scenarios. Although we give quite a few, we do not give all of them. There are simply too many. The scenarios we do describe, cover the activation of a slave, the crash of a slave, the crash of the master, the crash of the master or a slave during slave activation, a master or slave crash during the handling of a read or read/write request, and shutdowns of the different types of processes. We do not cover the activation of the master, the activation of a client, the handling of read requests, and the handling of read/write requests. These scenarios are the same as in the case of the non-fault tolerant master-slave protocol. In addition, we do not describe how a user must act when detecting a client crash. This simply requires the activation of a new client by the user, and when a request is in progress it also requires some mechanisms very similar to those of a client handling a request when a slave crash occurs. All other scenarios we do not cover are simple combinations of the given ones.

Slave Activation (Without Crashes During Activation)

Slave activation in the fault tolerant protocol differs form slave activation in the non-fault tolerant protocol, in contrast to master activation and client activation. Slave activation is different, as we want each slave to able to take over the role of the master. As already explained, we accomplish this by requiring each slave to know which other slaves are present. With respect to the activation of a new slave, this means every slave must learn about the new slave. In addition, it also means the new activated slave must learn about every other slave. Both the master and the slaves save address information about the slaves they think are active in a data structure we call the slave list.

Slave activation begins with the new slave sending a New Slave message to the master. This message contains address information. When the master receives the message, it adds the information to its slave list. Thereafter, the master sends a copy of the replicated data and a copy of its slave list to the new slave. Sending the slave list makes certain the new slave learns about all other slaves. The master also forwards the New Slave message to all other slaves, so they learn about the new slave too. The new slave waits for the replicated data and the slave list in the Wait D & L state at the top of Figure 2.5. When the new slave receives the data and the list, a transition takes place to the Wait Start state. The new slave waits in the Wait Start state until it receives a Start message from the master. The master does not send this message immediately. It first waits in the Wait Ack state for an acknowledgement to arrive from every slave that is not new. The slaves send these acknowledgements after they receive the forwarded New Slave message in their Slave state or Wait Reply state, and after they add the information from the message to their slave lists. The slaves do not handle the New Slave message in any of their other states, as they can be in those states only during their activation, or when the master crashes. In both cases, the master cannot handle slave activations. In the first case because it is already handling one. In the second case because it is no longer there. Nevertheless, eventually the new slave receives the Start message. It is then completely activated.

Of course, the process of slave activation should be capable of withstanding crash failures and shutdowns. Therefore, one of the other scenarios discusses crash failure resiliency during slave activation. That scenario also sheds light on the reason for using the acknowledgements and the start message.

Slave Crash (No Process is Handling an Event)

We now come to a first scenario that involves a crash failure. To be precise, the scenario involves the crash failure of a slave.

When a process detects a crash failure, it is likely the process must take some measures. The state diagrams depict this as state transitions. The text of the cause associated with such state transitions always reads Slave Crash. Note there are not letters between parentheses behind the Slave Crash cause. The letters are missing, because the reason for a crash may be unknown.

Almost all processes need to take some measures when a slave crash occurs. The only entities that do not need to do so are the client entities that forward the requests they receive to other non-crashed slaves. The master must take measures to stop sending updates to the crashed slave and to remove the slave from its slave list. The slaves that did not crash must also remove the crashed slave from their slave lists. The clients using the crashed slave to forward requests must start to use other slaves.

Assuming no events take place during a slave crash, the protocol starts with the master detecting the crash in the Master state and the relevant clients detecting the crash in their Client states. The master responds to the crash by removing the crashed slave from its slave list. As we may assume the master uses the slave list to decide whom to send updates to, removing the slave makes certain the master no longer sends updates to the crashed slave. The master also sends a Slave Removal message to all non-crashed slaves. After the slaves receive this message in their Slave state, they can also remove the crashed slave from their slave lists. When a client detects a slave crash, it responds by finding another slave to which it can forward requests. How to find another slave depends on the system. When the system uses a directory service to store slave addresses, querying the service may be enough. Of course, for the finding of slaves to work efficiently, the addresses of crashed slaves should be removed from the directory service. The master, for example, can do this when it detects the slave crash. When the addresses of crashed slaves are not removed from the directory service, a client may receive an address of a crashed slave. This is not a problem as long as the client queries for multiple addresses, and as long as it detects crashes while trying to contact found slaves.

The question now is, how exactly to detect a slave crash, or for that matter any crash. As Section 2.1 explains, we can detect crashes by looking for missing messages. There are now two possibilities for which messages to use. We can use the messages already needed by the protocol, or we can use special newly introduced messages. A problem with using the protocol messages is the irregularity in the sending of these messages. As the processes send the protocol messages in reaction to events, the time between a crash and the detection of the crash becomes dependent on the frequency in which events happen. With special messages, this is not a problem. The processes can send the messages at regular intervals. For example, processes can send *I-am-alive* messages every few seconds to tell they are still there and did not crash.

Master Crash (No Process Is Handling an Event)

As a second scenario involving crash failures, we discuss a crash of the master process. The scenario assumes no requests, slave activations, and shutdowns are in progress.

In the previous scenarios we described how each slave learns which other slaves are present. The current scenario shows how the slaves use this information to replace the master. In essence, all slaves must just select the same slave from their slave list, and that slave then becomes the new master. However, the slaves do not necessarily all have the same slave list at their disposal. There can be minor variations. Some slaves may have received certain messages that mutate their slave lists, such as New Slave messages and Slave Removal messages, while other slaves still need to receive these messages. The protocol solves this problem by making certain that the set of slaves in each slave list is a superset of the set of active slaves, and using only this property.

To begin with, let us assume each slave does know exactly which other slaves are active, and that the slave becoming the new master does not crash during its instatement as master. When the current master now crashes, each slave detects this in its Slave state. A slave cannot be in any other state, as we assumed no event handling is in progress. After detecting the master crash, each slave follows the transition to the Choice state in Figure 2.5. In this state, each slave determines, in the same deterministic way, which slave becomes the new master. After a slave has made a choice, it removes the chosen slave from its slave list. The slave does this as the chosen slave becomes the master, and as a master does not perform slave activities. After removing the slave from the slave list, two transitions are possible. First, if a slave chooses it self as the new master, it follows the transition denoted by the text in italics reading New Master. Second, if the slave does not become the new master, it follows the other transition possible from the Choice state.

When a slave selected another slave as the new master, the slave sends a copy of its replicated data to the new master. The new master waits for the replicated data from all slaves in the Wait D state of Figure 2.5. After the master has received all copies of data, it computes the most recent version of the replicated data. The new master does this, as we like the new master to continue operation with the most recent version of the replicated data. The new master may not have this version, as it may have detected the crash of the old master before it received all updates sent by the old master. When done computing, the new master sends the computed data, together with a copy of its slave list, to all slaves. Why the new master sends the slave list becomes clear later in the current scenario. After the slaves have received the data and the slave list, they use it to update their replicated data and slave list. After updating, the slaves continue their normal activities. The new master does not continue its slave activities. For that reason, the new master informs all clients that forward requests to it that it is no longer a slave. The new master does this by sending a Slave Promotion message to all clients. As the new master does not necessarily know which clients forwarded requests, a multicast channel can be helpful. Alternatively, the master can send a Slave Promotion message to a specific client when it receives a request from that client.

While the new master waits in the Wait D state, it is possible a slave crashes. If the slave does this before sending replicated data, the new master waits forever for data from the slave to arrive. To overcome this, we require the new master to detect slave crashes while in its Wait D state. When detecting a slave crash, the new master can make sure it no longer waits. In addition, the new master can also remove the crashed slave from its slave list, so that when it sends the slave list together with the most recent version of the replicated data, the other slaves also learn a slave crashed.

Although up until now we assumed that each slave knows exactly which slaves are active, this may not be the case. Some slave crashes may not have been detected yet, and some messages mutating slave lists may not have been received yet. However, as the current scenario assumes only slave crashes and no slave activations, it is clear that each slave must at least have knowledge about a superset of the set of active slaves.

If we look at the superset, we see that a slave can inadvertently choose a slave that is no longer active as the new master. As the choosing slave sends its replicated data to a non-existing process and then waits for a reply, it waits forever, as a non-existing new master cannot reply. To solve this problem, we require the choosing slave to detect the non-existence. The slave must do this while in the Wait D & L state at the bottom of Figure 2.5. The slave can detect the non-existence as a crash, as no messages originate from non-existing processes. When the slave detects the crash, a transition can take place back to the Choice state. Back in this state, the slave can choose another slave from its slave list. The slave already removed the non-existing slave the previous time it left the Choice state, so it cannot choose that slave again.

The last assumption we need to drop in the current scenario, is the assumption the new master does not crash during its instatement. If we drop the assumption, a situation can occur in which some slaves receive the message with the computed version of replicated data, while other slaves do not receive it. For the slaves receiving it, the crash looks just like a regular master crash, as they return to the Slave state after the reception of the data. The other slaves, however, are still in the Wait D & L state. In this state, a slave detects the non-existence of a new master, but as detection takes place by looking for missing messages, the slave can also detects a crash of a new master. Detecting the crash brings the slave back in the Choice state, making it possible to choose another new master.

Like earlier scenarios, the current scenario requires making changes in a directory service when one is used. For one thing, the address of the crashed master must be removed from the service, as the master is no longer available. In addition, it is necessary to change the entry of the new master, as the new master is no longer a slave. It is also necessary to remove the address of slave from the directory service when the new master detects a crash of that slave in its Wait D state. Obviously, the new master can perform all three operations on the directory service.

Process Crash During Slave Activation (No Process Is Handling any other Event)

We now come to a third scenario involving crashes. In this scenario, master and slave crashes occur during slave activation. Below, we first deal with the slave crashes. Thereafter, we look at master crashes. As in the previous scenarios, we assume no requests and shutdowns are progress.

The problem with slave crashes during slave activation is the requirement that each slave needs to send an acknowledgement to the master after receiving a New Slave message. When a slave crashes before it sends the acknowledgement, or before it even receives the New Slave message, the master waits forever for the acknowledgement to arrive. To circumvent this problem, the master must detect slave crashes while in its Wait Ack state. The Wait Ack state is the state in which the master waits when trying to receive the acknowledgements. If the master detects a slave crash, it can make sure it no longer waits for the acknowledgement from the crashed slave. In addition, the master can remove the crashed slave from its slave list, and send a Slave Removal message to all other slaves, including the new activated slave. Sending the Slave Removal message makes sure the slaves learn about the slave crash. Note that this solution is very similar to what the new master does while receiving the replicated data from the slaves.

When the master crashes during slave activation, the problem is more severe. Situations can occur in which some slaves know about a new slave, while other slaves do not know about it. These situations can occur, as not all slaves may have received a New Slave message yet. A consequence is that it is possible that some slave lists do not contain a set of slaves that is a superset of the set of active slaves. When the new slave now participates in a new master instatement, the new slave may need to send its replicated data to a process that does not know about it. The process is not expecting the data, and may thus have stopped waiting. To solve this problem we require the new slave to deactivate when it detects a master crash. When the new slave deactivates, every non-crashed slave again has a superset of the set of active slaves.

The problem of the new master not knowing about a new slave is the reason the protocol uses the mechanism with the acknowledgements and the Start message during slave activation, which we explained in the slave activation scenario. By using the mechanism, the new slave knows when all other slaves know about it, and when it does not need to deactivate anymore when the master crashes.

The slave state diagram shows the deactivation of a new activated slave as a state transition. To be precise, the transitions from both the Wait D & L state and the Wait Start state to the Wait Master state are deactivation transitions. When a new slave reaches the Wait Master state, it waits until a master becomes available again. Thereafter, the new slave tries to run its activation procedure once more. It does this by taking the transition back to the Wait D & L state.

The question, of course, is how to detect that a new master is available. One answer to this question is to use a directory service. If we require the directory service to contain the address of the master, and if a new master changes this address to its own address during its instatement, then a deactivated slave just needs to poll the directory service until the address changes.

Process Crash During Request Handling

Up until now, all scenarios involving crash failures assumed no read requests and read/write requests were taking place. The current scenario drops this assumption. As a result, clients and slaves can be in their Wait Reply states when a process crashes.

Let us first look at what happens when a slave crash occurs while a read request is in progress. In this case, a problem emerges when a client forwards the request to a slave that crashes, and when the crash occurs before the slave returns a reply to the client. Obviously, the client needs to wait forever for the reply to arrive. To prevent this, the client must be able detect when the slave it is waiting for crashes. Then, when detecting a crash, the client can search for another slave, and it can forward the read request to the found slave. Doing this makes certain the client eventually receives a reply it can return to the user. The client can forward the read request multiple times without any problems as read requests are idempotent operations.

When the master crashes while a read request is in progress, a problem occurs when a client sends the read request to the slave that becomes the new master. A new master does not handle read requests, and therefore, the client again waits for a reply that never arrives. However, the new master sends a Slave Promotion message during its instatement. Consequently, when a client is able to receive such a message while waiting for a reply, we can overcome the endless waiting. After the client receives the message, it can search for another slave, and forward the read request to that slave.

We now come to read/write requests. Unlike read requests, read/write requests need not to be idempotent. To overcome this difference, the master could remember the requests that led to the current version of the replicated data. If a client then forwards an already executed request, the master can simply discard the request. Of course, the master does need to return a reply, for a client forwards a request once more only when it did not receive a reply. When the master saves the reply associated with every executed request, returning a reply once more becomes possible.

Associating a unique identifier with every read/write request can help to remember the executed requests.[5] When the master stores the identifiers of all executed requests, it can compare the identifiers of newly received requests with the ones stored. As a result, the master can determine precisely which requests still need execution. To make this work completely, we need to store the identifiers alongside every copy of the replicated data. This makes sure that after a crash of a master the new master also knows which requests led to the current version of the replicated data.

A disadvantage of using unique identifiers when master crashes can occur is that it requires simultaneous sending of the identifier and the update associated with each request. Unfortunately, this is not always possible, in particular when requests may address peripherals. Consider, for example, a request involving the use of a printer. In this case, there are two possibilities for what the master can do. The master can print before it sends the update and the unique identifier, or it can print afterwards. If the master prints before sending and crashes directly after the printing, the slaves think the master did not execute the request. On the other hand, if the master does the printing after sending, and crashes just before the printing, the slaves think the master executed the request, while in reality it did not. To solve this problem we do not use unique identifiers. What we do is that we differentiate between read/write requests where it does not matter if the master executes them more than once, and requests where it does matter. Requests where its does not matter are called at-least-once requests. Requests where it does matter are called at-most-once requests.

Let us now look at what problems emerge when a slave crashes during the handling of a read/write request. Like in the case of a read request, a client may be waiting for a reply from a crashed slave. Therefore, the solution is again to let the client detect when the slave it is waiting for crashes. After the client detects a crash, it can search for another slave. What happens thereafter depends on the type of request. If the request is an at-most-once request, the client returns a failure to the user that submitted the request, and it returns to the Client state. The client does this as the request is an at-most-once request, and as execution of the request may have completed and only the reply may have been lost. If the request is an at-least-once request, on the other hand, the client can forward the request again. In case of an at-least-once request, it does not matter how often we execute the request.

When the master crashes during the handling of a read/write request, two problems can occur. The first of these occurs when a slave is waiting in the Wait Reply state for the master to return a reply. When the master crashes, the slave does not receive a reply and it waits forever. To overcome this, we require slaves to detect master crashes in their Wait Reply state. When a slave then detects a master crash, it can run the procedure of selecting a new master. The slave also needs to do something about the missing reply. To complicate the slave not more than necessary, we let the slave return only a Master Crash message to the client that sent the request. The client needs to be able to detect this message while waiting, and when it detects the message, it has to decide, depending on the request, if it wants to forward the request once more.

We now come to the second problem that can occur when the master crashes during the handling of a read/write request. This problem occurs when a slave did not start to handle the read/write request when it detects a master crash. In this case, the slave also needs to select a new master. The problem now emerges when the slave becomes the new master. The client waits for a reply from the slave, but as the slave is no longer a slave, it does not handle the request form the client. However, the slave sends a Slave Promotion message. If we let the waiting client detect this message, we can make sure it does not wait forever. When the client receives the Slave Promotion message, it can search for another slave and forward the request to that slave. It can always forward the request, as it receives a Slave Promotion message only as the first message from the slave that becomes master, when the slave was not already handling the read/write request. When the slave was handling the request, the client receives a Master Crash message before the Slave Promotion message. This is a consequence of the solution to the previous problem and of the assumption made in the previous section that only a single FIFO connection is present between every slave and each of its clients.

Shutdowns

We now arrive at our last scenario, which discusses shutdowns. We based the handling of shutdowns on the fact that shutdowns look like crashes, for both stop the production of output. When a process shuts down, we let other the processes perform operations that are equal or similar to the ones they perform in the case of a crash. Evidently, shutdowns and crashes are not entirely the same. Shutdowns mostly offer time to perform some operations, which is something that is not true in the case of a crash. We exploit the difference by letting processes handle shutdowns only in the state in which they are not handling any other event. In the state diagrams, these states are the states with the name of the processes. Handling shutdowns only in these states simplifies the processes. They do not have to check for their shutdown in every possible state.

Let us now start describing shutdowns by looking at client shutdowns. In this case, only users are affected. The master and slaves do not know the client exists. As shown in Figure 2.6, a user initiates a client shutdown. Consequently, it is reasonable to assume the user no longer needs the replicated data to which the client provides access, and the user thus does not need to start another client to which it can send future requests. Even though this is true, other users may still require the client that is being shutdown. These users do need to start a new client.

When a slave shuts down, it sends a Slave Shutdown message to both the clients forwarding requests to it, and to the master. Sending the message to clients possibly requires a multicast channel, as the slave does not need to know its clients. The slave sends the Slave Shutdown message to circumvent the crash detection mechanism, which is likely to function slower than sending a message. We require both master and clients to be able to receive the Slave Shutdown message in every state in which they are able to detect slave crashes.

The master reaction to the Slave Shutdown message is the same as its reaction to a slave crash. It is the same, as a shut down slave is no longer available to take over the role of the master. The client reaction is not completely the same. What is the same though, is that the client searches another slave. The difference is that the client always forwards a request again when it was waiting a reply during the shutdown. The client even does so when the request is an at-most-once request. It can do this, as it knows the slave must have been in the Slave state when it sent the Slave Shutdown message. This means the slave was not handling any request. In addition, the slave cannot have already finished handling the request, as the slave must then have send a reply before it sends the Slave Shutdown message, and as the slave uses a single FIFO network connection to send both messages.

Let us now look at what happens when the master shuts down. In this case, the master sends a Master Shutdown message to every slave, as this is again probably faster than using the crash detection mechanism. We require the slaves to detect the message in both their Slave states and their Wait Reply states. When detecting the message, the slaves must react by running the procedure for selecting a new master. Additionally, if a slave is waiting for a reply to arrive, it must send a Master Shutdown message to the client that forwarded the associated request. The client can then forward the request again. It can always do this, independent of the type of request, as the master did not handle the request before its shutdown. If the master did handle the request, the slave receives a reply before it receives the Master Shutdown message, again due to a single FIFO network connection. Of course, when the client forwards the request again it is possible the slave it forwards the request to, becomes the new master. This poses no problem, as the client can detect the Slave Promotion message when the promoting slave sends this message as response to an incoming request.

Evidently, it is possible a process crashes during its shutdown procedure. When this happens in case of the master or in case of a slave, the process possibly did not send all shutdown messages. Fortunately, this is not a real problem, as in almost every case a process reacts the same to a shutdown message as it does when it detects a crash. The only exception is the reaction of a client that waits for a reply to an at-most-once read/write request. If the client receives a shutdown message, it forwards the request again, which is something the client does not do in case it detects a crash. We did not try to overcome this difference, as a process can also crash during shutdown before it sends any shutdown messages. In that case, no process knows the crash happened during a shutdown. This makes it very difficult or even impossible to detect the shutdown, and thus to detect the difference between a crash and a shutdown.

We now have discussed almost everything that can happen in the case of a shutdown. There are only two exceptional cases left. The first of these cases occurs when the master shuts down during the activation of a slave. As described above, a slave sends a New Slave message during activation. If the master now shuts down before it receives the New Slave message, it does not handle the message. The master also does not send a Master Shutdown message to the new slave, as the slave is not yet on its slave list. Thus, in theory, the new slave waits forever. However, in practice it does not. This is because the new slave detects master crashes and because the master stops producing output after shutdown.

The second exceptional shutdown case occurs when the master crashes during a slave shutdown. In this case, the master does not send or only partially sends Slave Removal messages to other slaves. Fortunately, this is again no problem. The slave that shuts down also stops producing output. Therefore, a new master can detect the shutdown with its crash detection mechanism when it did not receive the Slave Removal message.

2.4 Protocol Optimisations

Having discussed the fault tolerant protocol, we now discuss three adaptations that improve this efficiency. The first adaptation is multithreading within processes. Sending only differences as updates is the second adaptation. The third adaptation is not sending the replicated data to the new master after a master crash. We did not include these adaptations in the protocol of the previous section, as they would have distracted from the essence of the protocol.

Multithreading

The first adaptation that improves efficiency is the use of multithreading within processes. We did not use multithreading up until now, as we restricted each process to doing only one thing at a time. This, unfortunately, is somewhat limited. Most of the time, a process can do multiple things without breaking the protocol. Take, for example, the read requests executed by a slave. Read requests do not change the data. Consequently, a slave can easily execute multiple read requests at the same time. As

1. Send the version number to the new master (all S)
2. Compute the most recent version number (NM)
3. Ask a slave with the most recent version number for its data (NM)
4. Wait for the data to arrive (NM)
5. Send the data to every slave that sent an old version number (NM)

Figure 2.7: Protocol for sending only the most recent version of the replicated data

another example, consider a slave waiting for a reply to a read/write request of a particular user. This slave can easily handle some read requests from other users while waiting.

What the previous examples have in common is that we can implement both of them by using of threads. As each example allows multiple events to be handled at the same time, adding threads can thus improve efficiency.

Differences as Updates

As the second adaptation, we consider the updates sent by the master. Up until now, we did not give an exact description of what an update is. Nevertheless, thinking of an update as a transfer of all replicated data is easiest. Unfortunately, the changes made to the data may be small. This makes it inefficient to send all data. When the changes are only small compared to the size of the replicated data, it is better to let the master send differences. In other words, the master must send only the replicated data that changed.

Not Sending the Replicated Data

We now come to the third adaptation that can improve efficiency. This adaptation prevents each slave from sending its replicated data to the new master after a master crash. Of course, it is not entirely possible to prevent this. We still want the new master and every slave to obtain the most recent version of the replicated data. However, letting each slave send its data to the new master is pointless. The new master requires only one copy of the most recent version of the replicated data. The new master does not require any old versions or more than one most recent version. To achieve sending only a single copy we can use version numbering. If we include a version number in the replicated data, and increment the number after every executed read/write request, we can use the protocol in Figure 2.7 to limit the sending of replicated data. The figure uses the letter combinations from Table 2.2 to designate the processes that execute a certain step.

Obviously, when the new master tries to receive the replicated data from one of the slaves, it must be able to detect a crash of that slave. By doing this, the new master does not wait forever when a crash occurs. After a slave crash, the new master can try to get the data from another slave.

An extra advantage of the protocol in Figure 2.7 is that it does not send the most recent version of the data to the slaves that already have it. This too improves efficiency.

Chapter 3

Assessment of the Master-Slave Protocol

In this chapter, we assess the fault tolerant master-slave protocol. We cover four issues. The first issue we cover is the correctness of the protocol. Thereafter, as a second issue, we look at the availability of the protocol. Efficiency of the protocol is the third issue. Finally, the fourth issue looks at the requirement of the network underlying the protocol being a synchronous network.

3.1 Correctness of the Protocol

With respect to the correctness of the fault tolerant protocol, we address two topics. As a first topic, we look at liveness of the protocol. This means we assess if the protocol is free of deadlocks and starvation, even in the face of crash failures. As a second topic, we address the functional correctness of the protocol. This topic requires us to define some sensible semantics to which requests operating the replicated data must adhere, and to verify the requests really adhere to the semantics.

3.1.1 Liveness of the Protocol

As defined above, liveness requires the absence of deadlocks and starvation, even in the face of crash failures. Therefore, to verify the liveness of the fault tolerant protocol we must give a formal proof that shows the protocol does not suffer from any deadlocks or starvation. However, a formal proof confirming the absence of deadlocks and starvation irrespective of the number of slaves, clients, and users is outside the scope of this thesis. Nevertheless, we did not want to ignore the issue of liveness completely. For that reason, we verified the liveness for the situation in which there are three slaves or less. We verified only the non-optimised version of our fault tolerant protocol, as we did not fully elaborate the optimisations of Section 2.4.

A protocol verification tool called Spin helped us to verify the liveness our protocol.[6] We used Spin version 3.4.1. Verification in Spin starts with building a finite automaton for every process. Thereafter, Spin computes the Cartesian product of all automatons and runs a liveness check on each reachable state in the product. Spin finds states by using a depth first search which starts at a designated start state. To build the finite automatons Spin requires definitions of processes expressed in a programming language called Promela, which is a simple imperative language.

The algorithms in tools like Spin function by recording every reachable state. As processes can have large numbers of states, the tools may need quite a lot of memory. In our case, Spin initially needed more memory than available. To overcome this, we decided to combine every slave with the users and clients that forward requests to it.

Combining slaves with clients and users limits the number of states, and thus the amount of memory needed. The limiting effect comes from the fact that we decided to include only those actions involving the master process, and not the actions involving only slaves, clients, and users. This means, we did not need to implement large parts of the slaves, clients, and users.

A disadvantage of combining slaves with clients and users is that the clients and users crash when the slave crashes or when the slave promotes to master. The only way to solve this problem is to separate the clients and users form the slaves. Unfortunately, this gives more states.

By combining the slaves with clients and users, it became possible to check the liveness of the protocol. Spin did not run out of memory, although it still needed about 500 megabytes. The liveness check did not reveal any deadlocks or starvation. We can therefore conclude that our protocol satisfies the liveness property when there are three slaves or less and when the slaves are combined with clients and users.

There is, however, one problem with the above conclusion, as Promela requires us to program all possible crashes ourselves. It is impossible to tell if we really did program all crashes. If we did not program all of them, we have effectively undermined what we know about the liveness of the protocol.

To conclude this section we must note that Appendix A gives the Promela version of our protocol we used to check liveness. In addition, we must note that the limited memory of the machines we had available made it impossible to verify liveness for the case of four slaves.

3.1.2 Functional Correctness of the Protocol

As told above, assessing functional correctness requires the definition of semantics to which requests operating on the replicated data must adhere. We decided each request must adhere to two conditions. First, when a user sends a request, it must take only a finite amount of time before the user receives a reply. Second, when a user sends a read/write request and thereafter sends another request, the other request must operate on a version of the replicated data that includes the changes made by the read/write request. We call this second condition the *See-Your-Writes* condition.

Finite Amount of Time

Let us first look at the finite amount of time condition. What we can see is that when no crashes occur, and when the actual execution of a request takes only a finite amount of time, a user must receive a reply within a finite amount of time. The reason for this is simple. As the network is synchronous, it delivers all messages within a fixed amount of time. Consequently, the network cannot cause a user to wait forever for a reply. In addition, it is reasonable to assume all processes handle the messages they receive in the order in which they receive them. Therefore, no message a process receives has to wait forever until the process handles it. Thus, as the execution of a request also takes a finite amount of time, a user does not have to wait forever.

Turning to the case in which a crash does occur, we can see that when the master or slave crashes, a user still receives a reply within a finite amount of time. The reason for this lies within the clients as they forward read requests and at-least-once read/write requests once more when a crash occurs, and as they return failures as replies in case of at-most-once read/write requests. In case a client crashes, the user also does not have to wait forever. The user must be able to detect a client crash and detection of a crash functions as reply.

In case of a shutdown, reasoning similar to crashes applies. As a result, a user also receives a reply within in a finite amount of time when a shutdown occurs.

Up-To-Date Replicated Data

When we look at the See-Your-Writes condition, we can see that this condition is the same as one of the requirements from section 2.2.1, which poses limitations on the updates the master sends. As shown in Section 2.2.2, we can satisfy the requirement by letting a user always use the same client, by letting a client always use the same slave, and by allowing only a single connection between each pair of processes. The result of this is that as long as no crash occurs the fault tolerant protocol satisfies the See-Your-Writes condition.

The question is whether the protocol also satisfies the See-Your-Writes condition in case of a crash. Unfortunately, this not turns out to be always the case. A problem occurs when a slave returns a reply to a read/write request, and when it crashes. If the client receiving the reply forwards another request for the user that sent the read/write request, the client must forward the request to another slave. However, the protocol does not make certain the other slave has received and applied all necessary updates, so it may not be possible to execute the request with an appropriate version of the replicated data. To make things even worse, when the master also crashes the necessary updates may have been lost completely. No other slave except the crashed one may have received the updates.

The only way to solve the above problems is to let the slaves acknowledge the reception of an update before the master sends associated the reply. Unfortunately, this is inefficient.

3.2 Availability of the Protocol

In this section, we look at the availability of the fault tolerant protocol. Looking at availability means, we assess how many slaves and clients we at least need to still be able to handle requests.

If we assume the liveness property of Section 3.1.1 holds for more than three slaves, and for clients and users not combined with slaves, then we can easily derive the availability of the protocol. As we can interpret liveness as the ability to handle requests as long as no crash occurs, it is obvious the protocol can handle requests as long as there is still one slave and one client. Of course, there must also be a master.

If the liveness property does not hold, availability of the protocol is unclear. It then very much depends on when deadlocks and starvation occur.

3.3 Efficiency of the Protocol

Efficiency is always a relative notion. Therefore, we assess the efficiency of our fault tolerant protocol by comparing it to the non-fault tolerant protocol of Section 2.2. As the non-fault tolerant protocol does not function when crashes occur, we compare only the operation in a crash free environment.

When comparing the protocols, we see that almost all operations they have in common are identical. Consequently, the efficiency of both protocols must also be almost identical. The only difference between the protocols is the handling of new slaves. This operation is more complicated in the case of the fault tolerant protocol. However, the activation of new slaves is not likely to happen very often, as activations are not related to the operations on the replicated data. Therefore, it should not really influence the efficiency.

As a final remark, we should note that the use of acknowledgements, as proposed in Section 3.1.2, probably does influence efficiency. Acknowledgements complicate the way in which the protocol handles read/write requests. Therefore, handling read/write requests becomes less efficient, and, unlike the handling of new slaves, read/write request do operate on the replicated data, and are thus likely to occur relatively often compared to slave activations.

3.4 Using the Protocol in Timed Asynchronous Networks

Up until now, we assumed the use of networks that deliver all messages within a fixed amount of time. Unfortunately, these so-called synchronous networks are rare. Most networks are timed asynchronous.[7] This means they deliver most messages within a fixed amount of time, but not necessarily all of them.

If we want our fault tolerant master-slave protocol to function in a timed asynchronous network, we need to adapt the protocol. We cannot just use the described protocol, as it assumes a crash when a message does not arrive. When we use a timed asynchronous network a crash did not necessarily occur. The network could just be stalling the message.

Although it is probably possible to adapt the master-slave protocol for timed asynchronous networks, we did not try to do this. Adapting a protocol is likely to be complicated. Take, for example, a membership protocol by Cristian.[8] When Cristian and Schmuck adapted the protocol for a timed asynchronous network it became much more complex.[7]

Of course, we do not want to completely ignore the subject of timed asynchronous networks. For that reason, we next present a small analysis of the problems involved. In addition, we present a solution that hardly requires any adaptation of our fault tolerant protocol. However, the solution puts some serious demands on the underlying distributed system.

Analysis of the Problem

As just said, a message not arriving in time does not necessarily mean a process has crashed. This observation has different consequences for the different types of processes.

If we look at clients, we see there are no real consequences. When a client notices a message from a slave is missing, it just tries to find another slave. After the client finds a slave, it can continue normal operation. The switching of the client to another slave does not affect the master and the slaves. They do not know the client exists. Of course, the switching can affect a user, as the client may have been handling a request when it noticed the missing message. Fortunately, forwarding a request once more, like in the case of a crash, also works when no crash occurs. Besides slaves, a client also exchanges messages with users. As a client does not care if a user is present, it also does not care if it misses a message from a user.

In case of a user missing a message from a client, something similar holds as in the case of a client missing a message from a slave. The reason for this is that the protocol requires a user to function in about the same way as a client does in the face of a crash.

Let us now turn to the master. Note that the master exchanges messages only with slaves. As a result, the master can interpret a missing message only as a slave crash. Again, this is not a real problem. The only thing that happens is that the master removes the slave from its slave list, and that it sends a Slave Removal message to all other slaves. This makes the slave from which the message is missing no longer part of the set of slaves. However, the slave can easily overcome this by running the slave activation protocol. Of course, the slave must know when it is no longer part of the set of slaves. We can accomplish this by letting the master send a removal notification message to the slave. Obviously, the notification message can also get lost. This brings us to the consequences of missing messages for slaves.

In contrast to a missing message noticed by a master or client, a missing message noticed by a slave can have serious consequences. A so-called split-brain can occur, which means that multiple masters come into existence at the same time.[3] This is an undesirable situation, as the master is responsible for ordering updates. Having more than one master can make a mess of the ordering.

For an example of how a split-brain can occur, let us look at an error that temporarily partitions the network. The error can separate a number of slaves from some other slaves and the master. As a result, the separated slaves think the other slaves and the master crashed. To overcome this, the slaves run the protocol for selecting a new master. This causes a new master to come into existence within the partition that contains the separated slaves. In addition to this, the master already present before the partition error continues to run. The partition error only causes the master to think the separated slaves have crashed. Consequently, we have two masters.

A split-brain problem can also have other causes. Although we do not present any other causes here, it should be clear missing messages in the communication between the master and the slaves are at the heart of the problem.

Besides exchanging messages with the master, slaves also exchange messages with clients. This exchange cannot cause any problems due to the pattern that the clients and slaves use to communicate. In the pattern, a client first sends a request to a slave. Following this, the slave receives the request, and handles it. Finally, the slave returns a reply to the client. As the clients send the requests at arbitrary moments, slaves do not know when they arrive. Consequently, slaves cannot know when a message from a client is missing.

Solution

A solution to the split-brain presented above is the use of a membership service. When the underlying distributed system offers such a service our fault tolerant protocol can use it to register the master and slaves. Obviously, for an offered membership service to be useful it must able to function correctly in a timed asynchronous network.

At least two systems exist that offer a membership service. These are Isis[9] and Transis.[10] Isis offers a membership service by means of virtual synchrony. Virtual synchrony precludes split-brains by allowing only the processes in one partition of a partitioned network to continue. Transis offers a membership service by means of extended virtual synchrony. Extended virtual synchrony does not preclude split-brains but it merges the masters after the network is no longer partitioned. This possibly requires some support from the protocol when a conflict occurs between the masters.

Chapter 4

Fitting the Fault Tolerant Protocol into Globe

Globe is a wide-area distributed system that uses replication to achieve scalability.[4] Besides other replication strategies, Globe uses master-slave replication. Unfortunately, the protocol Globe uses to implement master-slave replication is not fault tolerant. Although Globe could compensate for this lack of fault tolerance with other facilities, it does not. To be precise, Globe is not fault tolerant at all.

A first step to making Globe fault tolerant can be the introduction of our fault tolerant master-slave protocol. For this, we need to know precisely how to fit the protocol into Globe. In addition, as the fault tolerant protocol is different from non-fault tolerant ones, fitting the protocol can introduce scalability problems. Therefore, we also need to identify potential scalability problems and propose solutions to them.

Both the fitting of our fault tolerant protocol and the identification of the scalability problems are the subject of the current chapter. However, before we can start to describe both subjects it is necessary to know how Globe works. Therefore, we first introduce Globe. Thereafter, in Section 4.2 we fit our protocol. Finally, in Section 4.3, 4.4, and 4.5 we identify potential scalability problems and propose solutions to them.

4.1 An Introduction to Globe

Like many distributed systems, the basic idea behind Globe is sharing data between processes.[4] Globe achieves sharing by using distributed shared objects. These distributed objects are special objects that are accessible throughout the system. To make sharing scalable, each distributed object includes its own replication protocol. There is no universal replication protocol as the optimal protocol depends on the usage of the data and this usage may differ from object to object.

Each distributed object consists of a number of local objects or representatives, which may be located in different address spaces. Each local object of a distributed object may contain a replica of the data in the distributed object. However, this is not strictly necessary. A local object may also function as a proxy that gives access to local objects containing the replicated data.

Figure 4.1 shows the relation between a distributed object and its local objects. In addition to the relation, the figure also shows how each local object is itself composed of four different subobjects. Each of the subobjects fulfils a special function, as described next.

The semantics subobject contains a replica of the shared data. Accordingly, when a specific local object functions only as a proxy it does not include a semantics subobject. In addition to the data, the

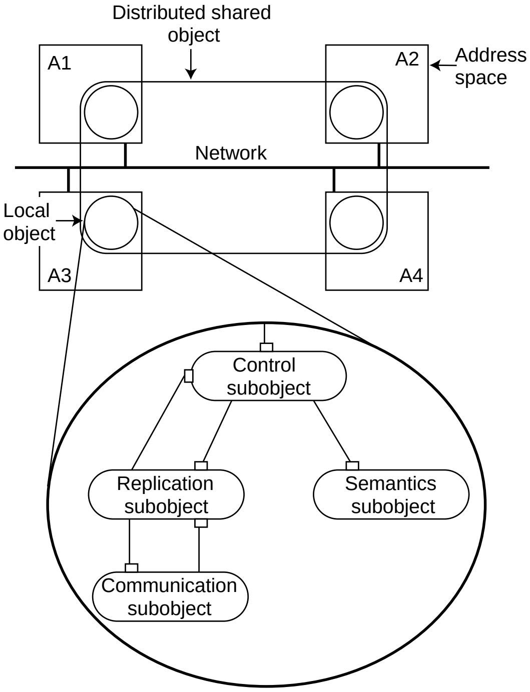


Figure 4.1: The relation between a distributed object, its local objects, and its subobjects

semantics subobject also contains the functionality of the methods that operate on the data. However, the functionality the subobject includes ignores the fact that other replicas may be present in other address spaces.

In the replication subobject, we find part of the replication protocol used by the distributed object that indirectly comprises the subobject. To be precise, we find the part the local object including the subobject needs to fulfil its role in the replication strategy.

The communication subobject offers an interface to the underlying network. As shown in Figure 4.1 the communication subobject interacts with the replication subobject. They interact, as it is likely that the replication subobject can function only when it is able to communicate with its counterparts in other local objects of the same distributed object.

Turning to the control subobject, we see that it interacts with both the semantics subobject and the replication subobject. In addition, the control subobject also provides the interface a user needs when it wants to use the shared data. Between the replication subobject and semantics subobject the control subobject functions as an intermediate. The two subobjects need an intermediate, as Globe demands the independence of the replication subobject and the semantics subobject, so the replication subobject can be used in different distributed objects.

We should note that a user is normally called a client process in Globe. However, for consistency with Chapter 2 and 3, and to avoid confusion with the client in the master-slave replication strategy, we continue to use the term user.

We now know what a distributed object looks like. What we do not know, however, is how a user binds to a distributed object so it can use the shared data.

In Globe, binding to a distributed object involves four steps. In the first step, the user that wants to bind sends the name of the distributed object to a naming service. This naming service resolves the name to an object handle, which is an identifier unique to every distributed object. After the naming service returns the object handle, the user continues with the second step of the binding process by sending the object handle to the Globe location service. The location service resolves the object handle to one or more contact addresses, which it returns to the user. Each contact address contains information on the address and the protocol needed to contact one or more local objects of a distributed object. After receiving the contact addresses, the user performs the third step of the binding process. In this step, the user loads an implementation of a local object from a repository. Finally, the fourth step initialises the implementation on the machine of the user. During the initialisation, the user hands the information from one of the received contact addresses to the implementation.

The result of the binding process is thus that the user has a local object on its machine. Following the binding process, the user can perform method calls on its local object to access the shared data.

To further support distributed shared objects, Globe uses a special type of server, called an object server. Object servers are specially designed to hold large numbers of local objects.

4.2 Fitting the Protocol

In this section, we show a possible way to fit our fault tolerant master-slave protocol into Globe. We cover three issues. The first issue explains how we can fit the actual protocol. The second issue shows how we can fit the crash detection mechanism the protocol needs. Finally, the third issue makes clear how the protocol can use the Globe location service.

Fitting the Protocol

When we want to fit our fault tolerant protocol into Globe, there is little room for variation. Globe only allows fitting replication protocols within distributed objects. In addition, in the distributed object we may fit the protocol only in the replication subobjects of the local objects. As a result fitting our protocol gives three different replication subobjects: one for the local object that becomes the master, one for the local objects that become slaves, and one for the local objects that become clients.

Fitting the Crash Detection Mechanism

Obviously, the crash detection mechanism and our replication protocol have a strong relation. For that reason, it is best to place the crash detection mechanism in the local objects that use the replication protocol. Within the local objects, we think it is best to fit the mechanism into a new subobject. This new subobject must be able to interact with both the replication subobject and the communication subobject. Interaction with the replication subobject is required as our replication protocol needs to know about crashes and as the crash detection mechanism needs to know which local objects it needs to monitor. With respect to the communication subobject, interaction is required as crash detection is possible only by using the network.

To show why we think it is best to introduce a new subobject, let us look at the four subobjects already present in a local object. In the case of the semantics subobject, it is obvious the crash detection mechanism should not be part of it. Crash detection has nothing to do with the shared data and the operations on the data. In the case of the control subobject, something similar holds. Crash detection has no relation to the interaction between the user, the replication subobject, and the semantics subobject. When we look at the communication subobject, we can see that this is also not the right subobject for placing the crash detection mechanism. The reason is that the communication subobject is present only to provide an interface to the underlying network functionalities, and crash detection is mostly not one of the network functionalities. If we now turn to the replication subobject, we can say that it is certainly possible to add the crash detection mechanism to the subobject, due to the strong relation between crash detection and fault tolerant replication. However, the replication protocol does not care how crash detection exactly takes place. Therefore, to make the crash detection mechanisms exchangeable, it is best also not to place it in the replication subobject. Consequently, as we have eliminated all subobjects already present, it is necessary to introduce a new subobject for the crash detection mechanism.

The Globe Location Service

As discussed in Chapter 2, the processes in the fault tolerant protocol sometimes require the address of the master or a slave to function correctly. To achieve this, the chapter suggested the use of a directory service to store the addresses. When we now look at the Globe location service, we can see that it stores contact addresses. Consequently, when we fit our fault tolerant master-slave protocol in Globe, we can use the location service to store the addresses of local objects that function as master or slave.

When we combine the Globe location service with the suggested use of the directory service, we get a number of cases in which the fault tolerant protocol uses the location service. If we look at clients, we see they use the location service when they activate, and when the slave they use crashes or shuts down. Turning to slaves, we can see that they use the location service to contact the master during activation. In addition, slaves also need to register at the location service to make themselves reachable for clients. In the case of the master, we see that it uses to location service to make its address known, to remove the addresses of crashed or shut down slaves, and to remove the address of the previous master when the master is a new master.

Note we explained above that the clients and slaves can use the location service when they activate. As activation occurs only during binding, the address given to the clients and slaves during binding is redundant when using the location service in the explained way.

For a more precise description of the use of the location service, we like to refer to Appendix B. This appendix gives pseudo-code implementations for the three different replication subobjects needed for our fault tolerant protocol. The implementations use the location service.

4.3 Problems with Crash Detection

In this section, we start to identify and propose solutions to potential scalability problems related to Globe and our fault tolerant protocol. In what follows next, we first look at scalability problems related to crash detection. Thereafter, Section 4.4 and 4.5 look respectively at problems related to the location service and object servers.

Up until now, we have proposed only one mechanism for crash detection. The mechanism required processes, or local objects in the case of Globe, to regularly send I-am-alive messages. As it is outside the scope of this thesis to investigate other crash detection mechanisms, we from now on assume all fault tolerant local objects use the proposed mechanism.

It should be obvious that the use of I-am-alive messages is limited only by the capacity of the underlying network. Therefore, to make the use of I-am-alive messages scale we need to limit the number of messages sent. To do this, there are two possible solutions, one per-local object solution, and one per-machine solution. The per-object solution is to piggyback the I-am-alive messages on other messages with the same destination as sent by a local object. The per-machine solution is to use message aggregation. This means the machine combines messages with the same destination.

Of course, the last solution is useful only when the messages of different local objects on the same machine have the same destination.

4.4 Problems with the Location Service

In this section, we identify and propose solutions to scalability problems at the location service, as caused by the use of the service proposed in Section 4.2. With respect to the location service, we only identify scalability problems that occur after crashes. We ignore problems that occur in crash free situations. The reason for doing this is that our fault tolerant protocol functions the almost same as a non-fault tolerant one when no crash occurs. Consequently, when no crash occurs using the fault tolerant cannot introduce new scalability problems.

As a further limitation, we only identify problems caused by object server crashes. We ignore problems caused by machines not running object servers. It is reasonable to do this, as we may assume the master and slaves of all distributed object are located on object servers, due to the lack of direct dependence on users. The other machines present can thus contain only clients. When these clients crash, they do not cause any local objects to use location service.

With respect to the object server crashes, we look only at complete object server crashes. Partial crashes generally cause the same problems, only on a less severe scale.

The identification and solving of scalability problems now proceeds in two parts. In the first part, we look at problems caused by the clients of different distributed objects. Thereafter, in the second part, we look at problems caused by the masters and slaves of different objects.

Problems Caused by Clients

When an object server crashes, and a client uses a slave located on that server, then the client has to use the location service to find another slave. This use of the location service can cause scalability problems. To identify the scalability problems we use the tree in Figure 4.2, and we look at every path from the root of the tree to a leaf. In the tree, each internal node represents a variable. By selecting one of the edges directly below an internal node, we give the variable a value. A leaf node tells us if a scalability problem arises when we assign to the variables on the path from the root to the leaf the values of the edges on that path.

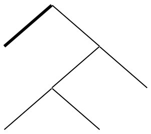
The numbers in the tree stand for the following variables:

1. Distribution of the detection of an object server crash by clients

2. Number of slaves on an object server in use by clients

3. Number of clients using a certain slave on an object server

The first variable defines the time in which all clients using a specific slave can detect an object server crash as a crash of the slave they use. With respect to this variable, there are two extreme values. Every client detects the object server crash at roughly the same time, or only a few clients detect do this at the same time. The second extreme effectively spreads the detection over a long period. In the tree, we use only the extreme values. We also do this in the case of the other variables, which should have a clear meaning. The extreme values of these other variables are many and few.



We now start the tour of our tree by looking at the path as depicted in the margin next to this paragraph. When we refer back to Figure 4.2, we see that the path spreads the crash detection by clients over a long period. As we also see, the spreading makes sure no scalability problem arises. The reason no problem arises lies in the fact that the clients use the location service directly after they

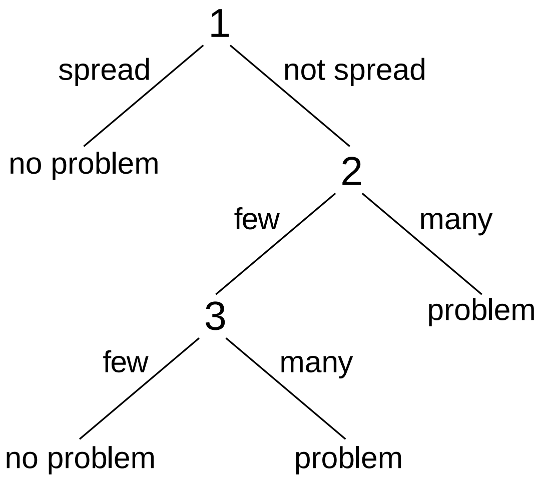
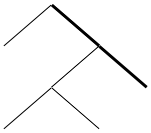
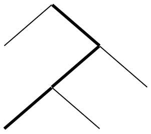


Figure 4.2: Tree for identifying problems with the location service caused by clients

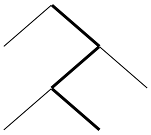
detect the crash. As spreading of the crash detection makes sure only a few clients detect the crash at the same time, there are also only a few clients use the location service at the same time.



If we now look at the path depicted next to this paragraph, we see that it leads to a scalability problem. The values of the variables on the path make sure every client detects an object server crash a roughly the same time. In addition, they also make certain many slaves on a crashed object server are in use by clients. As each client uses only a single slave at the time, having many slaves in use means there must be many clients using the object server. As these clients all detect the crash at roughly the same time, the location service receives many requests at same time. The service probably receives more requests than it can handle. We present possible solutions to this problem after we complete the tour of our tree.



Looking at the longest possible path that ends on the left side of Figure 4.2, we see that it does not give rise to any scalability problems. The reason for this is that the path assumes only a few slaves on a crashing object server are in use by clients, and that only a few clients use each slave. The result of these assumptions is that in total only a few clients have to go to the location service after an object server crash, which does not cause any problems. That all clients detect the crash at roughly the same time does not influence this.



If we now turn to the last possible path, we see that it differs from the previous one only because it assumes many clients use each slave. Although this is a small difference, it causes a scalability problem. It does this as we now have many clients that all detect an object server crash at roughly the same time. The result is that many clients send a request to the location service at the same time, and the service can probably not handle that many requests.

Before we continue with solutions to the encountered problems, we note, that although we concentrated on the clients that have to find a new slave because their slave crashed, there is also another category of clients that needs to find a new slave. This is the category of clients that use a slave which promotes to master after the previous master stopped operating due to an object server crash. Obviously, this category of clients can cause the same problems.

If we now look at the encountered scalability problems, we see that their cause lies within the fact that many clients want to use the location service at the same time. Therefore, to solve the scalability problems we need to spread the use of the location service. To do this there are at least two solutions.

The first solution is to just let every client wait a random time before using the location service. Unfortunately, this makes clients unusable during the time they wait. The second solution is not to use the location service. To accomplish this, a client must store a number of slave addresses locally, and choose one of these addresses after a crash. To acquire the addresses, a client could request more than one address from the location service during its activation, or it could request the addresses from the slave it uses. Acquiring the addresses from the slave is possible because every slave holds a list with all active slaves.

A problem with storing addresses locally is that the addresses can become stale. This happens when the slaves behind the addresses crash or shutdown. One possible solution to this problem is to do nothing. When using this solution, it may happen that all addresses a client stores are stale. In that case, the client needs to return to the location service. Unfortunately, as a number of clients may need to do this, the scalability problems related to the location service can surface again. However, clients can only find out that their addresses are stale by trying all stored addresses. As the time needed to do this probably varies from client to client, it is unlikely all clients return to the location service at the same time. This makes the re-emergence of a scalability problem with the location service highly unlikely.

Another solution to the problem of invalid addresses is to try to keep the information the clients store on addresses in agreement with the situation in the system. There are at least two possible ways to achieve this. First, the location service or a slave could multicast changes to clients. Second, the clients could regularly ask for changes at the location service or a slave. Of course, both solutions possibly introduce new scalability problems.

Problems Caused by Masters and Slaves

In theory, the masters and slaves of distributed objects with a local object on a crashed object server can cause scalability problems. However, we can avoid all problems by not letting the masters and slaves use the location service immediately in all cases. It is possible to do this, as the masters and slaves do not need the location service to function after a crash. They only use the location service to help clients that need to find another slave and to help clients and slaves that activate after an object server crash.

When we look at clients, we can see that it is not a real problem when the location service holds some invalid addresses. When a client retrieves an invalid address, it can simply request another address after it notices the invalidity. Of course, the location service must contain at least one valid slave address for the distributed object of which the client wants to become part.

In case a new slave wants to become part of a distributed object, the address of the current master of that object must be present in the location service. Therefore, when a slave promotes to master due to an object server crash, the address must be registered at the location service as soon as possible. However, doing this can cause scalability problems at the location service when the slaves of many different objects promote. To solve this, we could postpone the registration, and we could let a new slave retrieve the address of a slave and ask that slave for the address of the current master. Again, this requires at least one valid slave address to be present for the correct distributed object.

Although we can solve the scalability problem with the new master registration, the question is if we need to do so. The reason for this is that it is likely the master of each distributed object is located on an arbitrary object server. This makes the probability small an object server contains many masters. Consequently, when an object server crashes, hardly any slave promotions need to take place, and thus no scalability problems arise at the location service.

4.5 Problems with Object Servers

In this section, we identify and propose solutions to the scalability problems our fault tolerant protocol causes at the object servers that remain active after an object server crash. As in the previous section, we ignore the situation in which no crash occurs and the crashes of machines that do not run object servers. The reasons are again that in a situation in which no crash occurs our fault tolerant protocol does not differ from non-fault tolerant protocols, and that the master and slaves of all distributed objects are likely to be located on object servers.

To limit the discussion below to reasonable proportions, we assume that all objects servers have identical capacities and capabilities. In addition, we also assume that before an object server crash takes place all object servers have identical loads. The advantage of making these assumptions is that it avoids the problem of looking at different combinations of object server configurations. There can be many of these configurations.

In the following, we again split the identification of scalability problems into two parts. In the first part, we identify scalability problems caused by clients. Thereafter, the second part identifies scalability problems caused by masters and slaves.

Problems Caused by Clients

As in the previous section, we use a tree to guide our identification of scalability problems. The tree, which is shown in Figure 4.3, uses almost the same conventions as the tree in the previous section. The only difference is that the current tree contains an arrow. We explain this arrow when we encounter it in the tour of the tree.

In the tree, each number again corresponds to a variable of which we use the extreme values. The correspondence is as follows:

1. Relation of the request frequency and the capacity of an object server
2. Number of clients responsible for a request frequency
3. Number of slaves per distributed shared object
4. Number of slaves on an object server in use by clients
5. Degree of co-location of the local objects of different distributed objects

The first variable gives an indication of the relation between the number of requests an object server needs to handle and the maximum number of requests it can handle. The variable has two extreme values: the number of requests the server needs to handle is much lower than the maximum number, or the number of requests is almost as high as the maximum number. The second variable tells us how many clients are responsible for a certain request frequency. Its extreme values are many clients and few clients. When we look at the third and fourth variable, their meanings should be clear. The extreme values are in both cases many and few. The degree of co-location in the fifth variable tells us whether local objects of different distributed object are located on the same object servers or not. There can be a high degree of co-location or a low degree.

During the first part of the tour of our tree, we assume that clients using slaves on a specific crashed object server switch to a very limited number of other object servers. This corresponds with clients deterministically choosing another slave, and with a high degree of co-location of the distributed objects that lose a slave due to the object server crash.

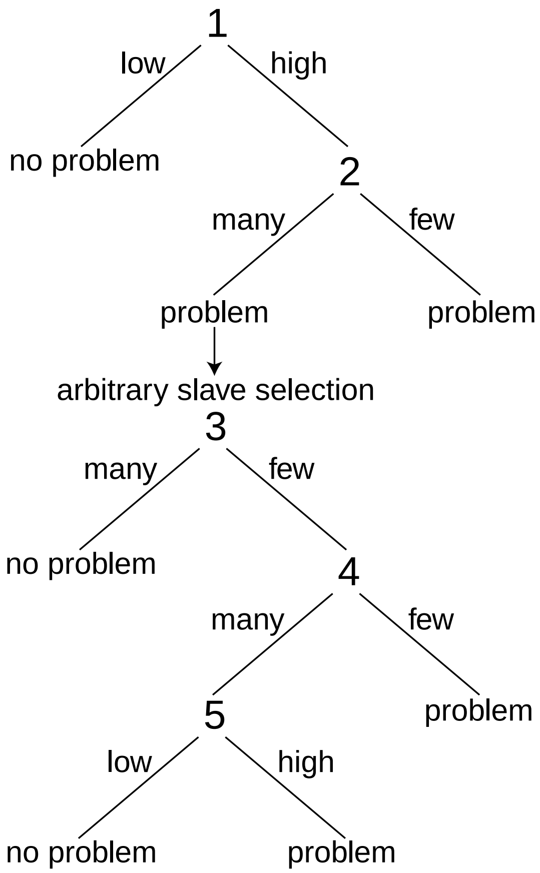
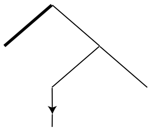
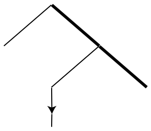


Figure 4.3: Tree for identifying problems with object servers caused by clients

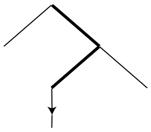


Starting with the shortest possible path from the root to a leaf of the tree in Figure 4.3, we see that it requires the request frequency of every object server to be limited. We also see that the path does not cause a scalability problem. The reason is that the object servers that remain after an object server crash have enough capacity to handle the switching clients because the request frequency is low at all object servers.



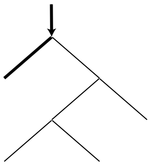
If we now look at the path depicted next to this paragraph, we see that it identifies a scalability problem. As the path assumes a few clients cause a high request frequency, it is obvious that at least one client must also have a high request frequency. After an object server crash, the client with the high frequency switches to another object server, which also has a high request frequency. Due to the high request frequencies, the additional load of the switching client may be too much for the object server.

A solution to the scalability problem identified above is to use something that looks like the slow start mechanism of TCP.[11] What this means is that when a client switches to another object server, it must start with sending only a limited number of requests to the object server. This gives the server time to identify a potential load problem. Following the identification the object server can solve the load problem by notifying the slave to which the switched client sends its requests. That slave can then make sure a new slave starts on a not so heavily loaded server. Thereafter, the slave can notify the switched client and ask the client to switch to the new slave to prevent overloading. Of course, this whole solution requires a distributed object to be able to dynamically start new slaves.

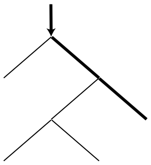


Continuing the tour of the tree with the path that ends at the same level as the previous one, we see that it also causes a scalability problem. The difference with the previous path is that many clients together are now responsible for the assumed high request frequency. As we assumed clients switch to a limited number of other object servers, the effect is the same as is in case when a single client with a high request frequency switches.

To solve the scalability problem we cannot apply a slow start, as each individual client has only a low request frequency. What we can do, however, is to let clients switch to arbitrary object servers which hold slaves from the same distributed objects of which the clients are a part. Unfortunately, it is by no means clear that using arbitrary slave selection solves all scalability problems. For that reason, we from now on assume arbitrary slave selection, and we continue to identify scalability problems. The arrow in Figure 4.3 depicts the use of arbitrary slave selection.

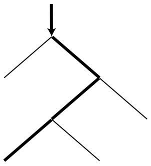


If we now look at the shortest path ending below the arrow in the tree, we see that it does not identify a scalability problem. The reason is the path assumes that the many clients that cause the high request frequency at a crashed object server are part of distributed objects with many slaves. Together with the arbitrary slave selection, this results in the clients switching to many different object servers. Consequently, the request frequency at the object servers can increase only marginally, which should be manageable.

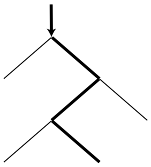


Turning to the path depicted next to this paragraph, we see that it assumes each distributed object has only a limited number of slaves, and that the number of slaves in use on each object server is limited too. Due to the limited numbers, the clients that need to switch after an object server crash switch only to a limited number of object servers. As the total request frequency of the clients is high, each of the object servers to which the clients switch experiences a substantial increase in the number of requests it receives. Consequently, as we assume each object server has a high request frequency, the switching clients are likely to cause scalability problems.

Solving the scalability problems is easy. We just have to make sure there are more slaves, so that when an object server crash occurs the clients switch to a large number of object servers, and so that the clients only marginally increase the number of the requests the servers need to handle.



If we now look at the left path ending at the lowest level of the tree, we see that the combination of assumptions on the path does not give rise to a scalability problem. The reason for this is that each object server holds many slaves that are in use by clients. Thus, each slave on an object server services a limited number of clients. As the degree of co-location is low, the clients that switch due to a crash of an object server, switch to many different object servers, even though each distributed object only has a limited number of slaves. Consequently, the high request frequency the switching clients jointly generate spreads over a large number of object servers, and thus causes the request frequencies of the object servers to increase only marginally.



Turning to the right path that ends at the lowest level, we see that it does cause a scalability problem. This is a consequence of the high degree of co-location, which causes the clients to switch to a limited number of other object servers after an object server crash. The number of requests these object servers need to handle thus increases substantially.

To solve the last scalability problem, we can again increase the number of slaves per distributed shared object. This spreads the clients over a larger number of object servers. Another solution is to decrease the degree of co-location. We can do this by locating the local objects of different distributed objects not on the same object servers but on object servers that lie near each other.

As in the previous section, we must note that the clients that switch directly due to an object server crash are not the only clients that switch. When a crashed object server holds a master, one of the remaining slaves must promote to master. This requires the clients that use the promoting slave to switch. As the cause of all identified scalability problems lies in the switching of clients, clients switching due to slave promotion can cause the same problems as clients switching due to an object server crash. Obviously, this means the same solutions apply.

Problems Caused by Masters and Slaves

Having identified scalability problems caused by clients, we now come to the problems caused by the masters and slaves of distributed objects.

When we look at the masters of the distributed objects that have a slave involved in an object server crash, we see that they have to send messages to all their other slaves telling them to remove the crashed slave from their slave list. If the masters of the distributed objects all send the messages at the same time, a scalability problem can arise. However, sending messages is not essential for the distributed objects to continue to function. The messages only help to limit the search slaves need to perform in the case of a master crash. Consequently, the masters of the distributed objects can wait an arbitrary time before sending the messages. This overcomes the potential scalability problem.

Turning to the slaves, we can see that they need to send messages when their master is involved in an object server crash. As many slaves may need to send messages, a scalability problem can arise. To solve this problem slaves could wait an arbitrary time. However, as long as a distributed object has no master, it is impossible to execute read/write requests. Consequently, if we use the solution and still want a reasonable response time for read/write requests, these requests should not occur too often.

It is, of course, the question if scalability problems really arise when the master crashes. The number of slave promotions is likely to be small. As explained in the previous section, this is because it is likely there are much more slaves than masters on a crashed object server.

Chapter 5

Object Server Recovery

Up until now, we discussed our fault tolerant master-slave protocol. Although the protocol can help to make distributed systems fault tolerant, we need to do more to acquire a system that is completely fault tolerant, like the introduction of recovery. For that reason, this chapter pays some attention to the recovery of local objects after the crash of a Globe object server.

In the discussion below, we first introduce some mechanisms that make the recovery of local objects possible. Thereafter, we present a number of policies for local object recovery. These policies are necessary, as an object server cannot recover all local objects at the same time.

5.1 Recovery Mechanisms

When we recover a crashed object server with its local objects, we at least need to know which local objects were present on the object server. To achieve this, the object server can store information of the present local objects on stable storage. Of course, it may not always be necessary to recover a local object. Take, for example, a distributed object that dynamically starts new local objects after a crash of a local object. In this case, newly started local objects make up for the crashed one, and recovery of the crashed local object becomes needless.

For an object server to be able to differentiate between the local objects that want recovery and those that do not, it is necessary for each local object to make its wishes known. We can achieve this by associating a flag with every distributed object which provides the necessary information. The object server then needs to store information only on those local objects with the flag set in the correct state.

Let us now briefly consider the recovery of a local object that is part of a distributed object using our fault tolerant master-slave protocol. When we look at a master that is present on a crashed object server, we see that a slave takes over its role. Consequently, to prevent a split-brain when recovery takes place, the master must not recover as a master but as a slave. When the recovering master wants to become the master again, we need to adapt the protocol. We must make it possible for the recovering master to degrade the slave that promoted to master.

5.2 Recovery Policies

To decide on the order in which to recover the local objects on a recovering object server several policies are possible. A first policy is to wait with the recovery of a specific local object until a request for it arrives from another local object. The disadvantage of this policy is that it probably does not recover all local objects present before an object server crash. For an explanation, let us look at our fault tolerant master-slave protocol. In the case of the protocol, all clients using a specific slave switch to another slave after they notice the slave crashed. Consequently, the slave does not receive any requests from clients after recovery. In addition, newly activating clients cannot send requests either, as the protocol removes the address of a crashed slave from the location service. The result is that clients cannot help in the recovery of a slave. Furthermore, the master and other slaves cannot help either, as they remove the crashed slave from their slave lists.

A problem with not recovering all local objects occurs when a distributed object cannot dynamically start new local objects. In that case, the number of local objects in a distributed object decreases, and is not replenished. This can eventually lead to the extinction of a distributed object after a number of object server crashes. To overcome this an object server could recover all local objects, even if no requests come in. Of course, as an object server cannot recover all local objects at the same time, it must decide on the recovery order. One possible order is an arbitrary order. Unfortunately, this may not recover the local objects needed most.

To achieve a better recovery order, an object server could save statistics, and use these statistics to first recover those local objects that are likely to receive most requests after recovery. Statistics that can be useful in this context are the frequency of the requests sent to a local object, and the frequency of the binds to the local object.

Let us first look at the frequency of the requests. The idea behind using this statistic is that it is likely that the local objects with the highest request frequency also have a high request frequency after recovery. Of course, when the local objects that send requests switch after a crash, using the request frequency from before the crash is not very useful. To compensate, the object server could try to estimate the request frequency after the crash by using extra statistics like downtime, the number of local objects responsible for a request frequency, and the distribution in time of crash detection by the local objects sending requests.

Let us now turn to the frequency of the binds to a local object. This statistic is mainly useful when a local object sends a request immediately after binding. When an object server also saves information on the average number of requests between every bind and its associated unbind, and when it saves information on the time between every bind and unbind, the binding frequency can also be useful to estimate the request frequency after recovery.

Chapter 6

Conclusions and Future

6.1 Conclusions

In this thesis, we have shown that making replication protocols and wide-area distributed systems fault tolerant are far from easy tasks. However, we have also shown that fault tolerance is at least partially achievable. In particular, it is possible to design a fault tolerant master-slave protocol, and to fit that protocol into the Globe wide-area distributed system. In addition, we have shown it is possible to recover the local objects of crashed object servers.

In the case of the fault tolerant master-slave protocol, we have seen that the protocol is far from trivial. Therefore, it is unclear to us why literature does not give exact explanations of such protocols, as we already noted in Chapter 1.

Turning to problems that appeared when we tried to fit our fault tolerant master-slave protocol into Globe, we can say that we have seen that quite a number of not so obvious problems. However, we can also say that all problems could be solved.

When we finally look at the recovery of object servers, we can see that we have kept the mechanisms to support local object recovery very simple. In addition, the policies to support local object recovery were not very complicated too.

6.2 Future

As a follow-up to this thesis, we could try to construct a complete liveness proof of our fault tolerant master-slave protocol. Further more, we could also try to construct fault tolerant replication protocols that implement other replication strategies. We, for example, could try to construct a fault tolerant active replication protocol.[12] Finally, we could also try to construct a fault tolerant master-slave protocol that can handle failures different from crash failures, such as Byzantine failures.[13]

References

[1] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., AND TOUEG, S. The Primary-Backup Approach. In *Distributed Systems*, S. Mullender, Ed., 2nd ed., ACM Press frontier series. ACM Press, New York, NY, 1993, ch. 8, pp. 199–216.

[2] WANG, L., AND ZHOU, W. Primary-Backup Object Replications in Java. In *Proceedings of the 27th International Conference on Technology of Object Oriented Languages and Systems* (Sept. 1998), IEEE.

[3] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM 34*, 2 (Feb. 1991), 56–78.

[4] VAN STEEN, M., HOMBURG, P., AND TANENBAUM, A. S. Globe: A Wide-Area Distributed System. *IEEE Concurrency* (Jan.-Mar. 1999), 70–78.

[5] GUERRAOUI, R., AND SCHIPER, A. Fault-Tolerance by Replication in Distributed Systems. In *Proceedings of the International Conference on Reliable Software Technologies* (1996), Lecture Notes in Computer Science, Springer-Verlag.

[6] HOLZMANN, G. J. *Design and Validation of Computer Protocols*. Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, NJ, 1991.

[7] CRISTIAN, F., AND SCHMUCK, F. Agreeing on Processor Group Membership in Timed Asynchronous Systems. Tech. Rep. CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.

[8] CRISTIAN, F. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing 4* (1991), 175–187.

[9] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM 36*, 12 (Dec. 1993), 37–53.

[10] DOLEV, D., AND MALKI, D. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM 39*, 4 (Apr. 1996), 64–70.

[11] TANENBAUM, A. S. *Computer Networks*, third ed. Prentice Hall, Englewood Cliffs, NJ, 1996.

[12] SCHNEIDER, F. B. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–320.

[13] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

Appendix A

Master-Slave Protocol – Promela Implementation

This appendix gives a Promela implementation of our fault tolerant master-slave protocol. We used this implementation to check the liveness of the protocol in Section 3.1.1. The implementation should be easy to understand after reading Chapter 2 and the book by Holzmann.[6]

```
/*
 * Promela implementation of the fault tolerant master-slave protocol.
 */

#define MASTER_SLAVE_MAX 5 /* maximum number of representatives */
#define CHANNEL_SIZE 8     /* maximum channel size, should be large enough */
#define UNUSED -1          /* dummy value */

mtype = {new_slave,        /* master to slave */
          new_slave_ack,   /* slave to master */
          rw_request,      /* slave to master */
          rw_reply,        /* master to slave */
          slave_removal,   /* master to slave */
          start,           /* master to new slave */
          state_info,      /* slave to new master */
          state_update,    /* master to slave */
          state_and_list}; /* master to new slave and new master to slaves */

typedef master_slave_data
{
  short master_slave_crashed;
  chan  master_slave_chan = [CHANNEL_SIZE] of {mtype, short};
  byte  is_slave_copy[MASTER_SLAVE_MAX];
}

master_slave_data master_slave[MASTER_SLAVE_MAX];

short number_of_procs = 0;

bool  master_present = false;
short index_of_master;
short master_slave_entries_used;

/* Slave representative: */
```

```
proctype slave(short slave_index)
{
  byte  is_slave[MASTER_SLAVE_MAX];
  chan  state_info_expected[MASTER_SLAVE_MAX] = [1] of {bool};
  chan  slave_chan = master_slave[slave_index].master_slave_chan;
  chan  master_chan;
  mtype message_type = 0;
  short index_1 = 0;
  short i = 0;
  short j = 0;
  short state_infos_expected = 0;
  short master_index;

start_over:
  atomic
  {
    master_present ->
      master_index = index_of_master;
      master_chan  = master_slave[master_index].master_slave_chan;
      master_chan!new_slave(slave_index);
  }
  do
  :: atomic
      {
        slave_chan??state_and_list(UNUSED) ->
        do
        :: i < MASTER_SLAVE_MAX ->
            is_slave[i] = master_slave[slave_index].is_slave_copy[i];
            master_slave[slave_index].is_slave_copy[i] = false;
            i++;
        :: i == MASTER_SLAVE_MAX ->
            i = 0;
            break;
        od;
        break;
      }
  :: atomic
      {
        master_slave[master_index].master_slave_crashed ->
          master_slave[slave_index].master_slave_crashed = true;
          slave_index = master_slave_entries_used;
          slave_chan  = master_slave[slave_index].master_slave_chan;
          master_slave_entries_used++;
          goto start_over;
      }
  od;
  do
  :: atomic
      {
        slave_chan??start(UNUSED) ->
          number_of_procs++;
          break;
      }
  :: atomic
      {
        master_slave[master_index].master_slave_crashed ->
          master_slave[slave_index].master_slave_crashed = true;
          slave_index = master_slave_entries_used;
          slave_chan  = master_slave[slave_index].master_slave_chan;
          master_slave_entries_used++;
          goto start_over;
      }
  :: atomic
      {
        number_of_procs >= 2 ->
          goto crash;
      }
  od;

normal_operation:
progress:
end:
  do
  :: master_chan!rw_request(slave_index);
      do
      :: atomic
          {
            slave_chan??[rw_reply(UNUSED)] ->
              slave_chan??state_update(UNUSED);
              slave_chan??rw_reply(UNUSED);
              break;
          }
      :: atomic
          {
            slave_chan??new_slave(index_1) ->
              is_slave[index_1] = true;
              index_1 = 0;
              if
              :: master_chan!new_slave_ack(slave_index);
              :: number_of_procs > 2 ->
                  number_of_procs--;
                  goto crash;
              fi;
          }
      :: atomic
          {
            master_slave[master_index].master_slave_crashed ->
              goto handle_master_crash;
          }
      :: atomic
          {
            number_of_procs > 2 ->
              number_of_procs--;
              goto crash;
          }
      od;
  :: slave_chan?state_update(UNUSED) ->
      /* do nothing */
  :: atomic
      {
        slave_chan?new_slave(index_1) ->
          is_slave[index_1] = true;
          index_1 = 0;
          if
          :: master_chan!new_slave_ack(slave_index);
          :: number_of_procs > 2 ->
              number_of_procs--;
              goto crash;
          fi;
      }
  :: atomic
      {
        slave_chan?slave_removal(index_1) ->
          is_slave[index_1] = false;
          index_1 = 0;
      }
  :: atomic
      {
        master_slave[master_index].master_slave_crashed ->
          goto handle_master_crash;
      }
  :: atomic
      {
        number_of_procs > 2 ->
          number_of_procs--;
          goto crash;
      }
  od;

handle_master_crash:
  atomic
  {
    if
    :: skip;
    :: number_of_procs > 2 ->
        number_of_procs--;
        goto crash;
    fi;
    /* clean up message from previous master */
    i = len(slave_chan);
    do
    :: i > 0 ->
        slave_chan?message_type(index_1);
        if
        :: message_type == new_slave
              || message_type == rw_reply
              || message_type == slave_removal
              || message_type == state_update
              || message_type == state_and_list ->
            /* do nothing */
        :: else ->
            slave_chan!message_type(index_1);
        fi;
        i--;
    :: i == 0 ->
        message_type = 0;
        index_1 = 0;
        break;
    od;
    /* select new master */
    do
    :: is_slave[i] ->
        master_index = i;
        master_chan = master_slave[master_index].master_slave_chan;
        is_slave[i] = false;
        if
        :: i == slave_index ->
            i = 0;
            goto synchronize_with_slaves;
        :: i != slave_index ->
            i = 0;
            goto synchronize_with_new_master;
        fi;
    :: !is_slave[i] ->
        i++;
    od;
  }

synchronize_with_slaves:
  atomic
  {
    do
    :: i < MASTER_SLAVE_MAX ->
        if
        :: is_slave[i] ->
            state_infos_expected++;
            state_info_expected[i]!true;
        :: !is_slave[i] ->
            /* do nothing */
        fi;
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;
  }

continue_state_info:
  do
  :: atomic
      {
        master_chan??state_info(index_1) ->
          state_info_expected[index_1]?true;
          state_infos_expected--;
          index_1 = 0;
      }
  :: atomic
      {
        state_infos_expected == 0 ->
          goto continue_synchronize;
      }
  :: atomic
      {
        is_slave[1] && master_slave[1].master_slave_crashed ->
          index_1 = 1;
          goto handle_slave_crash;
      }
  :: atomic
```

```
      {
        is_slave[2] && master_slave[2].master_slave_crashed ->
          index_1 = 2;
          goto handle_slave_crash;
      }
  :: atomic
      {
        is_slave[3] && master_slave[3].master_slave_crashed ->
          index_1 = 3;
          goto handle_slave_crash;
      }
  :: atomic
      {
        is_slave[4] && master_slave[4].master_slave_crashed ->
          index_1 = 4;
          goto handle_slave_crash;
      }
  od;

handle_slave_crash:
  atomic
  {
    if
    :: state_info_expected[index_1]?[true] ->
        state_info_expected[index_1]?true;
        state_infos_expected--;
    :: !state_info_expected[index_1]?[true] ->
        /* do nothing */
    fi;
    is_slave[index_1] = false;
    index_1 = 0;
    goto continue_state_info;
  }

continue_synchronize:
  do
  :: atomic
      {
        i < MASTER_SLAVE_MAX ->
          if
          :: is_slave[i] ->
              j = 0;
              do
              :: j < MASTER_SLAVE_MAX ->
                  master_slave[i].is_slave_copy[j] = is_slave[j];
                  j++
              :: j == MASTER_SLAVE_MAX ->
                  break;
              od;
              master_slave[i].master_slave_chan!state_and_list(UNUSED);
          :: !is_slave[i] ->
              /* do nothing */
          fi;
          i++;
      }
  :: atomic
      {
        i == MASTER_SLAVE_MAX ->
          i = 0;
          j = 0;
          break;
      }
  :: atomic
      {
        number_of_procs > 2 ->
          i = 0;
          j = 0;
          number_of_procs--;
          goto crash_master;
      }
  od;
  atomic
  {
    do
    :: i < MASTER_SLAVE_MAX ->
        master_slave[master_index].is_slave_copy[i] = is_slave[i];
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;
    number_of_procs--;
    run master(master_index);
    goto end_operation;
  }

synchronize_with_new_master:
  master_chan!state_info(slave_index);
  if
  :: slave_chan?state_and_list(UNUSED);
  :: atomic
      {
        master_slave[master_index].master_slave_crashed ->
          goto handle_master_crash;
      }
  fi;
  atomic
  {
    do
    :: i < MASTER_SLAVE_MAX ->
        is_slave[i] = master_slave[slave_index].is_slave_copy[i];
        master_slave[slave_index].is_slave_copy[i] = false;
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;
    goto normal_operation;
  }

crash:
  atomic
  {
    master_slave[slave_index].master_slave_crashed = true;
    goto end_operation;
  }

crash_master:
  atomic
  {
    master_slave[master_index].master_slave_crashed = true;
    goto end_operation;
  }

end_operation:
  skip;
}

/* Master representative: */
proctype master(short master_index)
{
  byte  is_slave[MASTER_SLAVE_MAX];
  chan  acks_expected[MASTER_SLAVE_MAX] = [1] of {bool};
  chan  master_chan = master_slave[master_index].master_slave_chan;
  mtype message_type = 0;
  short slave_index_1 = 0;
  short slave_index_2 = 0;
  short slave_index_3 = 0;
  short i = 0;
  short nr_acks_expected = 0;

  atomic
  {
    do
    :: i < MASTER_SLAVE_MAX ->
        is_slave[i] = master_slave[master_index].is_slave_copy[i];
        master_slave[master_index].is_slave_copy[i] = false;
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;
    master_present  = true;
    index_of_master = master_index;
    number_of_procs++;
  }

continue_operation:
progress:
end:
  do
  :: master_chan?rw_request(slave_index_1) ->
      do
      :: atomic
          {
            i < MASTER_SLAVE_MAX ->
              if
              :: !master_slave[i].master_slave_chan??[state_update(UNUSED)]
                    && is_slave[i] ->
                  master_slave[i].master_slave_chan!state_update(UNUSED);
              :: master_slave[i].master_slave_chan??[state_update(UNUSED)]
```

```
                  || !is_slave[i] ->
                /* do nothing */
            fi;
            i++;
        }
    :: atomic
        {
          i == MASTER_SLAVE_MAX ->
            i = 0;
            break;
        }
    :: atomic
        {
          number_of_procs > 2 ->
            i = 0;
            slave_index_1 = 0;
            number_of_procs--;
            master_present = false;
            goto crash_master;
        }
    od;
    atomic
    {
      master_slave[slave_index_1].master_slave_chan!rw_reply(UNUSED);
      slave_index_1 = 0;
    }
:: master_chan?new_slave(slave_index_1) ->
    atomic
    {
      is_slave[slave_index_1] = true;
      do
      :: i < MASTER_SLAVE_MAX ->
          master_slave[slave_index_1].is_slave_copy[i] = is_slave[i];
          i++;
      :: i == MASTER_SLAVE_MAX ->
          i = 0;
          break;
      od;
      master_slave[slave_index_1].master_slave_chan!state_and_list(UNUSED);
    }
    do
    :: atomic
        {
          i < MASTER_SLAVE_MAX ->
            if
            :: is_slave[i] && i != slave_index_1 ->
                master_slave[i].master_slave_chan!new_slave(slave_index_1);
                nr_acks_expected++;
                acks_expected[i]!true;
            :: !is_slave[i] || i == slave_index_1 ->
                /* do nothing */
            fi;
            i++;
        }
    :: atomic
        {
          i == MASTER_SLAVE_MAX ->
```

```
              i = 0;
              break;
          }
      :: atomic
          {
            number_of_procs > 2 ->
              i = 0;
              slave_index_1 = 0;
              nr_acks_expected = 0;
              number_of_procs--;
              master_present = false;
              goto crash_master;
          }
      od;

continue_wait_ack:
      do
      :: atomic
          {
            master_chan??new_slave_ack(slave_index_2) ->
              acks_expected[slave_index_2]?true;
              nr_acks_expected--;
              slave_index_2 = 0;
          }
      :: atomic
          {
            nr_acks_expected == 0 ->
              goto continue_new_slave;
          }
      :: atomic
          {
            is_slave[1] && master_slave[1].master_slave_crashed ->
              slave_index_2 = 1;
              goto handle_slave_crash_new_slave;
          }
      :: atomic
          {
            is_slave[2] && master_slave[2].master_slave_crashed ->
              slave_index_2 = 2;
              goto handle_slave_crash_new_slave;
          }
      :: atomic
          {
            is_slave[3] && master_slave[3].master_slave_crashed ->
              slave_index_2 = 3;
              goto handle_slave_crash_new_slave;
          }
      :: atomic
          {
            is_slave[4] && master_slave[4].master_slave_crashed ->
              slave_index_2 = 4;
              goto handle_slave_crash_new_slave;
          }
      od;

handle_slave_crash_new_slave:
      atomic
```

```
{
  if
  :: acks_expected[slave_index_2]?[true] ->
      acks_expected[slave_index_2]?true;
      nr_acks_expected--;
  :: !acks_expected[slave_index_2]?[true] ->
      /* do nothing */
  fi;
  is_slave[slave_index_2] = false;
  /* clean up messages from crashed slave */
  i = len(master_chan);
  do
  :: i > 0 ->
      master_chan?message_type(slave_index_3);
      if
      :: slave_index_3 == slave_index_2 ->
          /* do nothing */
      :: slave_index_3 != slave_index_2 ->
          master_chan!message_type(slave_index_3);
      fi;
      i--;
  :: i == 0 ->
      slave_index_3 = 0;
      message_type  = 0;
      break;
  od;
}
do
:: atomic
    {
      i < MASTER_SLAVE_MAX ->
        if
        :: is_slave[i] ->
            master_slave[i].master_slave_chan!slave_removal(slave_index_2)
        :: !is_slave[i] ->
            /* do nothing */
        fi;
        i++;
    }
:: atomic
    {
      i == MASTER_SLAVE_MAX ->
        i = 0;
        slave_index_2 = 0;
        goto continue_wait_ack;
    }
:: atomic
    {
      number_of_procs > 2 ->
        i = 0;
        do
        :: i < MASTER_SLAVE_MAX ->
            if
            :: acks_expected[i]?[true] ->
                acks_expected[i]?true;
                nr_acks_expected--;
            :: !acks_expected[i]?[true] ->
```

```
                      /* do nothing */
                  fi;
                  i++;
              :: i == MASTER_SLAVE_MAX ->
                  i = 0;
                  break;
              od;
              slave_index_1 = 0;
              slave_index_2 = 0;
              number_of_procs--;
              master_present = false;
              goto crash_master;
          }
      od;

continue_new_slave:
      atomic
      {
        master_slave[slave_index_1].master_slave_chan!start(UNUSED);
        slave_index_1 = 0;
      }
  :: atomic
      {
        is_slave[1] && master_slave[1].master_slave_crashed ->
          slave_index_1 = 1;
          goto handle_slave_crash;
      }
  :: atomic
      {
        is_slave[2] && master_slave[2].master_slave_crashed ->
          slave_index_1 = 2;
          goto handle_slave_crash;
      }
  :: atomic
      {
        is_slave[3] && master_slave[3].master_slave_crashed ->
          slave_index_1 = 3;
          goto handle_slave_crash;
      }
  :: atomic
      {
        is_slave[4] && master_slave[4].master_slave_crashed ->
          slave_index_1 = 4;
          goto handle_slave_crash;
      }
  :: atomic
      {
        number_of_procs > 2 ->
          number_of_procs--;
          master_present = false;
          goto crash_master;
      }
  od;

handle_slave_crash:
  atomic
  {
    is_slave[slave_index_1] = false;
    /* clean up messages from crashed slave */
    i = len(master_chan);
    do
    :: i > 0 ->
        master_chan?message_type(slave_index_2);
        if
        :: slave_index_2 == slave_index_1 ->
            /* do nothing */
        :: slave_index_2 != slave_index_1 ->
            master_chan!message_type(slave_index_2);
        fi;
        i--;
    :: i == 0 ->
        slave_index_2 = 0;
        message_type  = 0;
        break;
    od;
  }
  do
  :: atomic
      {
        i < MASTER_SLAVE_MAX ->
          if
          :: is_slave[i] ->
              master_slave[i].master_slave_chan!slave_removal(slave_index_1);
          :: !is_slave[i] ->
              /* do nothing */
          fi;
          i++;
      }
  :: atomic
      {
        i == MASTER_SLAVE_MAX ->
          i = 0;
          slave_index_1 = 0;
          goto continue_operation;
      }
  :: atomic
      {
        number_of_procs > 2 ->
          i = 0;
          slave_index_1 = 0;
          number_of_procs--;
          master_present = false;
          goto crash_master;
      }
  od;

crash_master:
  master_slave[master_index].master_slave_crashed = true;
}

/* Initial process: */
init
{
  short i = 0;
  atomic
  {
    do
    :: i < MASTER_SLAVE_MAX ->
        master_slave[i].master_slave_crashed = false;
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;
    do
    :: i < MASTER_SLAVE_MAX ->
        master_slave[0].is_slave_copy[i] = false;
        i++;
    :: i == MASTER_SLAVE_MAX ->
        i = 0;
        break;
    od;

    master_slave_entries_used = 4;

    run master(0);
    run slave(1);
    run slave(2);
    run slave(3);
  }
}
```

Appendix B

Master-Slave Protocol – Pseudo-Code

This appendix gives a pseudo-code implementation of our fault tolerant master-slave protocol. The implementation includes an example of the use of the Globe location service, as explained in Section 4.2.

B.1 Master Implementation

```
task master is
    add master contact address to location service;
    loop
        get event;
        case event is
            when read/write request:
                handle read/write request;
            when new slave:
                handle new slave;
            when slave crash;
                handle slave crash or shutdown;
            when slave shutdown;
                handle slave crash or shutdown;
            when shutdown:
                handle shutdown;
        end case;
    end loop;
```

```
procedure handle read/write request is
    execute read/write request;
    send state update to every slave;
    send read/write reply to slave;
```

```
procedure handle new slave is
    add new slave to slave list;
    send state and slave list to new slave;
    send new slave message to other slaves;
    handle new slave acknowledgement reception;
    send start message to new slave;

procedure handle new slave acknowledgement reception is
    while not received relevant event from every slave loop
        get event;
        case event is
            when new slave acknowledgement:
                note new slave acknowlegement reception;
            when slave crash:
                handle slave crash or shutdown;
                note slave crash;
            when slave shutdown:
                handle slave crash or shutdown;
                note slave shutdown;
        end case;
    end loop;

procedure handle slave crash or shutdown is
    remove slave contact address from location service;
    remove slave from slave list;
    send slave removal message to other slaves;

procedure handle shutdown is
    send master shutdown message to slaves;
    exit;
```

B.2 Slave Implementation

```
task slave is
    slave initialization;
    loop
        get event;
        case event is
            when read request:
                handle read request;
            when read/write request :
                handle read/write request;
            when state update:
                update state;
            when new slave:
                handle new slave;
            when slave removal:
                remove slave from slave list;
            when master crash:
                handle master crash or shutdown;
            when master shutdown:
                handle master crash or shutdown;
            when shutdown:
                handle shutdown;
        end case;
    end loop;
```

```
procedure slave initizalization is
    start new slave incarnation;
    while not success loop
        send new slave message to master;
        receive state and slave list from master;
        receive start message from master;
        add slave contact address to location service;
    exception
        when master crash:
            start new slave incarnation;
            wait for new master;
            retry;
    end loop;
```

```
procedure handle read request is
    execute read request;
    send read reply to client;
```

```
procedure handle read/write request is
    send read/write request to master;
    loop
        get event;
        case event is
            when read/write reply:
                send read/write reply to client;
                return;
            when state update:
                update state;
            when new slave:
                handle new slave;
            when master crash:
                send master crash message to client;
                handle master crash or shutdown;
                return;
            when master shutdown:
                send master shutdown message to client;
                handle master crash or shutdown;
                return;
        end case;
    end loop;
```

```
procedure handle new slave is
    add new slave to slave list;
    send new slave acknowledgement to master;
```

```
procedure handle master crash or shutdown is
    while not success loop
        select slave from slave list;
        remove selected slave from slave list;
        if other slave selected then
            synchronize with new master;
        else
            synchronize with slaves;
        end if ;
    exception
        when new master crash:
            retry;
    end loop;
```

```
procedure synchronize with new master is
    begin
        send state to new master;
        receive state and slave list from new master;
    exception
        when new master crash:
            raise new master crash ;
    end;
```

```
procedure synchronize with slaves is
    remove master contact address from location service;
    remove slave contact address from location service;
    handle state reception;
    send state and slave list to every slave;
    send slave promotion notification to clients;
    start master representative task;
```

```
procedure handle state reception is
    while not received relevant event from every slave loop
        get event;
        case event is
            when state:
                update state;
                note state reception;
            when slave crash:
                remove crashed slave contact address from location service;
                remove crashed slave from slave list;
                note slave crash;
        end case;
    end loop;
```

```
procedure handle shutdown is
    send slave shutdown message to master;
    send slave shutdown message to clients;
    exit;
```

B.3 Client Implementation

```
task client is
    select slave using location service;
    loop
        get event;
        case event is
            when read request:
                handle read request;
            when read/write request:
                handle read/write request;
            when slave crash:
                select another slave using location service;
            when slave promotion:
                select another slave using location service;
            when slave shutdown:
                select another slave using location service;
            when shutdown:
                exit;
        end case;
    end loop;
```

```
procedure handle read request is
    while not success loop
        send read request to slave;
        receive read reply from slave;
        return read reply;
    exception
        when slave crash:
            select another slave using location service;
            retry;
        when slave promotion:
            select another slave using location service;
            retry;
        when slave shutdown:
            select another slave using location service;
            retry;
    end loop;
```

```
procedure handle read/write request is
    while not success loop
        send read/write request to slave;
        receive read/write reply from slave;
        return read/write reply;
    exception
        when slave crash:
            select another slave using location service;
            case request semantics is
                when at least once:
                    retry;
                when at most once:
                    return read/write failure;
            end case;
        when slave promotion:
            select another slave using location service;
            retry;
        when slave shutdown:
            select another slave using location service;
            retry;
        when master crash:
            case request semantics is
                when at least once:
                    retry;
                when at most once:
                    return read/write failure;
            end case;
        when master shutdown:
            retry;
    end loop;
```