Contents

Chapter 1

Introduction

1.1 Overview

Writing correct concurrent code is no easy feat. It is very difficult to get an unflawed design for a reasonably sized concurrent program [16] - in particular, most analysts and programmers, are not accustomed to deal with numerous threads or processes potentially modifying data in parallel and unordered manner. Moreover, testing such applications also provides an extra challenge to concurrent software development. Due to the massive number of possible code executions, debugging concurrent code tends to fail to expose all the possible bugs in a system [16]. This results in systems which are unstable when in production environment.

Distributed system design is at least as challenging as concurrent system design, with different kinds of failures threatening to make the job of producing correct code even harder. Some of the problems incurred with distributed systems are *process failures*, *link failures*, *lost messages* and *varying message arrival delays*[19]. Dealing with these problems in real life and ensuring that the code prevails any of these conditions, is and active area of research.

Lately, there seems to be an overall interest in shifting towards a decentralized paradigm for distributed computing. This paradigm forms the basis of various recent architectures and technologies including Web2.0, Cloud-computing and Peer-to-peer technologies. In the context of Distributed

Programming, this means that systems have to be programmed from a local observers [1] view which handles all interactions autonomously, without the need of a global coordinator. Such an approach overcomes the issue of single point of failure, nevertheless brings about various intricacies which need to be resolved to ensure the correctness of the distributed algorithms.

1.2 Aims and Objective

The aim of this dissertation is to investigate various decentralised distributed computing problems and study their applicability and possible implementation. The distributed system model will be analysed and the underlying difficulties when attempting to program in such a model will be explored.

A number of assumptions about this model shall be made. These assumptions are very much identical to those followed by alternative work on programming distributed systems [8]. In this context, different approaches to distributed programming will be assessed.

In this dissertation, some of the most popular distributed agreement classes will be investigated. Different specifications, for each class shall be presented and established algorithms will then be studied.

Following this, a common framework for implementing these algorithms should be developed in Erlang. This framework shall be built on top of Erlang's existing modules and built in characteristics. This way, we attempt to provide a practical implementation framework, for these algorithms.

A number of algorithms will be implemented in a reusable manner, in Erlang. Together, these will form a suite of algorithms, which abstracts away the intricacies involved in these algorithms, yet providing their functionality in a reusable form.

Finally, a test case application will be implemented, so as to assess the applicability of this implemented suite of algorithms.

1.3 Approach

The work taken throughout this project was organised in a particular structure. In particular, towards the early stages, we dealt more with research and attempted to get accustomed to the notion of distributed programming. Following this, we dealt more with the actual implementation of the project itself.

The various stages involved in this project were:

1. Performing research on distributed computing systems and the problems associated with programming is such systems.
2. Performing research on Distributed Agreement Problems.
3. Familiarizing ourselves with Erlang, and implementing some of the algorithms studied, for localized environment.
4. Developing an framework, with which to build the actual distributed algorithms, on top of Erlang's features.
5. Implementing and testing the distributed algorithms themselves using the developed framework.
6. Building a simple test case scenario which uses the algorithms developed as its core.

1.4 Dissertation Overview

The content of this report is divided into the following sections:

Chapter 2, lays out the background which is a prerequisite for this project. In introduces distributed systems and various other aspects dealing with distributed programming. In particular, the system model is outlined. Here, Agreement problems and the main classes of these problems are outlined. Different distributed programming approaches and languages are evaluated, and Erlang's strengths and advantages are pointed out.

Chapter 3 gives out a common framework for implementing the distributed agreement algorithms, in Erlang. This chapter attempts to bridge the theory and practice. It shows how we built on top of existing mechanisms in Erlang, so as to achieve this framework.

In Chapter 4 we start our study of the agreement algorithms. This chapter studies the Reliable Broadcast problem. Different specifications of this problem, are outlined and established algorithms are studied. An overview of the actual implementation in Erlang is given.

In Chapter 5, the Consensus class of agreement problems is studied. This chapter investigates different specifications and algorithms, and also gives details about the implementation in Erlang.

In Chapter 6, the Atomic commit class of agreement problems is investigated. Once again, here different algorithms are studied, and are implemented in Erlang.

Chapter 7 provides the details of the testing which was required for the system. It gives out the testing strategy followed for the implementation of the developed suite of algorithms. It also outlines the implemented test case application.

In Chapter 8 evaluates various aspects of this project. In particular, the usage of the algorithm suite, for the development of the test case, is investigated. The choice of Erlang, is also evaluated.

In Chapter 9, we conclude this work and propose suggestions for future work.

1.5 Conclusion

This project aims to provide the basic building blocks, which make the development of robust distributed systems easier. A number of recurrent distributed problems are outlined and studied in depth, in the light of process failures. Techniques for safe distributed programming, failure detection and fault tolerance are also investigated. A suite of protocols, providing solutions to the distributed problems outlined, is built. Finally this suite is tested and evaluated in a real world distributed scenario.

Chapter 2

Background

2.1 Introduction

This chapter provides an overview of the background for this project. Primarily, it attempts to motivate the use of distributed system. It presents a simple, yet realistic, example of a distributed system. A number of recurrent distributed problems are then introduced in the light of this example. All relevant terminology is defined and all design choices are explained and justified. In particular, an implementation language is chosen after being contrasted with other 'candidate' languages.

2.2 The need for distribution

In the context of computer systems, a Distributed System consists of a number of nodes operating together in a coordinated manner. Distributed computing is not a novel field. In fact, it has been around since the early days of networking and the internet. In reality, the need for such a system was felt since the early days of computing, but throughout recent years, their applications increased drastically.

Often computer systems have to deal with resources which are remotely located. These resources may vary from computing resources, to machinery and or even human resources. A distributed system would be needed to ensure communication amongst users and the sharing of these resources. As an example, the bank ATM systems may be considered. At the outer level, clients see a terminal were they can access banking services, but in reality these consist of a distributed computing systems, consisting of a number of resources.

Distribution is needed when a continuous operation needs to be guaranteed. In general, for a system to be failure proof, it needs to be replicated in different locations and when one component fails, the other takes over [4]. Moreover, distribution might also be needed to offer load balancing of core services - if a server is heavily loaded, one might consider to split the computation over a number of processors.

Finally, in today's world, where internet access is becoming more and more widespread, distributed systems have become part of every day life. Different computing devices nowadays access remote services for their operation, providing us with the instant and up to date information.

2.3 Motivational Scenario

One of the main problems with centralized systems, is their susceptibility to central point of failure - if this central node fails, service stalls. Clearly, overcoming this problem requires some form of distribution. This section presents a simple distributed, peer-to-peer filesystem *p2pFs* and presents typical problems incurred in such a distributed environment. This distributed system is used as a running example throughout this document - to clearly demonstrate instances of distributed problems being studied.

p2pFs is a simple distributed file system, *p2pFs* with the following properties:

1. **Decentralized** There is no central overall leader coordinating the filesystem.
2. **Consistent** At a particular time, all files appear the same from all nodes

This file system provides basic handling operations:

1. *File Creation:* Files can be created and are immediately available for other nodes.
2. *File Access:* Files can be read and written concurrently and reliably.
3. *File Deletion:* Files can be safely deleted.

Throughout this chapter, reference will be made to a particular setup of the *p2pFs*. For the sake of simplicity, this setup consists of just four servers: *North*, *South*, *East* and *West*. These four servers are interconnected together, however the underlying details of these connections are irrelevant to us - what is important is that a message sent from a server, *can* ultimately be received by the destination server. Figure 2.1 (a) shows the basic layout of this example.

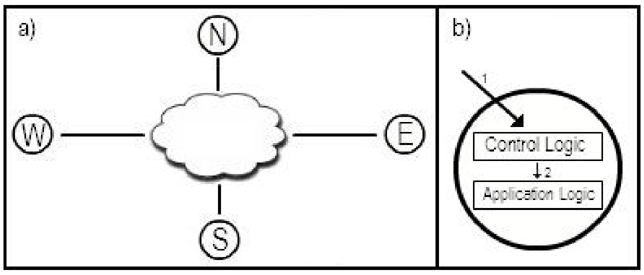


Figure 2.1: **(a) A distributed system using p2pfs. (b) Node composition**

When a node **sends** a message, this message is given to the underlying network to be delivered. It is assumed that the underlying network will eventually *transfer* the message with the same content as sent, to the destination (yet this message may take arbitrarily long to arrive). When a message is transferred to a node, it is said that the message is **received** by that node. Note, that here a distinction is made between a node and the underlying application (in this case *p2pFs*) - despite that the node received a message, *p2pFs* still does not know anything about this message and of its existence. When the message is passed to the application to which it is intended (*p2pFs*), the message is said to be **delivered** (refer Figure 2.1 (b)). Note that we are hence introducing the existence of a layer between the node and the application itself - which might decide that the message should not be delivered to the application, despite being received by the node. Alternatively this layer might decide that it needs to receive more messages before deciding to deliver this message or not.

It is noted that such a file system has very simple semantics, yet the intricacies can be very subtle. For example, how can a node ensure that updating a file will result in all nodes having the same version of the file? What if multiple nodes try to update it at once? What if some nodes fail after transferring their updates to only half of the nodes? What if a file is deleted while it is being updated by some node?

Such problems are yet another instance of *agreement* problems in a distributed system. This work, investigates and analyses different solutions to these problems. Ultimately these solutions are used to build a reliable version of the p2pFs, as described here.

2.4 Distributed Agreement Problems

Fault-tolerant distributed systems present various challenges which make this area of computer science an active area of research. An number of problems are recurrently dealt with when attempting to perform distributed computation[8]. Such problems explore different situations of data exchange, in a way to keep the overall systems consistent, despite the unreliability of the processes themselves and the interconnecting links. These set of problems are more commonly known as *Agreement Problems.* Here, a brief overview of these problems is given, and an example in the light of the distributed scenario being studied, is outlined. These problems will be studied in depth later.

1. **Reliable Broadcast** *Reliable Broadcast deals with the safe transfer of a message to a number of nodes, despite the eventuality of having nodes which fail during this process - including the sender itself.* [14] For example, in the *p2pFs* scenario outlined, consider the case where the W server wants to send an update a to all other servers. However, it might happen that it succeeds in sending the update to N and E, but crashes exactly before sending it to S (as shown in Figure 2.2)

   Clearly this leaves the system in an inconsistent state since server S does not get the update. A *Reliable Broadcast* protocol would be rather handy in this situation to ensure that a message is safely broadcast under all possible eventualities. Hence, if a *reliable broadcast* protocol is followed, all correct nodes should have a consistent view of the system.

2. **Consensus** *Consensus deals with agreement within a group of processes, in the presence of failures. Processes propose to each other and finally each process should decide on the values it received. In the*

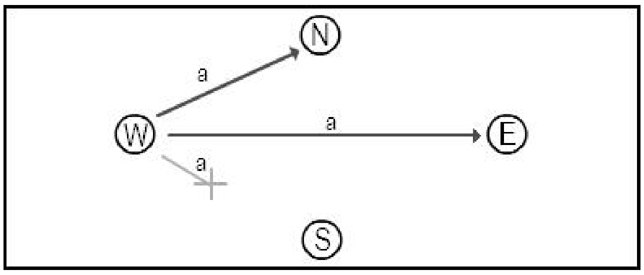


Figure 2.2: **Failure of simple broadcast mechanisms: Node S only manages to broadcast to part of the system.**

end all decisions should be consistent. [14] For example, assume that across the nodes there are different files with the same name. This is shown in figure 2.3 below, where a file with the same name (named A), is present on all nodes. Note also, that the size of file A is indicated at each node. Such a situation can occur when the distributed filesystem is initialised for the first time. If node W requests to open file A, which version should it open?

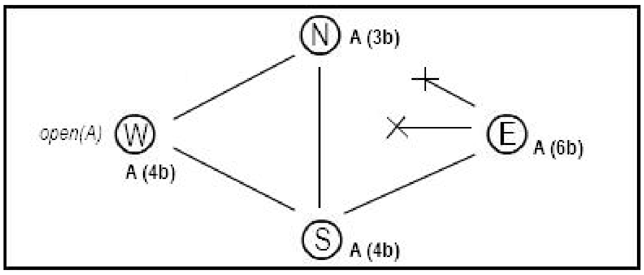


Figure 2.3: **Failure of Node E after submitting its version of the file to S, but before submitting to the rest, results in an inconsistent system.**

A simple solution might be everyone broadcasting his version of the file to oneanother and the receiver nodes always keeping the file which has a largest filesize . However, similar to the reliable broadcast scenario, if say node E crashes after sending its 6b version of file A to node S but before even sending the file to the rest, then node E will keep the 6b version of the file, whereas the other nodes will keep the $4b$ version of file A since this is the version of A with the largest filesize, they get to see. Clearly this is a problem and hence the *p2pFs* requires some consensus protocol to ensure that all nodes agree on the same version of the file.

3. Atomic Commit *The Commit problem deals with having a number of processes which try to perform some action together. This action should be aborted and rollbacked, if any of the processes disagrees to carry out this action, or crashes.* [14]

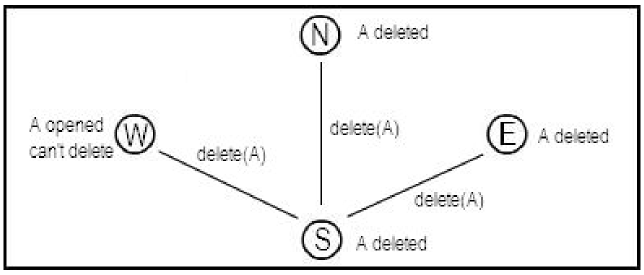


Figure 2.4: S **tries to delete a file from all nodes, however, node** W **cannot accomplish this operation because the file is open.**

Now consider that a *delete* operation on file A is issued from node S. This operation is broadcast to all other nodes (assume any a reliable broadcast), as seen in figure 2.4. However, the file is deleted from nodes N and E but is currently open at node W, and hence can not be deleted from node W. Again this would leave the system in an inconsistent state. An *atomic commit* would ensure that either all files are deleted or else no file is deleted until it can be deleted from all nodes.

These problems have been the basis of various research[19] [14]. A number of results have been established which identify limitations on the solutions to these problems, in various contexts of time and failures. Nevertheless, since these problems are central in the design of fault-tolerant distributed systems, various approaches have been proposed to go around these limitations in a well defined manner.[8]

2.5 Distributed Computing Models

This section will outline various computational models, which are relevant when investigating a distributed system. Such models can be used to analyse the correctness and bounds of the problem being studied.

2.5.0.1 Programming Distributed Systems

There exist two main schools of thought when programming for a distributed architecture. These are the **Shared-Memory Model** and the **Message-Passing Model**[16]:

Shared-Memory Model In the Shared-Memory model, processes interact by using common resources such as memory pages and registers. Higher level shared structures can also be composed on top of these structures, such as shared queues or stacks. Access to the shared objects is one classical instance of the Critical Section problem - it might be safe to have multiple readers but not multiple writers or a mixture of readers and writers accessing the shared object.[16] Various solutions to the *Critical Section Problem* have been proposed. The main ones are:

- **Semaphores and Mutexes**: These are constructs which restrict the number of processes executing within a Critical Section. In a nutshell, mutexes allow just one process to execute its Critical section at a time whereas semaphores allow an arbitrary number of processes to execute their Critical Section concurrently, depending on the initial value of the semaphore. Simply put, these lock constructs are a memory barrier. When a process tries to access a Critical Section, it waits until it acquires a lock. Contention on the locking depends on the underlying fairness of the lock and the scheduling algorithm. Having acquired the lock, a process executes its Critical Section and releases the lock.[16]

  Despite being rather straight-forward, lockful programming gives rise to various caveats including deadlock, livelock, priority inversion and the convoy effect. Moreover, lockful algorithms are not suitable for SMP because of their scalability limitations.

- **Lock-free** and **Wait-free Programming**: In an attempt to overcome the problems related to lockful programming, hardware manufactures started providing atomic primives such as *Test-And-Set*, *Compare-And-Swap*, *Load Linked* and *Store Conditional* operations. Such operations allow algorithms to be devised which despite avoiding the usage of locks, guarantee correct access to the shared resources for all possible histories of an algorithm. This requires the algorithm to be proven correct for all possible interleavings.[15] This is generally a streneous task, and despite that there are proven algorithms for some common data structures, more complex data strucutures and custom/hybrid data structures will, of course, have no guarantee of correctness. Moreover, lock-free code is note guaranteed to remain correct across different architectures (example if an assignment takes more than one CPU instruction). Wait-free programming goes a step further from Lock-Free programming, by giving timing constraints on the execution of the algorithm.[15]

- **Transactional Memory**: An emerging field with Shared Memory Programming, is that of Transactional Memory. Transactional Memory provides the programmer with an *atomic* construct for coarse grained atomic actions, which allows code within it to be executed concurrently in a safe manner. The approach is analogous to that of transactions in a database system: start executing the *atomic* code, if some other process interferes with this process' execution, then rollback the changes made and ensure that the processes execute in a consistent manner. This approach, especially the field of *Software Transactional Memory* started gaining ground, however as systems starts getting larger and the amount of concurrent *atomic* actions start to escalate. In particular, lots of processes start interfering, and more memory needs to be copied to allow for the reversing of its contents - degrading performance substantially. Moreover, such a system is sometimes deemed unsuitable for everyday computing, because most system calls, such as *read*() and *write*() are not easily reversible (especially if these access external resources such as the network).

Despite the severe performance problems outlined, lockful code seems to be the mainstream, possibly because of its programming ease and computational correctness. In order to overcome this problem, some programming languages, usually provide optimised libraries for common data structures based on lock-free and wait-free algorithms.

Message-Passing Model In a Message-Passing architecture, processes communicate by exchanging messages over some interconnecting link. Processes in this model, have local memories (distributed memory), which can be acted upon depending on the messages received. This model is also referred to as the actor model - processes can be seen as actors which interact with each other solely through the exchange of messages. Message Passing architectures can be classified as either *Asynchronous Message Passing* and *Synchronous Message Passing.* In Asynchronous Message Passing, *sending* messages is non-blocking, whereas in Synchronous Message Passing the sender blocks until the receiver delivers the message[15].

2.5.1 Degree of Synchrony

One other distinguishing attribute in distributed systems is the degree of synchrony - whether the system is synchronous or asynchronous. In section 2.7 , partially synchrony is also analysed as a way to circumvent the limitations of the problems outlined.
A system is *synchronous* if it obeys the following properties[14]:

1. A known upper bound exists on the message delivery delay.
2. A known upper bound exists on the time between execution steps of a process.
3. A known upper bound exists on the clock drifts of the interacting processes.

On the other hand, the *asynchronous* distributed system model does not provide any timing bounds on its operations.
The asynchronous model is more general but algorithms are harder to design in this model due to the lack of timing bounds[17]. Yet this model is very attractive since it has simple semantics, its algorithms are easier to adapt to real life applications due to the lack of any strict timing constraints and such algorithms are guaranteed to work with arbitrary timing bounds. [8]
The agreement problems outlined in 2.6 have been proven not solvable in an asynchronous model with crash failures[12], nevertheless these are solvable in

the synchronous model. The impossibility result for asynchronous systems revolves on the impossibility of distinguishing between failed processes and slow processes due to the lack of timing constraints.

It might be tempting to define *reasonable* upper-bounds in an asynchronous system, to be able to reason synchronously about the system. However, the aim of these models of synchrony, is to provide means of being able to prove properties about the system and there is no way to guarantee that such bounds will hold deterministically. Nevertheless, such approach has to be taken when all other measures fail [8].

2.5.2 Failure Model

During an execution, a component is said to have *failed*, if its behaviour differs from that specified by the underlying algorithm. On the other hand, *correct* components are ones which abide by the behaviour specified by their respective algorithm. A *Failure Model* specifies to what degree, this behaviour is tolerated to deviate, for the study of some underlying algorithm.

2.5.2.1 Process Failures

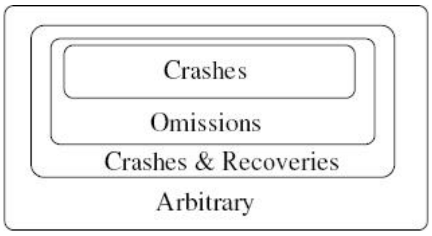


Figure 2.5: **Types of Process Failures. (source [14])**

A process is said to have failed, when it deviates from the algorithm which describes it. Upon failure, any other components under direct control of this failed process are assumed to also fail - and remain in this state unchangingly.

Figure 2.5, gives a diagramatic classification of different kinds of *Process Failures.*

The most general of these failures, are the **Byzantine Failures**. Such failures are denoted by arbitrary behaviour by the failed processes - processes can exhibit incorrect behaviour by sending incorrect messages or ommitting messages which should be sent. Likewise, such failed processes can react non-deterministically to received messages. Such abstraction can be seen as a process operating maliciously either intentionally or unintentionally - example due to miswritten code or user error. Byzantine Failures were first defined in the seminal paper [18], where the *Byzantine Generals Problem* is defined.

Crash Recoveries happen in processes whose operation is *transient* - such processes fail, remain in the failed state for some time and then go back to *correct* operational state. A process can continuously and repeatedly experience this behaviour. Processes which are transient failure resilient, generally store their internal state in some reliable storage medium. Whenever such processes recover, apart from consulting the stored state, the current state of the system is requested from the interacting peers of the distributed system.

Ommission failures occur when a process fails to send or receive a message. Such failures can be caused by link failures and also due to network congestion.

Crash failures Crash failures (or *fail-stop failures*) occur when a process fails and remains unchangingly in the failed state. A process can experience a crash failure in an arbitrary stage of execution, however under such model, no invalid messages are sent (in real life this is generally handled by transport layer protocols). In this work, we will focus mainly on *Crash Failures.* [14].

2.5.2.2 Link Failures

An interconnecting link may be considered as nonoperational if messages get garbled or are lost. The former case can be handled trivially with checksums. Lost messages on the other hand can be somewhat more difficult to handle[8]. From the point of view of processes, link failures appear as *ommission* failures - link failures may be the reason why certain messages are *ommitted.* In particular, in the case of network partition, such situation can give rise to various problems. For example consider a link failure causes a partition of network into two parts A and B. Processes in part A will think that all processes in part B have failed whereas processes in part B will think that all processes in part A have failed - yet both sets of processes will continue to operate in isolation.

Link Failures can be transient - certain messages are successfully delivered whereas others are lost. In practice, this happens due to router convergence delays[16]. When abstracting about process communication, it is common to assume a direct link between every communicating peer. In general, this direct link is most of the times an abstraction of the networking system which incorporates various other devices such as routers, switches, bridges, cables etc. The underlying networking system provides higher protocols with point to point communication seemless of the complexities involved. Moreover transport layer protocols (such as TCP) also handle convergence delays upto a limit - through mechanisms such as retransmission and windowing. On the other hand, a sender can ensure that a message was delivered by continuously retransimitting until an acknowledge is received[14].

In this work, only Process failures are considered. Links are assumed to be perfect.

2.6 Recurring Agreement Problems in Distributed Computing

Earlier on in this chapter, a number of recurring distributed problems were outlined. In this section we will study these problems in further detail.

2.6.1 Reliable Broadcast

Reliable Broadcast truly is a term which captures a number of broadcast primitives - different abstractions for reliably sending messages to a group of processes.

Broadcast abstractions provide different degrees of reliability: *Regular Reliable Broadcast* ensures that all correct processes deliver the same set of messages, *Uniform Reliable Broadcast* ensures broadcast agreement between both fault and non-faulty processes, *Total Order Broadcast* further ensures that all processes in the system deliver the messages in the same order whereas *Terminating Reliable Broadcast* further ensures that a process will not keep waiting to receive a message if no process has ever seen that message.

The *Regular Reliable Broadcast* algorithm, ensures that all correct processes agree on the message to deliver. A regular reliable broadcast algorithm satisfies the following properties [14]:

- **Validity**: If a correct process p broadcasts a message m, then p eventually delivers m.
- **Agreement**: If a correct process delivers a message m, then all correct processes eventually deliver m.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.

Regular reliable broadcast ensures agreement amongst all non-faulty processes. However, there can be a situation where the sender manages to send a message to one process and fails. The sender then fails without managing to deliver the process to any other process. Immediately afterwards the unique receiver of the message fails too, without having the opportunity to propagate the message it has received to the other nodes. In such a situation, there is inconsistency amongst the faulty and correct processes. Such a situation can be detrimental in certain situations and hence the *Uniform Reliable Broadcast* fixes this anomaly by strengthening the broadcast specification with the following property [14]:

- **Uniform Agreement**: If a message m is delivered by some process p_j (correct or faulty), then m is eventually delivered by every correct process p_i

Moreover, it is sometimes desirable to ensure that all broadcast messages (involving different senders) are delivered in the same order, globally by all processes. For example in the context of the p2pfs, consider the situation where one process alters the first character of a file, whereas another process tries to change the first character to uppercase. Clearly reordering the two operations, yields different results - hence it should be ensured that broadcast updates to files on the p2pfs are *totally ordered. Total Order Broadcast* ensures all process do not just agree on the set of messages received, but also the sequence of these messages. It also satisfies the following property [14]:

- **Total order**: If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

2.6.2 Consensus

The consensus abstraction deals with having nodes in a network trying to agree on a common value. The nodes themselves, initially propose values to one another and after that the consensus algorithm terminates, all nodes should have decided on the same value.

The most basic consensus abstraction is the *Regular Consensus*. A regular consensus algorithm satisfies the following properties [14]:

- **Agreement**: No two correct processes decide differently.
- **Validity**: If a process decides v, then v was proposed by some process.
- **Integrity**: No process decides twice.
- **Termination**: Every correct process eventually decides on some value.

Again as with the case of *Regular Reliable Broadcast*, there can be a situation where faulty processes decide differently (before failing), than correct processes. In order to restrict this situation, *Uniform Reliable Broadcast* satisfies another property [14]:

- **Uniform Agreement**: No two processes (correct or faulty) decide differently.

The problems with achieving reliable consensus is that of having processes which fail at any point during the consensus algorithm - processes can end up having different views of the set to be agreed upon. In [12], an impossibility result on the solvability of consensus in an asynchronous system, is given. This result states that even with simple crash failures, no algorithm can deterministically solve consensus in an asynchronous system. Nevertheless, in real life there are various ways of circumventing this limitation (refer 2.7).

In a synchronous system, consensus is solvable for the crash failure model but with a lower bound on the number of rounds - a round can be seen as an exchange of messages between the various nodes of the system. It is shown in [11], that if there are at most t failures, then consensus is solvable after $t + 1$ rounds.

2.6.3 Atomic Commit

The *Atomic Commit* problem deals with having a group of processes in a distributed system agreeing to perform an action consistently. Such an abstraction is particularly used in the domain of distributed database transactions - transactions may either be committed together or aborted together.

In a commit operation, the coordinator (the node initiating the commit operation), will request to perform a commit. All nodes attempt to perform the associate action and send **Yes** or **No** indicating whether this associated action succeeded or failed. If the coordinator sees that all nodes succeeded then it send a request to **commit** the result, otherwise it sends a request to **abort** the commit.

The first Commit abstraction which is considered is the *Blocking Commit*. This abstraction satisfies the following properties [14]:

- **Agreement**: No two processes decide to take different actions
- **Validity**:
  1. If any process votes **No**, then **abort** is the action which will be done.
  2. If all processes vote **Yes**, and there is no failure, then **commit** is the action which will be done.
- Weak Termination: If there is no failure, then all processes eventually decide.

This abstraction, is said to be *Blocking* because if the coordinator fails on particular stages in the algorithm, then the other nodes will not be able to decide whether to **commit** or **abort**

This problem is overcome by the **Non-blocking** commit protocol. This abstraction replaces the *Weak Termination* property with the following property [14]:

- **Non-Blocking Termination**: All processes eventually decide.

It has been proven in [8], that Atomic Commit is unsolvable in asynchronous systems, whereas the $t + 1$ lower bounds on the number of rounds, is still applicable for synchronous systems.

2.7 Circumventing Impossibility Results for Asynchronous Systems

Despite the fact that the agreement problems outlined here are unsolvable in asynchronous systems, such problems tend to crop up so often, that ways around these limitations had to be devised. In general, the correctness requirements are relaxed, the synchrony parameter is strengthened, or both [8]. There are three main techniques which are used: *Partial Synchrony*, *Failure Detectors* and *Randomisation*.

2.7.1 Partial Synchrony

Partially Synchronous systems attempt to strengthen properties of asynchronous systems, to obtain solutions to these distributed computing problems.

A *Partially Synchronous* model lies somewhere in between of a synchronous and an asynchronous model. Such a model has bound Δ on the message delay and bound ϕ on the relative speeds of the processes (slowest process takes at most ϕ multiplied by time taken by fastest process to perform the same action). Partially synchronous systems try to strengthen the bounds of asynchronous systems in some way. For example, a partially synchronous system may assume that the timing bounds do exist but are unknown. Alternatively, these bounds are also given a value which only applies after some time. Such restrictions are not found in asynchronous system, yet are far from those of totally synchronous systems [9].

2.7.2 Randomization

Randomization techniques allow processes to make probabilistic choices such that the properties of the abstraction are satisfied with some known probability. Processes make use of *random oracles* to determine what action to take at various points throughout their execution. Even though randomization gives no full correctness guarantees, sometimes it is the only practical way to avoid the limitations of the fully synchronous and asynchronous models. However, despite not providing full reliability guarantees, this model guarantees that the algorithms will eventually terminate.

Two important algorithms which solve consensus with randomization are the Ben-Or algorithm and the Rabin algorithm. The Rabin algorithm uses a shared oracle, whereas the Ben-Or Algorithm uses truly distributed oracles. [8]

2.7.3 Failure Detectors

Failure Detectors are distributed components which provide processes with information regarding which processes have failed and which are still running at a particular point in time. A failure detector is said to be *unreliable* if it can suspect that a process is crashed, when in reality this process is still running. However, these failure detectors can remove a suspected process p from their failed set, as soon as a p is detected not to have crashed. Each process in the system has its own private failure detector, and at any point in time, their output (set of suspected processes) can be different - however eventually these failure detectors are expected to converge their suspected processes sets.
A *Failure Detector* is defined by a pair (c, a):

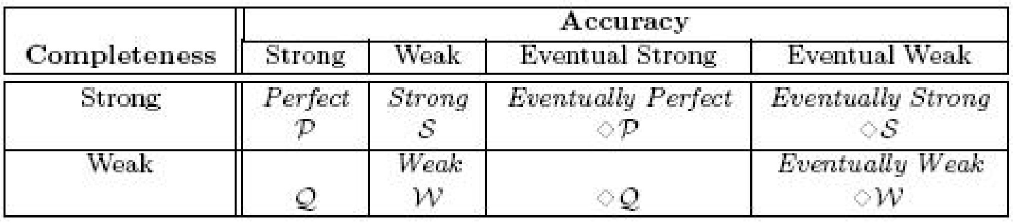
- c: A *Completeness* property specifying that all failed processes should be eventually detected
- a: An *Accuracy* property restricts the number of mistakes made by the *Failure Detector*.

Chandra and Toueg in [7], define two completeness and four accuracy properties. The *Completeness properties* are:

- **Strong Completeness**: Every process that crashes is eventually permanently suspected by *every* correct process.
- **Weak Completeness**: Eventually every process that crashes is permanently suspected by *some* correct process.

The *Accuracy Properties* are:

- **Strong Accuracy**: No process is suspected before it crashes.
- **Weak Accuracy**: Some correct process is never suspected.
- **Eventual Strong Accuracy**: There is a time after which correct processes are not suspected by any correct process.



Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Figure 2.6: **Tabulating the failure detector properties into eight classes (taken from [7])**

- **Eventual Weak Accuracy**: There is a time after which some correct process is never suspected.

Note that properties such as *Weak Completeness* and *Eventual Weak Accuracy* require that some property will hold permanently. In practice, this is never achievable because no process runs forever. Hence it is enough if such properties are satisfied for times which are "long enough" for the algorithm to make progress.[7]

Figure 2.6, shows a taxonomy given in [7] showing different failure detector classes according to the properties of accuracy and completeness. In [7] , Chandra and Toueg further show that a failure detector in any class of this Taxonomy can be used to solve consensus. It is interesting to note that the weakest[1] failure detector class $\diamond W$ in this taxonomy, can also be used to solve consensus. It is shown that $\diamond W$ is guaranteed to solve consensus only if $n > 2f$ where n is the number of processes and f is the number of failed processes - there is a majority of correct processes. Furthermore, in a related paper [6], it is proven that $\diamond W$ is the weakest failure detector class to solve consensus - that no weaker failure detector class can solve consensus in an asynchronous system. This is done by formalizing reducability of failure detectors and showing that any failure detector which solves consensus, is reducible to $\diamond W$. It is also shown that if a failure detector solves consensus with $n <= 2f$, then that failure detector is in a class stronger than $\diamond W$.[8]

It is important to note, that since the failure detector is an external component from the actual processes utilizing it, should the failure detector continuously suspect the wrong processes, such a condition would affect the

[1]A Failure Detector class is said to be weaker in specification than another Failure Detector class, if it can capture a larger number of failure detectors

"liveness but not the safety" [7] of the underlying algorithm. For example, a specifically written consensus solving algorithm utilizing such a failure detector, should result in nodes which never agree but never in nodes which agree inconsistently.

Failure detectors are quite an elegant approach to fault tolerant distributed systems. However, ultimately, how can failure detectors be sure that a process failed since they are still operating in an asynchronous environment? Quoting directly from [7]: *"Since we specify failure detectors in terms of abstract properties, we are not committed to a particular implementation. For instance, one could envision specialised hardware to support this abstraction. However, most implementations of failure detectors are based on timeout mechanisms".* This paper goes on to show a failure detector which works with timeouts. Every time a process is detected to have been suspected wrongly, its timeout is increased. The paper then states that despite this not being exactly the accuracy property of $\diamond W$, in most practical system such a scheme would eventually ensure that there is a correct process which is not suspected (*Weak Accuracy* of $\diamond W$) - recall that it is enough for such a property to hold for periods which are "long enough" for the algorithm to be able to make progress[7].

2.8 Fault Tolerant Systems

Fault Tolerance is a property which specifies the degree by which systems continue to operate reliably in the presence of failures. In the context of distributed computing, fault-tolerance requires some form of replication amongst at least two physically separate nodes. In his paper, *Why do Computers stop and what can be done about it?* [13] [4], Jim Gray points out that systems should be decomposed into units, which when fail, they do not affect the operation of other units. It is further pointed out *processes* are the ideal concurrent elements (as opposed to threads) since these do not share state amongst them and hence fully satisfy this requirement. Processes should then communicate with "copy messages" such that all interaction is message oriented and there is no shared state amongst processes. Schneider in [22], points out various properties which an individual processor should have for fault-tolerant operation. Processors satisfying these properties are called *Fail Stop* processors. The properties satisfied by *fail stop* processors are:

- **Halt on Failure**: Upon failing, a processor should cease operation immediately
- **Failure Status Property**: Processors should be notified when another processor fails, and given a reason for this failure.
- **Stable Storage Property**: Processors should be able to store data in some reliable medium which persists a node failure.

Furthermore, in [13], Gray applies this idea of *fail stop* processors to processes which is referred to as *fail fast*. Gray states that through defensive programming (such as checking all parameters, intermediate results and data structures), one can detect instantly detect failures. Upon such eventuality, the process should signal failure and stop. Such mechanism would avoid further damage caused by the errors and ensure slow latency on detecting errors. Renzel in [21], stresses this point even further by stating that the higher the latency time between the *occurence of the fault and the existence of the error*, the more complicated it becomes to perform a *backward analysis* of the error.

2.9 Distributed Programming Languages

A distributed programming language is any language which can be used to develop software which runs and controls a distributed system[1]. This class of programming languages, can be subdivided into two subclasses - 1) 'traditional' sequential programming languages augmented with libraries for distributed programming and 2) concurrent languages with distributed programming support.[1] The first type of languages are popular because these basically extend on languages to which programmers are accustomed. On the other hand, concurrent languages have gained popularity in the last few years because of their inherent support of constructs for communicating processes. Such languages utilize either of the two memory models outlined earlier: shared-memory or distributed-memory (message-passing). When programming distributed systems, message passing models have been found to be generally more appropriate because the programmer does not need to know where the other processes are located[1]. Distributed processes can even live on heterogeneous architectures - and all this does not affect the programmer of the system itself. [1]

In the past few years, a number of distributed programming languages each implementing its own flavour of message passing distribution, have been proposed. Andrews in [1] suggests that these languages should be compared according to their basic communication mechanism. Two classes of languages are identified according to their communication mechanism:

- **Send/Receive communication** The languages rely on send and receive primitives for communications. Languages with send/receive communication differ in the way in which communication is synchronized. For example Occam and PFX for .NET (Parallel Extension Framework)[2] use synchronous communication with blocking send and receive. Another distributed language with send/receive communication, is Erlang. Erlang uses asynchronous message passing with mailbox style concurrency - messages are delivered to the application not in the same way as they are received (or even sent), but in the order that they are requested by the application.

- **Remote Procedure Call** Remote procedures call (RPC) languages try to provide an abstraction of a normal procedure call. With such languages, a process can call a procedure or function which executes on another computer. After that this function terminates, it returns the result back to the caller. One such implementation is Remote Method Invocation (RMI) in Java. However, there are various arguments against RPC's. Steve Vinoski and Joe Armstrong argue that wrapping a remote operation and making it look as if it were local, will lead to problems because the failure modes of local and remote operations are completely different[5]. Not being able to determine whether a call is local or remote is also problematic when trying to optimize the code.

Though when it comes to computational power, both models are equally powerful, in practice every distributed language provides its own development techniques for solving distributed programming problems (such as coping failure)[1]. Moreover, in languages such as Erlang, it is rather easy to

[2]These two languages are not really distributed languages - rather these are concurrent languages.

”mimic” RPC’s with custom behaviour [5].

2.9.1 Erlang

Erlang is a functional programming language designed from ground up with a concurrency model in mind. Erlang follows the *asynchronous message passing* model but it also provides various other features resulting in a novel way to program reliable concurrent and distributed systems.

2.9.2 Programming in the presence of failures

The primary goal of this project is to provide basic building blocks for reliable distributed systems. Most of the distributed programming languages mentioned leave the job of failure detection and failure handling to the hands of the programmer. Moreover, no language takes into consideration software errors which might be caused by the programmer forgetting to handle certain types of data. One language which stands up against such arguments is Erlang, which as Joe Armstrong states, its purpose was to provide a mechanism by which to ”program systems which behave in a reasonable manner in the presence of software errors” [4]. Fault-tolerant distributed systems require careful design and Erlang’s primary aim is to address this requirement. Whereas Erlang does not directly solve the agreement problems outlined in section 1.2, Erlang does provide a language which by design eliminates most classical problems incurred in distributed programming, facilitates the structuring of code to avail from the natural concurrency of the underlying application and also provides a runtime environment with advanced distributed operating systems capabilities. Moreover, Erlang also helps overcoming problematic cases of unhandled datatypes, thanks to its dynamic typing system which can, according to Armstrong[4], helps handling instances which the programmer misses to explicitly handle in the code.

These features set apart Erlang from other concurrent languages (such as Occam) and concurrent/distributed libraries for traditional programming languages (such as OpenMP for C++ and Parallel Extensions Framework for .Net). In fact, these languages fail to define explicitly behaviour under failure.

2.9.3 The strength of Erlang

In his Phd thesis, Joe Armstrong identifies the major requirements for a fault-tolerant system and later on specifies how each of these requirements are met in Erlang [4]. These requirements are derived from the requirements outlined by Gray[13], Schneider[22] and Renzel[21] (see section 2.8).

These requirements are discussed in the context of Erlang in [4]. The following is a subset of these requirements which are directly related to this project:

- **Concurrency**: Erlang processes are very lightweight and the Erlang runtime supports thousands of concurrent processes. Processes interact by exchanging messages. In this project, the suite of Erlang behaviours will serve as an intermediate layer between user applications and the system, hence fast turnaround is definitely a requirement.
- **Seamless Parallel and Distributed Portability**: Erlang abstracts the distributed model into just mere parallelization. Processes communicating and interacting on a multicore machine, work well when distributed amongst several nodes. This will help us structure the testing and evaluation of code for this project - the libraries can first be test as a parallel program (on a single machine) and then distributed amongst a number of nodes.
- **Failure Detection and Fault Identification**: When a process fails, all linked processes are notified with the failure and given a reason for this failure. This will form the basis of failure detection in this project.

Moreover, Erlang also adheres with *fail fast* specifications of Schneider[22] and Renzel[21] Erlang processes immediately stop with a reason when a function is called with incorrect arguments.

Erlang was built to target robust code. The functional syntax of Erlang might be a repelling feature for, however it reduces the amount of code required between four to ten times [10]. Generally, more code means more bugs and hence these are suppressed as well. Moreover, thanks to the requirement of non mutable state functions, Erlang cuts down on the *shared state nightmares* of deadlock and race conditions [16] - this is because functions have no internal mutable state due to single assignment. These problems can only occur when interacting with the external world or when poorly handling asynchronous messages (example receiving messages out of order or when assigning wrong priority). Erlang also boasts a number of industry strength characteristics. The Erlang runtime machine provides advanced operating systems features such as automatic memory management and distribution of processes between nodes [3]. The Erlang runtime is also very robust with uninterrupted uptime. Moreover, Erlang comes with OTP (Open Telecom Platform) which provides standard patterns (or Behaviours) to extend fault tolerant code. This library includes behaviours for: Supervisors, Generic Servers, Generic State Machines and Event Logging [2]. Though not being a mainstream language, support for Erlang is not hard to find. It is very easy to interface Erlang with other languages. So, for example, Erlang can be used to code a robust server core, whereas the application layer components can be written in another language, with which developers are more confident, and ultimately plugged in to form one system. Finally, there is growing community of users which actively improves the whole Erlang environment with libraries, documentation and even ports of the runtime machine to different operating systems - highlighting the fact that Erlang is actually gaining ground after twenty years from its inception.

Most, if not all, of these features are clearly missing in alternative concurrent languages or frameworks - making Erlang the natural language of choice for various distributed projects. Among the various systems which are running Erlang code underneath the hoods, one finds: the Facebook's chat server, Wings 3D - a 3D graphics engine and modeller, Amazon's distributed database SimpleDB and an on demand distributed computing routing mesh at Heroku.

Erlang is usually criticized for its "odd" syntax and programming style. Recently, various clones of Erlang started emerging (such as Scala and Retlang library for .Net) and are proposing to overcome this problem. However, in doing so, these break the immutability Erlang boasts[20]. For this project, the protocol suite will be coded in Erlang. This will ensure that its code is availing itself from the benefits and robustness of Erlang. The application logic will then simply attach to this code - this need not even be written in Erlang itself. Erlang will serve as the layer which robustly performs critical and recurrent tasks, which then interfaces with the user's code.

2.9.4 Conclusion

This section served to introduce the main concepts behind this project. Various aspects of distributed computing have been outlined. In particular, a number of recurrent problems were outlined. These problems are the basis for the proposal in the next section. The requirements of distributed programming languages for reliable code and robustness in the presence of failures, where then outlined. Finally, Erlang is chosen as the implementation language for this project.

Chapter 3

Implementation Framework

3.1 Introduction

Various literature exists on the subject of *Distributed Agreement Protocols*. In particular, different algorithms have been proposed as a solution to the main distributed agreement problems (outlined in the previous chapter). In this thesis, various established algorithms are studied and implemented. These algorithms are adaptations of the work of Nancy Lynch [19] , Leslie Lamport [17], Toueg and Chandra [7], Guerraoui and Rodrigues [14].

In this chapter, a common framework for the implementation of such algorithms in Erlang, is outlined. This chapter builds on Erlang's features, to achieve a framework which provides the necessary environment for the implementation of such algorithms. In particular, this chapter proposes a way on how the agreement algorithms shall be implemented, so as to bridge the gap between theory and practice.

3.2 Distributed Computing Abstractions

Distributed computing consists of a number of hardware and software entities which compose the distributed system itself. From a conceptual view, processes communicate to one another, however there are various other entities which need to be considered such as nodes, failure detectors, links and the network itself. In practice, these are all depend on the deployment environment, so in this project an abstraction for these entities is done. These abstractions are mainly influenced by the way Erlang deals with these entities and the interface it presents to the programmer.

3.2.1 Processes

A distributed system is composed of a number of elements each performing some form of computation[16]. In this project, any computational element is abstracted with the notion of a *process*. A process is assumed to be characterised by:

- **Thread of Execution**: Every process consists of the execution of some code. Though no assumption is made on execution time, it is assumed that there is no unjustified indefinite waiting, but processes run to completion.

- **Local Memory**: Every process consists of a set of local variables. The value of all these variable at a particular point, make up the state of the process.

- **Unique Identifier**: Every process can be identified by a *process identifier* (PID). It is assumed that no ambiguity exists with processes identification - that is despite the distributed environment, no two processes (not even on distributed nodes) can have the same PID. Note that this is a fair assumption to make because Erlang guarantees such a property is true by encoding data about the node identification in the PID itself. Another way by which to identify processes, is through a *global identifier*. A *global identifier* is a systemwise unique name, which is an alternative to using PID's. Of course, with global names, the responsibility of appointing unique names is on developer himself.

- **Mailbox**: Processes can communicate by sending messages to PID's or *global identifiers*. When these messages are eventually posted to the process' mailbox - a storage place for these messages. The process can then fetch these messages and read their content.

3.2.2 Nodes

It is probably best to think of a node as a distinct interacting computer. However in reality, a node can consist of cluster of computers. Alternatively multiple node environments can interact on the same physical machine, without the need for any message to touch the network [16]. So it is better to think of a node as any form of replicated set of processes which together form this single entity - the node.

In this project, every Erlang's runtime environment (or shell), is considered as a node. Every shell, hosts a number of Erlang processes. Multiple Erlang shells can be opened on the same machine, or else every shell can be hosted on a different machine.

A node is taken as the unit of failure - if a process crashes, all other processes together in that node would crash. In terms of Erlang, this means that all processes within a shell are linked, and the failure of one process, would lead to the abrupt termination of all the other processes in that shell.

3.2.3 Communication Links

Interconnecting distributed nodes involves various infrastructure such as routers and switches. This setup varies greatly depending on the underlying network architecture. In this project, all this infrastructure is captured by the *link* abstraction. A *link* exists between two nodes if a message can eventually be delivered between the two. This connectivity is the responsibility of lower level protocols such as routing protocols.

In practise, a link might exist between two processes, but messages could be lost in transmission. Hence, network communication is assumed to be unreliable. Note that however, the network is assumed not to generate messages on its own.

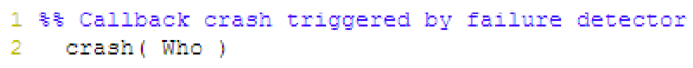
A simple workaround is used to overcome the problem of unreliable communication. Whenever a process P_i sends a message to P_j, this message can be repeatedly sent by P_i until it receives an acknowledgement from P_j. Such a mechanism would guarantee *eventual delivery* the message, given that none of the two processes actually crashes. [14] Moreover, so as to avoid that P_i blocks until the message is actually delivered by P_j, this repetitive sending could be done by a separate process. Note that such a mechanism, does not guarantee that every message sent by P_i will reach P_j - it could happen that a message P_i sent is lost and P_i crashes before resending the message again.

Such a mechanism is typically handled by lower level protocols such as TCP. Erlang makes use of such protocols [2], and hence the usage of such a mechanism is implied. Note also that links are assumed not to create message erroneous message on their own (*no creation*).

3.2.4 Failure Detectors

In the previous chapter, the notion of failure detectors was explained. These provide information as to which processes it detects as having crashed. Failure detector can only give information as to whether a process is alive or not, but does not give information about the state of the processes (such as its point of execution).

Moreover, in this project a *Perfect Failure Detector* is assumed. This means that the failure detector classifies a process as having crashed, then that process really did crash (*accuracy*). Moreover, it will eventually detect all crashed processes (*completeness*). When this *Perfect Failure Detector* classifies a process as having crashed, it triggers a *crash* event. The prototype of this event is given in listing in figure 3.1 below:



```
%% Callback crash triggered by failure detector
  crash( Who )
```

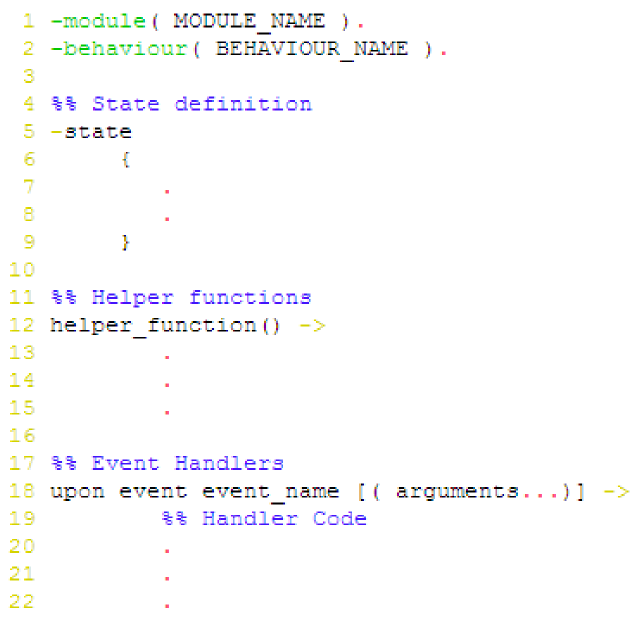
Figure 3.1: **Prototype of the Perfect Failure Detector crash event**

The failure detector triggers this event on all modules which utilize it. With this event, it returns an argument *Who*, which denotes the name of the process which crashed.

3.3 A Common Structure for all Algorithms

This section will present the overall structure of *agreement algorithms* implemented. This structure serves as a common way to present these algorithms in this project. Moreover, various syntactical details will be outlined and briefly described. Later on in this chapter, an actual Erlang implementation of this proposed layout, is presented.

The algorithms presented here will follow an *Asynchronous Event-handling Model*[14]. Basically all algorithms here will have the format of an *event* and its *handler*. The *template* for algorithms given in this presentations is given below:



```
-module( MODULE_NAME ).
-behaviour( BEHAVIOUR_NAME ).

%% State definition
-state
      {
          .
          .
      }

%% Helper functions
helper_function() ->
          .
          .
          .

%% Event Handlers
upon event event_name [( arguments...)] ->
          %% Handler Code
          .
          .
          .
```

Figure 3.2: **Structure of algorithms given in this presentation**

As can be seen in figure 3.2, algorithms consist of three main sections: a *State* definition, a number of *helper* functions and a number of *event* handlers. Prior to these there is also a module name and behaviours implemented. All these are explained in detail below:

- **Module Name**: The module definition defines the name of the module. It is used when external modules need to call functions or trigger events locally defined in this module.

- **Behaviours implemented**: This concept comes directly from the Erlang behaviours. In Erlang, a behaviour is a guarantee that this module will implement a number of functions which are requested by the behaviour module. Typically the behaviour module will then callback these functions during its operation.

- **State Definition**: This *State* definition outlines the constituent elements of the State record. Simply put, the state record is a globally available record which contains data stored by the algorithm until some particular point in time. In practice, due to Erlang's programming methodology, internally this state is passed as a parameter to the handler function or to the message handling loop.

- **Helper Functions**: Helper functions are common routines which are either used extensively, or which help structure the code better. These functions are invoked from within the event handlers or from other helper functions. These functions can take arguments and return a value to its caller.
- **Event Handlers**: Event handlers define the actions which need to be taken when an event is triggered. These actions are invoked in isolation and each handler needs to be non blocking. Event handlers are essentially what makes up the algorithm itself. These can be seen as the entry points to the algorithm.

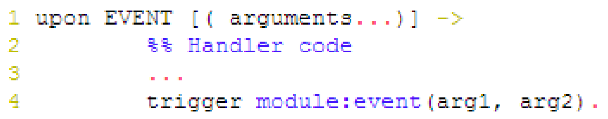
In a nutshell, every algorithm is presented as a sequence of responses to particular events. During an execution of the algorithm, events received are queued in chronological order. The program will then repeatedly dequeue an event and execute the corresponding handler or wait for more events to be triggered. Event handling is carried out asynchronously.

3.3.1 Types of event triggering

Events can be triggered in two ways. Firstly, an event may be **directly-triggered** by some module through the use of the *trigger* keyword. The other type of event triggering is **predicate triggered** event triggering. Such events will be triggered when a particular conditional expression becomes true.

3.3.1.1 Directly-triggered events

Directly-triggered events can be triggered from any module. Conceptually, these can be seen as function calls which have asynchronous message handling semantics - an event is triggered in a similar fashion to a function call, however, the process handle them sequentially, even if multiple processes trigger events to this process, at the same time. As an example of directly triggered events refer to the listing in figure 3.3. The event header is given on line 1, preceded by the *upon* directive. The *upon* directive essentially helps create a distinction between normal helper functions and event handlers. The remaining code, after the event header is the handler code. As an example of the *directly-triggered* event notation, a sample triggering is done on line 4. This notation for triggering an event (and even external helper functions) is identical to the MFA (Module Function Arguments) notation of Erlang (*Module name : function name(arguments...)*). Note that the *trigger* keyword is used to signify that the event is invoked asynchronously as opposed to a function call.

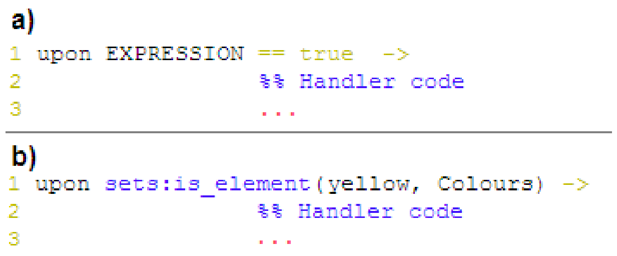


```
1 upon EVENT [( arguments...)] ->
2         %% Handler code
3         ...
4         trigger module:event(arg1, arg2).
```

Figure 3.3: **Syntax of direct triggered events and event handlers.**

3.3.2 Predicate-triggered events

Predicate triggered events are triggered when some particular conditional predicate becomes true. These are like runtime monitors for a particular state (or sub-state) to be reached. When this happens, a local event is triggered. As an example of predicate-triggered events, refer to the listing in figure 3.4. Listing 3.4a shows the template code for a predicate triggered event handler. On line 1, preceded by the upon keyword, there is the predicate condition which will trigger this event. As an example, listing 3.4b shows an event which is triggered when a set (Colours) gets an element with value blue.



a)
```
1 upon EXPRESSION == true  ->
2                 %% Handler code
3                 ...
```
b)
```
1 upon sets:is_element(yellow, Colours) ->
2                 %% Handler code
3                 ...
```

Figure 3.4: **Syntax of predicate triggered events.**

3.4 Module Composition

The suite of Erlang behaviours given here is composed of a number of modules, each implementing some particular algorithm. Moreover, much of these modules build on one another; several modules extend the functionalities of other modules. This can be achieved rather easily through the use of behaviours. Essentially this creates a layered stack of modules which interact with each other through well defined interfaces. These interfaces are simply well defined events which may be triggered or which can be handled by the module. Hence an *interface* for a module will consist of:

- **External Events Handled**: These are events, which can be triggered by some external module and for which an event handler resides within the module. These can be seen as asynchronous requests which can be posted to the module.
- **Callbacks Expected**: A callback is an event which is triggered and is expected to be handled by the user of this module. Typically these events are triggered at "milestone" points during the execution, and signify that some operation requires attention. For example, a module may invoke a callback when it has processed or received some form of data.

Through these well defined interfaces, modules can extend and interact with each other much like with interfaces and inheritance in Object Oriented Programming. This is done by having modules which trigger events in other modules. This latter module performs some processing and then it triggers an event in the first module in the form of a callback.

As an example, consider a module for broadcasting messages called *broadcaster*. This module is able send out messages, by handling the *broadcast(Data)* event. Moreover, when a message is received, it callbacks the *deliver(Data)* event. Now consider, a module called *namer*, which is used to send and receive names to other processes. It handles the *send_name(Name)* event when a name is to be sent and callbacks the *received_name(Name)* event, when a name is received. For the sake of the example, assumed that the *namer* module, does some additional processing on the *Name* before broadcasting it.

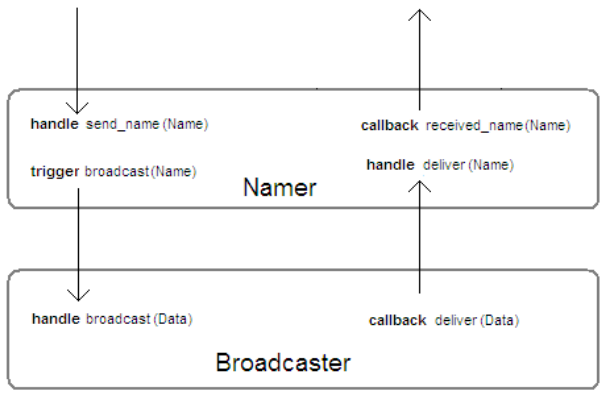


Figure 3.5: **An example of module layering**

Figure 3.5 gives a possible composition of the two modules. The *namer* module, makes use of the *broadcaster* module to transmit and receive names, and perform additional processing on them. It should be noticed that the *namer* module, not only requires to trigger the right event on the *broadcaster* module, but also needs to implement the necessary callback function, expected by the *broadcaster* module.

3.5 Delving deeper

In this section, a brief overview of various elements which make up the language will be given. The essential constructs used throughout this presentation as well as standard notations, will be outlined. Note that this is merely an overview and not a complete explanation of these tools or techniques being used in this presentation. Most of these techniques are directly borrowed from Erlang (or functional programming in general). The reader is expected to be familiar with functional programming techniques.

3.5.1 Variables

As with any programming language, this language will have variables. Variables identifiers consist of an alphanumeric string starting with an uppercase alphabetic character. In order to keep algorithms concise, in this presentation the single assignment is relaxed - state variables can be assigned mutliple values. However in section 3.2.4, a method to convert to pure single assignment, as used in Erlang, is described.

3.5.2 Atoms

Atoms are simply symbols (identifiers) which can be assigned. As an example, *yellow* in Listing 3.4, is an example of an atom. Atoms are different from enumerated variables, in that two atoms with different names, can never match. Atoms are identifiers starting with a lowercase alphabetic character. Atoms are one of the main elements of Erlang.

3.5.3 Comments

Comments can be written anywhere in the code by preceding the comment with a % character. This comment will span until the first newline which is encountered.

3.5.4 Tuples

Tuples are a way in which data elements can be grouped together as a single entity. These are very much like records in conventional programming languages. These can be used to group together different data elements, and pass them together as a single variable.

3.5.5 Pattern Matching

Pattern Matching is a technique widely used in functional programming. Simply put, pattern matching will ensure that a particular action is taken, when data of a particular format is to be processed. Pattern matching can be used in most of the constructs including if and case statements, and also for function execution itself. A function can be declared with a particular data pattern in the header. This function will only be executed when that pattern matches the data at runtime (and no other function has pattern matched). Putting a Variable, in the place of a pattern, will match all data patterns. Although understanding pattern matching in practice, can be rather straight forward, giving a complete explanation of this subject would require a lengthy explanation. For this reason, for a complete explanation of pattern matching please refer to [4].

3.5.6 Sending and Receiving messages

Since this project deals with distribution over remote nodes, there must be a way for communication amongst these nodes to take place. This is done by using message sending and receiving constructs. These constructs are taken directly from the Erlang language.

Sending messages is done through the use of the ! construct. This construct has the following structure:

```
<PROCESS IDENTIFIER> ! <MESSAGE}
```

The semantics of this construct is that the message on the RHS of the ! is sent to the process identified by the process identifier given. Process Identifiers are a way by which to reference processes, and these are explained in more detail in a later section 3.2.1. Note that the message sent can be of any type.

Receiving messages is done using the *receive* construct. This structure of this construct is explained below:

```
receive
PATTERN1 -> ACTION
PATTERN2 -> ACTION
...
[ELSE  -> ACTION]
end.
```

The receive construct waits until a message is available and then invokes an action corresponding to the message type. Since a message can have different types, pattern matching is done on the messages received to determine what action to take.

3.6 An Implementation in Erlang

The various notions which have been outlined so far need to be implemented in Erlang. This section gives an overview of the mapping between the abstractions given here and how these were implemented in Erlang.

3.6.1 Process structure

Every distributed node contains a particular organisation of Erlang processes which together form the orchestration layer of every node. These processes have structure shown in figure 3.6 below:

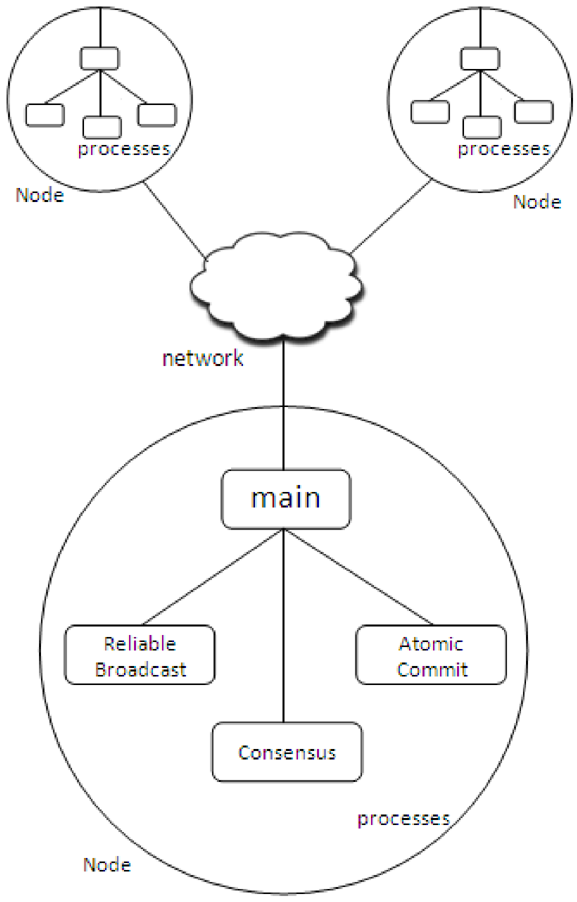


Figure 3.6: **Organisational structure of processes in distributed nodes.**

3.6.1.1 Inter-node Communication

Every node contains one *Main process*. The aim of this process is to serve as an incoming message gateway: all messages coming from external nodes will be received by this process and forwarded to the associated receiver module. It can be noted that for a process to be able to send a message to another node, it must know the PID of the receiver. This means that if it requires to communicate with all processes, it would require to store all PID's for the processes from all nodes (even though such communication pattern is rarely required). However, in the approach taken with the *Main* process, only the PID of the *Main* processes is required to be known. Sender processes will send their messages to the *Main* process of the destination node, which will in turn take care of "forwarding" the message to the corresponding receiver process.

But how does the main process know which process is the intended receiver of the message? Here some pure Erlang techniques come into play.

Since it is assumed that this suite is replicated on all nodes, the set of processes for this suite is the same on every node. Moreover, as explained earlier, Erlang provides a way by which to register process names - giving names to processes which can be used for communication in the place of the PID. Hence, each process can be given a pre-defined name and the same names are used on all the nodes.

Now in order for the sender to indicate to the *Main* process, which process is the intended receiver, it can send the pre-defined process name of the receiver as part of the payload. In Erlang, this is achieved by sending the message data which was originally intended to be sent, together with the pre-defined name of the process, as a single tuple. When the *Main* process, at the destination node, receives this message, it can read the registered name of the process directly from the message, and forward it the original message.

The actual "forwarding" of this message will occur by triggering an event. The event triggered is the *received* event. Its prototype is given below in the listing in figure 3.7:

```
%% Callback received triggered when data is received
received( Msg )
```

Figure 3.7: *received* **event prototype**

So now, if a process receives a message, a *received* event will automatically be triggered. This means that the process does not need to block and wait for some messages to be received, but rather when the message is received, this event handler will be triggered.

3.6.1.2 Utilizing the Erlang Failure Detector

A lot of emphasis has been made on the use of failure detectors as means by which to abstract knowledge about failures. However, in figure 3.6, these do not seem to have a directly associated process. The reason for this is because here we are building on top of Erlang's existing failure detector. Erlang provides means by which to monitor processes for liveness. This is done by *link*ing processes together and setting up processes to *trap EXIT* signals. When a process crashes, all the other processes will be notified with this *EXIT* message.

The handling of the *EXIT* messages sent by Erlang's failure detector are handled in the *Main* process. The *Main* process is *linked* to all other remote *Main* processes in the system. It also traps *EXIT* messages and triggers a crash event to all processes on the node.

Internally, the Erlang's failure detector uses a heartbeat mechanism to detect failures. In terms of failure detectors as presented by Chandra and Toueg in [7], this has the following implications:

- **Completeness**: Given that if a process crashes, it will not send out any heartbeats and so, given that these messages could only originate from the failed node itself, all monitoring nodes will detect its failure. This is known as *Strong Completeness*.
- **Accuracy**: A heartbeat mechanism is, however, prone to network delays especially when there is a network overload [8]. This could lead to a situation where the failure detector would classify a process as faulty, when in reality it is still alive. Moreover, Erlang's failure detector will not attempt to reassess the liveness status of a process which is detected as failed - once a process is suspected to be failed, it will not be classified as alive (unless the process is manually re*link*ed). This violates the *accuracy* property because it causes processes to be falsely detected as having failed.

Hence Erlang's failure detector can be merely seen as performing any special failure detection. However, as limited as it is, it still helps separating concerns about failure and failure detection from the rest of the code. Moreover, studying and devising reliable algorithms to ensure the strength of the failure detector, would be another research project on its own.

For this reason, it was chosen to work with the existing failure detector and propose the extension of this failure detector for future work. In order to partially 'patch' this shortcoming, in this project, a simple approach was taken - any process which is detected as failed will be sent a *KILL* message to truly kill itself and broadcast that failure message to all other processes. This technique ensures that the accuracy property is now satisfied, despite inaccuracies due to network delays. Note that however, the process which is detected as having failed, keeps doing its execution until it receives the *KILL* message. During this period, it may even take conflicting actions with what the process which detected it expects. This might break certain properties of the algorithm. In this work, examples of where this might happen are pointed out.

3.6.2 Local Processes Interaction

Figure 3.6 indicates that every node houses at least three other processes, apart from the *Main* process. These are *Reliable Broadcast*, *Consensus* and *Atomic Commit*. In reality, these are the classes of algorithms which are implemented in this project - each class contains a number of algorithms. Essentially, these are each handled by a separate process which maintains the state of that particular algorithm.

These algorithms were each implemented in separate modules. This module, essentially, handles events it receives and changes the state accordingly. Below is a list of all the modules implemented:

- **Reliable Broadcast**
    1. **be_rb**: Best-Effort Reliable Broadcast
    2. **r_rb**: Regular Reliable Broadcast
    3. **u_rb**: Uniform Reliable Broadcast
    4. **c_rb**: Causal Reliable Broadcast
    5. **to_rb**: Total Order Broadcast
- **Consensus**
    1. **rf_c**: Regular Flooding Consensus
    2. **rh_c**: Regular Hierarchical Consensus

3. **uf_c**: Uniform Flooding Consensus

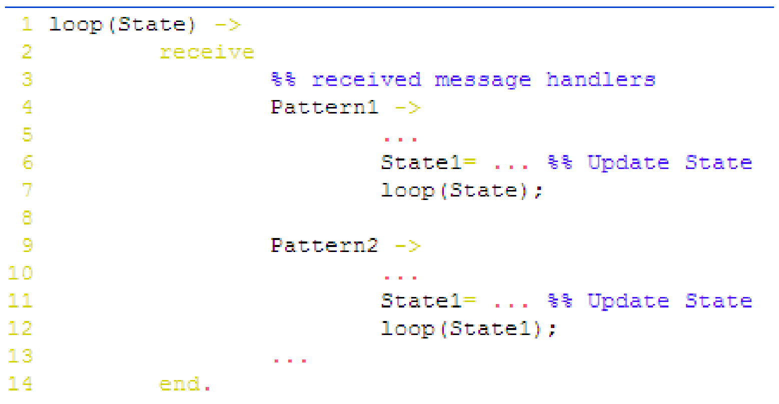
- **Non-Blocking Atomic Commit**

1. **nb3p_ac**: Three Phase Commit
2. **cb_ac**: Consensus Based Commit

Each of these modules will be explained in detail in the coming chapters.

3.6.3 A Design Pattern for all modules

All processes maintain an internal state by having a recursive function which waits for new events, handles them and recurses with the new state. If this is to be done in Erlang, it would look something like the code in figure 3.8.



```
1  loop(State) ->
2          receive
3                  %% received message handlers
4                  Pattern1 ->
5                          ...
6                          State1= ... %% Update State
7                          loop(State);
8
9                  Pattern2 ->
10                         ...
11                         State1= ... %% Update State
12                         loop(State1);
13                 ...
14         end.
```

Figure 3.8: **Maintaining an internal state through recursive processes in Erlang**

Here, a *loop* function is defined. This function takes one argument - the State. It waits until a message is received. The corresponding handler is invoked depending on the message type. Note that the handler will update the State (line 6) and then recurse with the new state. The new State is function of the old state and the message received. Note that generally the state consists of a tuple with a number of data values.

A more generic form of this design pattern, has been implemented as part of Erlang/OTP modules. This is known as the *gen_server* behaviour, which is an implementation of a generic server. The *gen_server* is written to make code more scalable and maintainable [4]. It also gives way to easier debugging and easily provides features such as dynamic upgrades. Moreover, by using the generic server, the code will be structured in a 'standard' way - making it easier for external programmer to understand and work with the code.

In this project, all processes are implementations of the *gen_server* behaviour. As mentioned, all algorithms listed in 3.6.2 have an associated process. This process implements the *gen_server* behaviour. Moreover, these processes can be seen as a real-world implementation of the template for all algorithms described here (refer to figure 3.2). The *gen_server* provides the roadmap from this template code to actual Erlang code.

The following section present another process template - this time, it shows how everything discussed in this chapter can be implemented in Erlang.

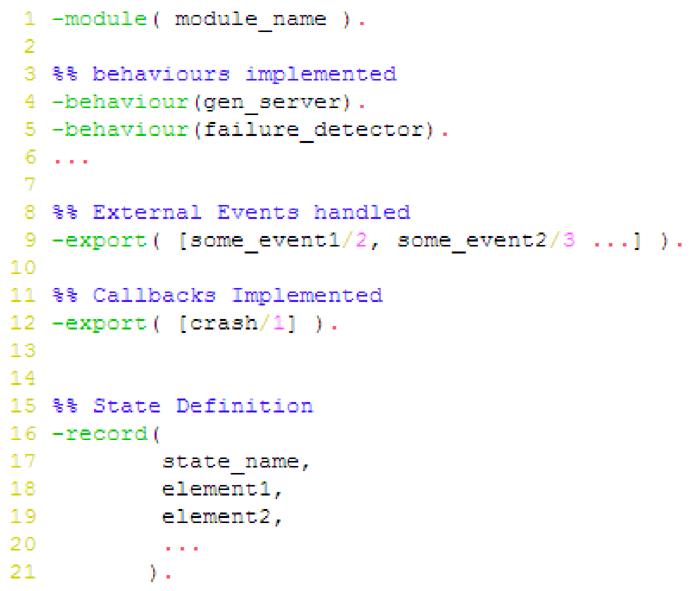
3.6.3.1 Putting everything together

In order to present everything together, a template depicting the organisational structure of source code, as implemented in Erlang, was devised. In this section, various parts of this template will be discussed, and their mapping with what has been discussed in this chapter, will be outlined.

Listing 3.9 shows the first part of this template code. Apart from the *export* declarations (lines 7 to 11), this code is very similar to that in figure 3.2. In fact, here the module name, behaviours implemented and state record are defined. Here the *gen_server* behaviour (line 4) and *failure_detector* behaviour (line 5) are implemented. The *export* statement on line 9 is used to list out the functions which other modules can call to 'trigger an event'. These exports are given as a list of function names followed by their arity: so here two events are being published (*some_event1* and *some_event2*). The *export* on line 12, lists the functions which act as the callbacks implemented for other modules. The *crash* event is triggered by the failure detector and should be implemented by all processes using the failure detector.

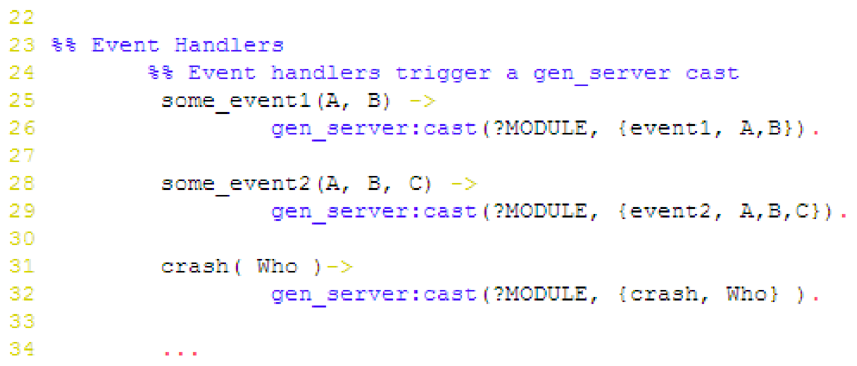
Thus, in terms of code, an event is simply a function call defined in a module. That function call will, however, not handle the event itself.

Listing 3.10 gives a definition of the two example events exported and also of the *crash* callback. These consist (lines 26, 29 and 32) of just a *gen_server cast.* A *gen_server cast* is simply an asynchronous request to a *gen_server*



```
-module( module_name ).

%% behaviours implemented
-behaviour(gen_server).
-behaviour(failure_detector).
...

%% External Events handled
-export( [some_event1/2, some_event2/3 ...] ).

%% Callbacks Implemented
-export( [crash/1] ).


%% State Definition
-record(
        state_name,
        element1,
        element2,
        ...
       ).
```

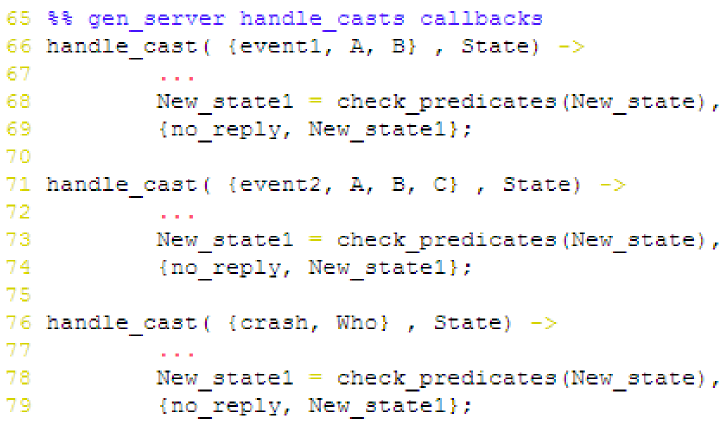
Figure 3.9: **Template with code for all algorithms implemented (Part 1)**



```
%% Event Handlers
       %% Event handlers trigger a gen_server cast
        some_event1(A, B) ->
                gen_server:cast(?MODULE, {event1, A,B}).

        some_event2(A, B, C) ->
                gen_server:cast(?MODULE, {event2, A,B,C}).

        crash( Who )->
                gen_server:cast(?MODULE, {crash, Who} ).

        ...
```

Figure 3.10: **Template with code for all algorithms implemented (Part 2)**

which is not expected to return anything back. Here a cast is sent to the *gen_server* started by this module (identified by the ?MODULE macro). Notice that in the cast, the second parameter is a tuple with all the parameters of this event and an atom identifying the event. This tuple is the request to be handled asynchronously by the *gen_server*. The reason for not handling these directly here is twofold. Firstly because at this point we have no access to the *State* and secondly is to ensure that all requests are processed sequentially.

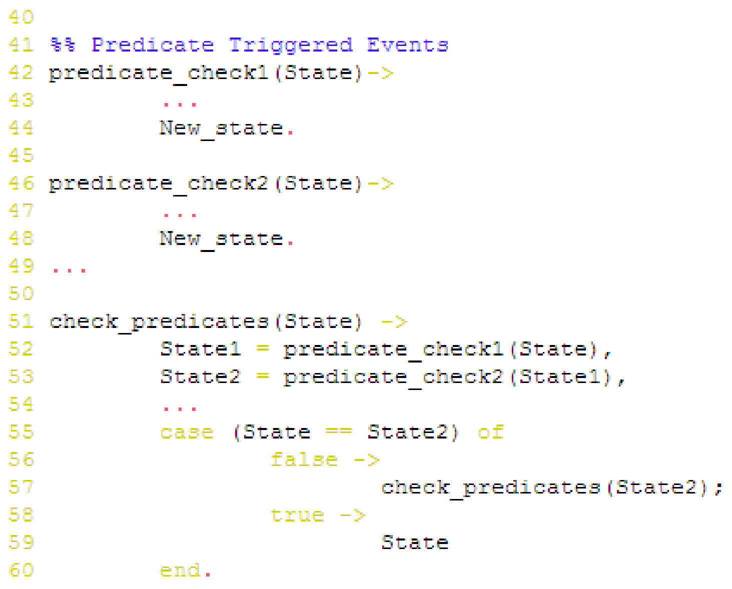


```
65 %% gen_server handle_casts callbacks
66 handle_cast( {event1, A, B} , State) ->
67         ...
68         New_state1 = check_predicates(New_state),
69         {no_reply, New_state1};
70
71 handle_cast( {event2, A, B, C} , State) ->
72         ...
73         New_state1 = check_predicates(New_state),
74         {no_reply, New_state1};
75
76 handle_cast( {crash, Who} , State) ->
77         ...
78         New_state1 = check_predicates(New_state),
79         {no_reply, New_state1};
```

Figure 3.11: **Template with code for all algorithms implemented (Part 3)**

But where are these *gen_server* casts handled? The *gen_server* will invoke a *handle_cast* callback whenever a cast can be handled. Listing 3.11 defines the *handle_cast* for this process. Note that the *handle_cast* function takes two arguments (lines 66, 71 and 76): the first one is the request which was *cast*ed to the server, whilst the second one is the *State* of the server. Note that this function uses function pattern matching to choose which of the functional clauses to invoke. Internally this handler will perform the algorithmic action required to handle the request. Notice that this function returns a tuple which holds the updated state. This state will be the one stored internally by the *gen_server* and which will be passed to the subsequent *handle_cast*.

In the *handle_cast* handlers, one can note that exactly before returning the new state, a call to *check_predicates* is made. This call is used as part of the implementation of *predicate triggered events* (refer 3.3.2). Since there is nothing like *predicate triggered events* in Erlang, a check is made exactly at the end of every *gen_server* cast handler.



```
40
41 %% Predicate Triggered Events
42 predicate_check1(State)->
43         ...
44         New_state.
45
46 predicate_check2(State)->
47         ...
48         New_state.
49 ...
50
51 check_predicates(State) ->
52         State1 = predicate_check1(State),
53         State2 = predicate_check2(State1),
54         ...
55         case (State == State2) of
56                 false ->
57                         check_predicates(State2);
58                 true ->
59                         State
60         end.
```

Figure 3.12: **Template with code for all algorithms implemented (Part 4)**

Listing 3.12, shows how *predicate triggered events* are implemented. The *check_predicates()* function (line 50 onwards), takes the *State* as a parameter. Here two predicate checks are made by calling *predicate_check1()* and *predicate_check2()* - these internally check whether a predicate is satisfied, and if so perform the associated action and return with the new state. Note that here, there is a little subtlety involved: it could be that after checking the predicates, an action is taken, however, at this point, the second predicate check action could have caused the first predicate to be satisfied once again. This requires that the checks should be made again until none of the predicates matches. This is handled on line 55, where the final state after checking all the predicates is compared with the initial state of the *check_predicates()* function. If there a change in the *State*, then this function recurses to recheck the predicates. Otherwise, the passed *State* is returned.

3.6.4 Working with Single Assignment

Being a functional language, Erlang has single assignment of variables. Essentially this means that every variable can only be assigned a value once. However, the algorithms presented in literature, never assume they are working with such a paradigm. However, as explained, every algorithm will be implemented with the concept of a module which maintains an internal state.
In order to be able to change part of the state, it is required to create a new variable with the new values, and copy the unchanged values from the previous state. This, however, does make the code longer and less readable, and for this reason, the internal code of every event handler in this dissertation, will not be presented using this single assignment.

3.7 Conclusion

The aim of this chapter was to gradually bridge the gap from theory to practice - and propose how various concepts were implemented in reality. It gave us the techniques by which all algorithms shall be described here and finally showed, to some extent, how these can be implemented in reality in Erlang.
The following three chapters will outline various algorithms for solving Reliable Broadcast, Consensus and Atomic Commit problems. These algorithms are the core of this project and this chapter presented the techniques which serve as a means to implementing these algorithms.

Chapter 4

Reliable Broadcast

4.1 Introduction

The first class of agreement algorithms studied in this project, are *Reliable Broadcast* algorithms. These algorithms deal with the transmission of messages which need to be delivered by all processes, including the sender itself. These algorithms study this problem, whilst keeping process failures into consideration. The algorithms presented here, assume the system model outlined so far.

In this chapter, various *Reliable Broadcast* specifications are presented, each having its own set of properties and characteristics. For each specification, an algorithm attempting to achieve these specifications, is outlined. For all algorithms, first the basic goal it attempts to achieve, is explained. Following this, the algorithms are explained in further detail, and any subtle intricacies are investigated. All of these algorithms are implemented in the framework presented in the previous chapter.

4.2 Interface

All reliable broadcast algorithms, follow this interface:

- **External Events Handled**
  - **init()**
    Initializes the algorithm.

  - **broadcast(Msg)**
    Sends *Msg* to all other processes.

- **Callbacks Expected**
  - **deliver(From, Msg)**
    Indicates that Msg was received from the process whose PID is *From*.

4.3 Best Effort Broadcast

4.3.1 Overview

The *Best Effort Broadcast* is the weakest reliable broadcast specification to be studied. It is a form of broadcast which does not provide any fault tolerance guarantees and is consistent as long as the sender does not crash whilst broadcasting. Implementations for this broadcast specification is typically found in non-fault tolerant code; however, this broadcast will be the basis for various stronger algorithms, which wrap logic around it to address its shortcomings.

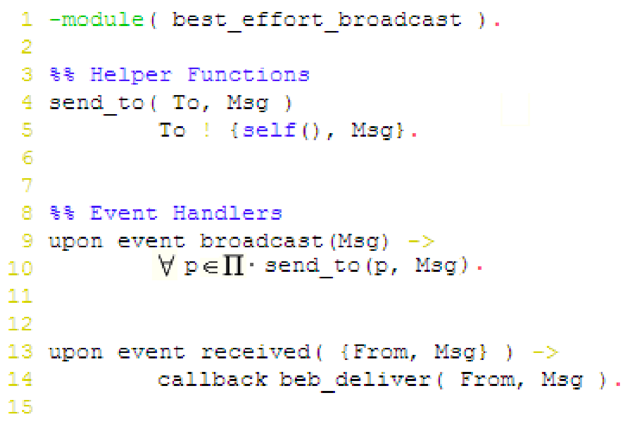
4.3.2 Specification

The Best Effort Broadcast abstraction is a form of *Reliable Broadcast* which only guarantees consistent delivery if and only if the sender remains alive throughout the whole of the sending process. A Best Effort Broadcast abstraction should satisfy the following properties:

- **Validity**: If a correct process p broadcasts message m, this eventually gets delivered by all correct processes in the system.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.

4.3.3 Algorithm

Figure 4.1 gives an algorithm which implements the *Best Effort Broadcast* specification. This algorithm is known as the *Basic Broadcast* algorithm. The algorithm consists of just two event handlers. The *broadcast* event



```
 1 -module( best_effort_broadcast ).
 2
 3 %% Helper Functions
 4 send_to( To, Msg )
 5         To ! {self(), Msg}.
 6
 7
 8 %% Event Handlers
 9 upon event broadcast(Msg) ->
10         ∀ p∈Π · send_to(p, Msg).
11
12
13 upon event received( {From, Msg} ) ->
14         callback beb_deliver( From, Msg ).
15
```

Figure 4.1: **Basic broadcast algorithm**

handler (line 9) is triggered by the user of this module, and it initiates a best effort broadcast. It iterates through all the processes in the system (denoted by Π on line 10) and sends them the passed message. Notice that a helper function *send_to()* is used, which sends the PID of the sender (returned by *self()* function on line 5) together with the message itself.

The *received* event (line 13), is triggered whenever a new message is received (as explained in section 3.6.1.1). This event indicates that some other process has broadcasted a message which is returned to the user by triggering a callback *beb_deliver*.

4.3.4 Erlang Implementation

Despite being very short, there are still some points which are worth outlining. Firstly, a list with all PID's (as required in line 10) can be returned by calling *node_utils:get_all_peers()* which returns a list with all the PID's of the *Main* process on each node. Having this list, it is worth noticing that the predicate on line 10, can be neatly implemented using *lists:foreach/2*. This function takes the list of all processes and the *send_to()* function, as a higher order function, which is almost identical to that on line 10.

4.3.4.1 Implementation Optimisations

There are not many performance considerations to be made for this simple algorithm; however, one simple tweak was to use a failure detector, and avoid wasting resources to send to processes which are known to be crashed.

4.3.5 Evaluation

The suggested algorithm meets the *Best Effort Broadcast* specifications due to the properties of links taken in consideration in this project (refer 3.2.3). The message broadcasted, should eventually reach the destination, and since this is sent to all processes, the *validity* property is satisfied. Moreover, the *integrity* property, follow directly from the *no creation* and *no duplication* properties of links (refer 3.2.3). The *Basic Broadcast* algorithm, however, only guarantees agreement if the sender is not faulty.

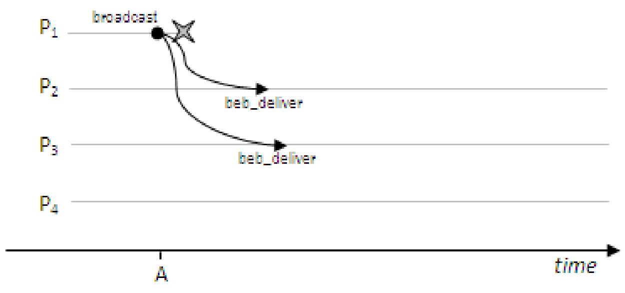


Figure 4.2: **Violation of agreement with the basic broadcast algorithm with a faulty sender**

Figure 4.2, gives a run of this algorithm. At point A, Process P_1 starts a broadcast, but crashes without managing to send messages to all processes. As can be seen, processes P_2 and $P3$, do receive and deliver the message, however, process P_4 does not. This could possibly leave the system in an inconsistent state.[14].

In terms of performance, in general the algorithm suggested requires $|\Pi|$ messages to be transferred over the network with every broadcast event.

4.4 Regular Reliable Broadcast

4.4.1 Overview

The *Regular Reliable Broadcast* protocol strengthens the specifications of the *Best-effort Broadcast* by taking into consideration process failures and taking the appropriate measure to ensure that the system still remains consistent. *Regular Reliable Broadcast* is resilient to sender crash failures: it ensures that either all processes get the message broadcasted, or that none at all gets the message.

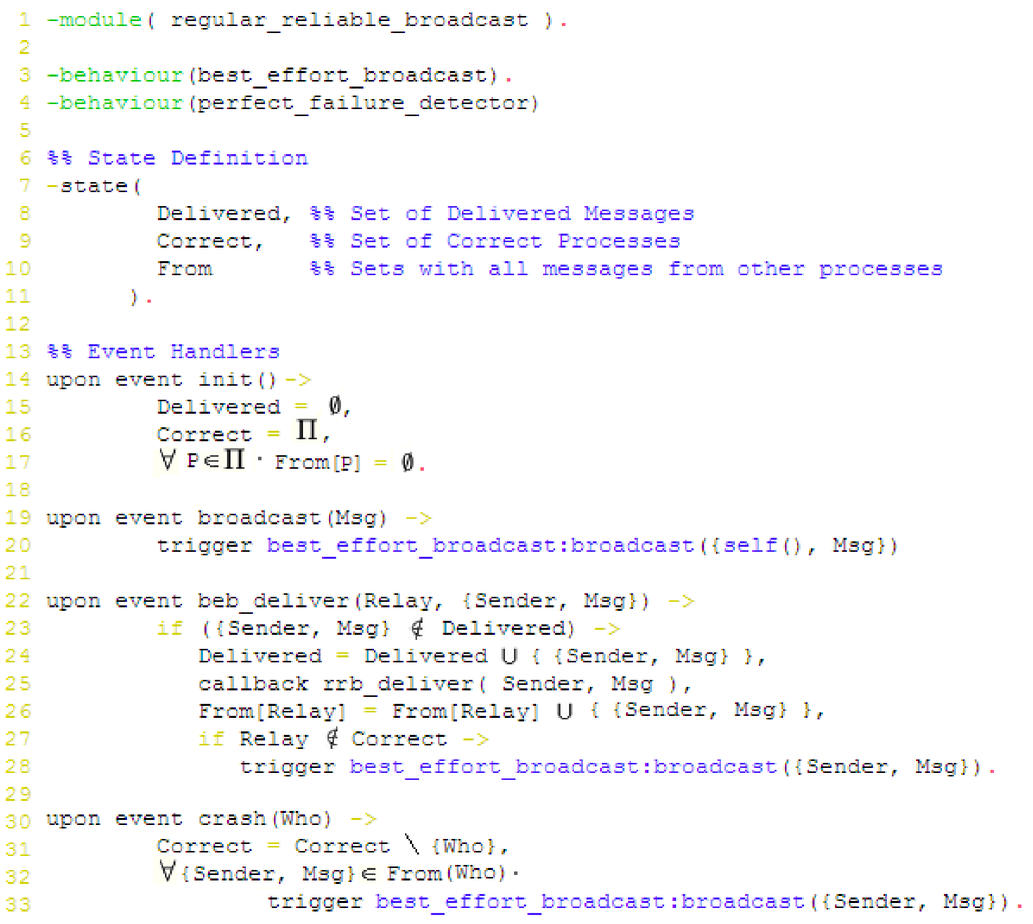
4.4.2 Specification

The *Regular Reliable Broadcast* guarantees that the all correct processes are consistent with respect to the broadcast abstraction. Note that this protocol does not, however, provide any guarantees on the state of the crashed processes. A *Regular Reliable Broadcast* protocol satisfies the same *validity* and *integrity* properties as the *Best Effort Broadcast* specification. In addition to these it also guarantees an *agreement* property, as follows:

- **Agreement**: All correct processes deliver the same set of broadcast messages.
- **Validity**: If a correct process p broadcasts message m, this eventually gets delivered by all correct processes in the system.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.

4.4.3 Algorithm

One particular implementation of the *Regular Reliable Broadcast* is the *Lazy Reliable Broadcast* algorithm, given in the listing in figure 4.3. This algorithm builds up on the *Best-Effort Broadcast* protocol to ensure that the *Agreement* property is satisfied. It utilizes a failure detector (line 4) to ensure that correct processes will *take over* when a sender crashes. The notion of the algorithm is that every correct receiver should broadcast the messages it received from some process which is detected to have crashed. In this sense, a process which receives a message from a faulty process becomes



```
-module( regular_reliable_broadcast ).

-behaviour(best_effort_broadcast).
-behaviour(perfect_failure_detector)

%% State Definition
-state(
        Delivered,  %% Set of Delivered Messages
        Correct,    %% Set of Correct Processes
        From        %% Sets with all messages from other processes
      ).

%% Event Handlers
upon event init()->
        Delivered = ∅,
        Correct = Π,
        ∀ P∈Π · From[P] = ∅.

upon event broadcast(Msg) ->
        trigger best_effort_broadcast:broadcast({self(), Msg})

upon event beb_deliver(Relay, {Sender, Msg}) ->
        if ({Sender, Msg} ∉ Delivered) ->
           Delivered = Delivered ∪ { {Sender, Msg} },
           callback rrb_deliver( Sender, Msg ),
           From[Relay] = From[Relay] ∪ { {Sender, Msg} },
           if Relay ∉ Correct ->
              trigger best_effort_broadcast:broadcast({Sender, Msg}).

upon event crash(Who) ->
        Correct = Correct \ {Who},
        ∀{Sender, Msg}∈ From(Who)·
                trigger best_effort_broadcast:broadcast({Sender, Msg}).
```

Figure 4.3: **Lazy reliable broadcast algorithm**

a *Relay* for all the previously sent messages by that process. Hence every process must keep track of the messages it received from all processes, in order to be able to relay these messages when their sender crashes. These are held in a map data structure, called *From*.

A basic run of this algorithm is given in the *process/time* diagram in figure 4.4.

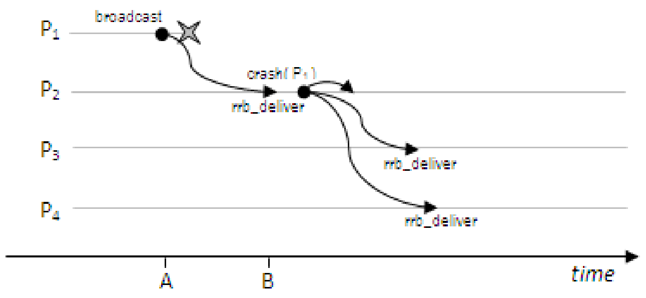


Figure 4.4: **A run of the lazy reliable broadcast algorithm with faulty sender**

In figure 4.4, process P_1 starts a *regular reliable* broadcast (Point A) and crashes shortly afterwards, in a way that the message is only actually sent to process P_2. Process P_2 receives and delivers the message (point B). At this stage, the system is in an inconsistent stage, because process P_2 delivered a message which processes P_3 and P_4 did note even receive. However, because of the failure detector, process P_2 knows or will eventually know that process P_1, the sender of the message, is dead. At this point, process P_2 will relay the message to the other processes in the system. In this run of the algorithm, only one process (P_2) was lucky enough to be sent the message before the death of the sender. If more processes were sent this message, more than one process would become a relay of the sender process. Since the job of the relay process is to retransmit the message to all other processes, having multiple relays would signify that the processes would receive the message multiple times. But due to the fact that this is the same message being received multiple times, the algorithm should only callback the *rrb_deliver()* event once.

In order to solve this problem, the *Delivered*, a set containing all the messages which have been delivered *Delivered*, is kept. Before triggering *rrb_deliver()*, a check is made to determine whether the message has already been delivered (is an element of *Delivered*). This event will be triggered only when the message has not already been delivered.

4.4.3.1 Dissecting the Algorithm

In the listing in figure 4.3, the state of the algorithm is given in lines 7 to 11. This state holds the set of correct processes, *Correct* and the set of messages which have been received and delivered, *Delivered.* It also contains a map between all processes and the messages received from each process. The initial state is defined in lines 15 to 17 as a handler to the *init()* event. Initially, the *Delivered* set is empty, the *Correct Set* holds all the processes, and all the sets in the *From* map are empty. Broadcasting a message is the same as the broadcast event of the *best effort broadcast* specification. This is shown in broadcast handler (lines 19 and 20), where a *best effort broadcast* event is triggered. It is important to note that the content of the message being broadcasted is a tuple with the following structure:

```
{ Sender, Message}
```

The *Sender* contains the PID of the initial sender: because the message may be *relayed* by other processes; however, these do not change this sender field. The *Message* contains the data passed with the *broadcast()* event.

When a *beb_deliver* event is *callback*ed (line 22), the message received is entered in the *Delivered* set and a *regular reliable broadcast* deliver (*rrb_deliver*) event is triggered (line 25). This can only happen if this message has not been previously delivered; that is if it is not a member of the *Delivered* set (line 23). The message is also added to the *From* map to the set of messages originating from the relay process.

So far no fault tolerant measures were taken. The remaining pieces of code deal with the fault tolerant characteristics of the algorithm. First, however, it is important to outline two scenarios which encapsulate the way failures are detected and handled. Consider a system with a sender process which broadcasts a message which crashes whilst broadcasting. There are two ways in which a receiver process, which has been sent the message, detects the failure:

1. The receiver process detects it has crashed **before** receiving the broadcasted message.

2. The receiver process detects it has crashed **after** receiving the broadcasted message.

These two cases stem from the fact that no assumption with regards to message ordering, is made. The algorithm being outlined here handles both scenarios.

The first case is handled by lines 27 to 28 where after delivering a message, the relay process of that message is checked to ensure that it still makes part of the *Correct* processes. If this is not so, then it might be that this process was one of the (lucky) processes who managed to get the message. Hence, this message should be broadcast (*best effort* broadcast) to the other processes - so as to ensure that all other processes managed to get the message despite that the sender crashed.

The second scenario is handled in the callback of the failure detector (ie the *crash()* event handler in lines 30 to 33). In line 32, all messages received from the crashed process, are relayed to the other processes. This set of messages is fetched from the *From* map.

All histories of failures can be classified as any of these two scenarios or of a third scenario in which the sender process does not live long enough to actually send the original message to any other process. The latter scenario would still leave the system in a consistent state and its effect the same as if the *regular reliable* broadcast event was not even triggered at all.

4.4.4 Erlang Implementation

The *Regular Reliable Broadcast* algorithm contains a number of elements which will reappear in subsequent algorithms and require some attention in their implementation. Particularly, one should notice that there are a number of sets being used. These were implemented in Erlang by using the *sets* module. This module provides routines to perform the common set access and comparison operations required.

One important consideration deals with having a process broadcasting the same message. If this happens, the algorithm given here would simply not trigger a *deliver* event except for the first broadcast. In order to go around this problem, a unique number is appended with the message. This way, if the user actually rebroadcasts the same message, this number will have a different value and the algorithm would not think it has already delivered this message. However, in the case when multiple relay processes broadcast the same message, this number is not altered and hence the receiver would determine that it has received a message which has already been delivered.

Another important consideration is the implementation of the map data structure, used for the *From* map. This is implemented through the use of the *dict* Erlang module. A *dict* provides a map (or dictionary) data structure in Erlang.

4.4.4.1 Implementation Optimisations

The main aspect of the algorithm which entails further investigation, is definitely the storage requirements of the algorithm. The *Delivered* and *From* structures continue to get populated with every message which gets delivered. It can be noted that, for the *Delivered* structure, it is not necessary to store the whole message; a hash of the message would suffice. In this project, the message is hashed with an md5 algorithm available in the Erlang *crypto* module. Nevertheless, this set still grows indefinitely and even worse, this techniques cannot be used with the *From* structure.

One attempt to solve this problem might be that of having each process which delivers a message, acknowledge one another. When a process detects that a message was delivered by all other processes, it can remove this message from these sets. It turns out, however, that this method is prone to network message reordering which would lead to inconsistencies in the system. In order to solve this issue, it would be necessary to study the use of timestamps for such system. This idea is suggested as part of the future work and is outlined in section 9.

4.4.5 Evaluation

The *Lazy Reliable Broadcast* Algorithm meets the *Regular Reliable Broadcast* properties. It satisfies the *integrity* and *validity* properties due to the underlying usage of the *Best Effort Broadcast.* It satisfies the *Agreement* property because once a sender crashes, if a receiver exists, its failure detector will eventually cause a *crash* event and the message will be sent to the other nodes. The *Regular Reliable Broadcast* provides fault tolerant properties which ensure a consistent state amongst all correct processes. It is also rather efficient in terms of network utilization. The best case, when no failures occur, it requires $|\Pi|$ message exchanges. Once a sender crashes, the message will be sent by all receivers, to the remaining processes which are alive, by all receivers. Hence, in the worst case, the number of messages exchanged is in the order of $|\Pi|^2$.

4.5 Uniform Reliable Broadcast

4.5.1 Overview

The *Regular Reliable Broadcast* protocol outlined in the previous section provides a fault tolerant broadcast mechanism which guarantees agreement amongst all correct processes. This, however, means that some process might deliver the message and crash, without sending the message to the other nodes. This can create problems if the delivery of such a message involves interaction with some outer system, leaving an inconsistent side effect. The *Uniform Reliable Broadcast* abstraction ensures that there is agreement on the delivered message between both correct and faulty processes.

4.5.2 Specification

The *Uniform Reliable Broadcast* protocol ensures that the set of messages delivered by faulty processes is subset or equal to the set of messages delivered by correct processes. This will result in changing the *agreement* property of the *reliable broadcast*, into a *uniform agreement* property. The *uniform reliable broadcast* specification guarantees the following properties:

- **Uniform Agreement**: Every message delivered by a process (correct or faulty), will be delivered by all correct processes.
- **Validity**: If a correct process process p broadcasts message m, this eventually gets delivered by all correct processes in the system.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.

4.5.3 Algorithm

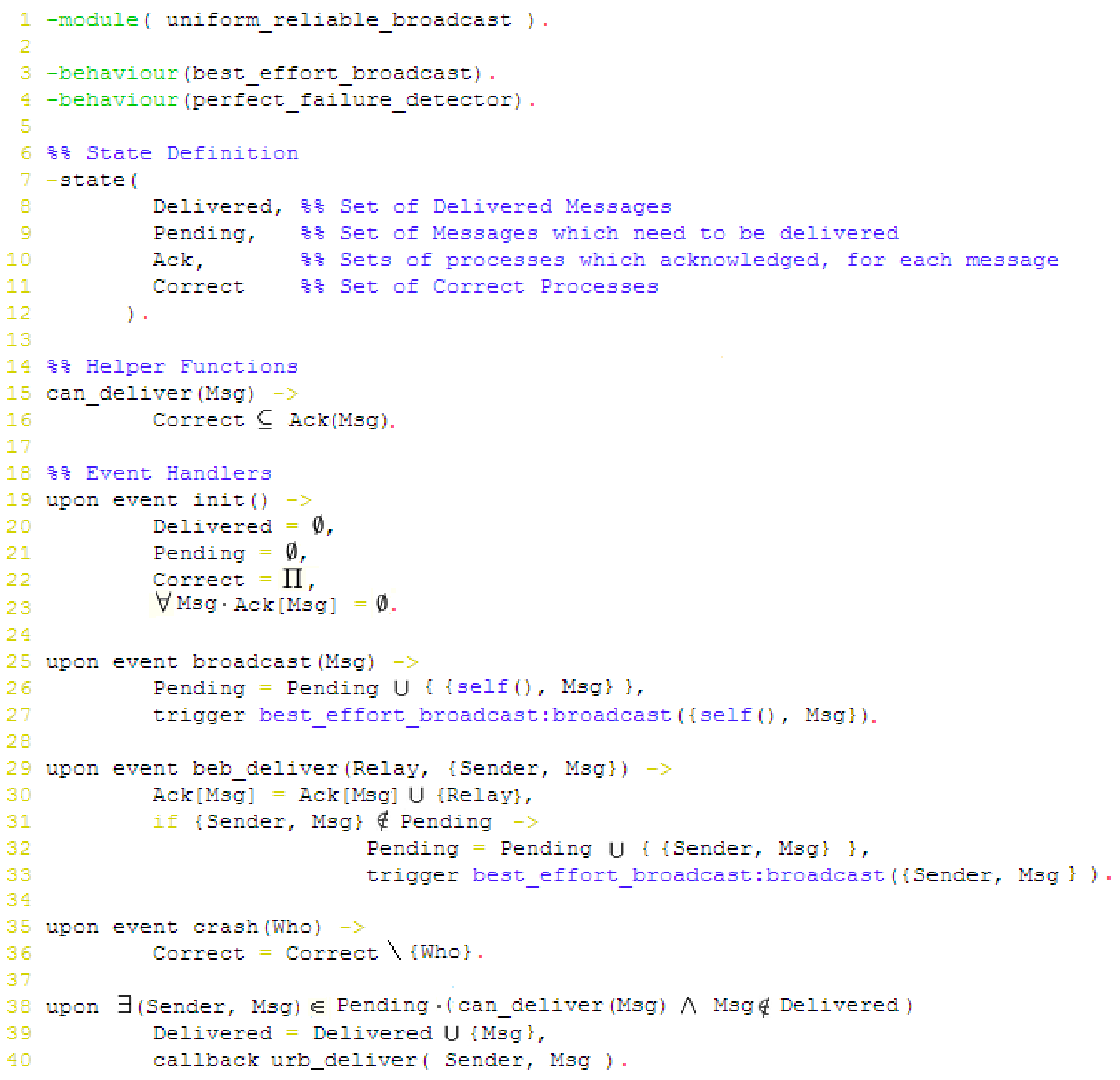
An algorithm to implement the *Uniform Reliable Broadcast* abstraction needs to ensure that all nodes have received the message before actually delivering the message. By analyzing the algorithm given for the *Regular Reliable Broadcast* (going back to figure 4.3), it can be noted that the uniform agreement is not guaranteed because the message is delivered (line 25) before checking whether this needs to be sent to the other nodes (lines 27-27). A naive patch may be to simply move the delivery of the message to be done after checking whether this needs to be sent to the other nodes - ie putting line 25 after line 28. However, this still is not correct as a process may crash before detecting that a sender process has crashed, but after that it delivers a message - possibly violating *Uniform Agreement.*

Another approach may be to always *relay* the messages immediately and then deliver the message after that all messages have been sent. There is still a subtle problem with such an approach: just because a message is sent, it does not mean that it will get delivered. Referring to section 3.2.3, it was noted that messages might be lost, and eventual delivery could only be guaranteed if the sender does not crash. Hence, in this case, if the *deliver* event is triggered after sending to all, and then the process crashed, it could be that some other process still does not receive the message broadcast. This would violate the *uniform agreement* property.

In order to solve this problem, a process should only deliver a message, when it is sure that all other processes received it.

Listing 4.5 presents the *All-Ack Uniform Reliable Broadcast* algorithm which is an implementation of the *Uniform Reliable Broadcast* protocol. The Algorithm ensures that the messages has reached all processes before actually delivering it. This is done by having each process immediately rebroadcast each message. Moreover, since every process will broadcast the same message, every process also keeps note of the processes from which it has received the message. Whenever a process, determines that it has received the message from all other correct processes, then it can deliver the message - since now the process knows that all other processes have received the message.

A basic run of this algorithm is given in figure 4.6. This run consists of three correct processes performing a *uniform reliable broadcast.* Process P_1 starts the broadcast by sending the message to the other participants. When P_2 and P_3 receive the message, they also broadcast the message to the other processes. In the run given here, process P_2 would have received the message from the rest of the processes at point marked A. At this stage, it can deliver



```
-module( uniform_reliable_broadcast ).

-behaviour(best_effort_broadcast).
-behaviour(perfect_failure_detector).

%% State Definition
-state(
        Delivered, %% Set of Delivered Messages
        Pending,   %% Set of Messages which need to be delivered
        Ack,       %% Sets of processes which acknowledged, for each message
        Correct    %% Set of Correct Processes
      ).

%% Helper Functions
can_deliver(Msg) ->
        Correct ⊆ Ack(Msg).

%% Event Handlers
upon event init() ->
        Delivered = ∅,
        Pending = ∅,
        Correct = Π,
        ∀Msg·Ack[Msg] = ∅.

upon event broadcast(Msg) ->
        Pending = Pending ∪ { {self(), Msg} },
        trigger best_effort_broadcast:broadcast({self(), Msg}).

upon event beb_deliver(Relay, {Sender, Msg}) ->
        Ack[Msg] = Ack[Msg] ∪ {Relay},
        if {Sender, Msg} ∉ Pending ->
                Pending = Pending ∪ { {Sender, Msg} },
                trigger best_effort_broadcast:broadcast({Sender, Msg } ).

upon event crash(Who) ->
        Correct = Correct \ {Who}.

upon ∃(Sender, Msg) ∈ Pending·(can_deliver(Msg) ∧ Msg∉ Delivered)
        Delivered = Delivered ∪ {Msg},
        callback urb_deliver( Sender, Msg ).
```

Figure 4.5: **Uniform Reliable Broadcast algorithm**

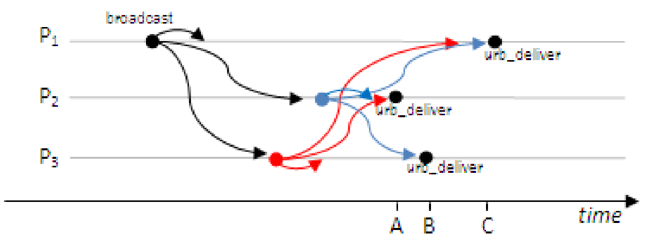


Figure 4.6: **A run of the all-ack algorithm for uniform reliable broadcast**

the message. Process P_3 receives the message from all nodes at point B, and the same happens for process P_1 at point A.

4.5.4 Dissecting the Algorithm

In the algorithm in figure 4.6 the state contains a *Delivered* set and a *Correct* set, whose purpose is the same as in the *regular reliable broadcast*. It also contains two other data items: *Pending* and *Ack*. *Pending* is a set containing messages which have not yet been received from all processes and hence cannot be delivered. *Ack* is a mapping between messages and a set with processes from which that message was received.

As hinted earlier, this algorithm delivers a message only when it has been received from all other processes. The *can_deliver()* helper function (lines 15 to 16) checks whether the *Correct* set is a subset or equal to the set of processes from whom this process have received a particular message. This checksly whether a message has been received from all correct processes.

It should be noted that this algorithm requires three distinct structures to hold messages: the *Pending* and *Delivered* sets and the *Ack* map. It is worth analysing whether each of these serves a purpose - yields some information which cannot be inferred from the other structures. The necessity of the *Ack* map is rather obvious: to determine the processes from which the message has not yet been received. The requirement for two different sets *Delivered* and *Pending*, is more subtle to analyze. However, it is important to note that the message will only be delivered when it is received from all process. When the message is received for the first time, it needs to be broadcast to all other processes. The *Pending* set is used to determine whether the message is being received for the first time or not (line 31). This information can also be acquired from the *Ack* set. However, the *Pending* set is also used in the predicate check in line 38, to test whether its elements can be delivered or not. Here, without this set, it would not be possible to determine which are the undelivered message which need to be tested.

On the other hand, the *Delivered* set is used to avoid having a process, deliver the same message more than once. The true necessity of this set can be seen in histories such as the run shown in figure 4.7. Here P_1 starts the broadcast. Process P_2 receives this message and rebroadcasts it, however it crashes shortly afterwards. Note that since here a totally asynchronous network is assumed, process P_3's failure detector can determine that process P_2 crashed, before actually receiving the message which was previously sent from P_2. In fact, at point A in figure 4.7, process P_3 delivers the message because it has been received from all correct processes. However, some time after, at point B the message from process P_2 is received, and unless the *Delivered* set exists, it would not be possible to determine that this message has already been delivered. The *Delivered* set hence, avoids the problem of erroneously redelivering the same message.

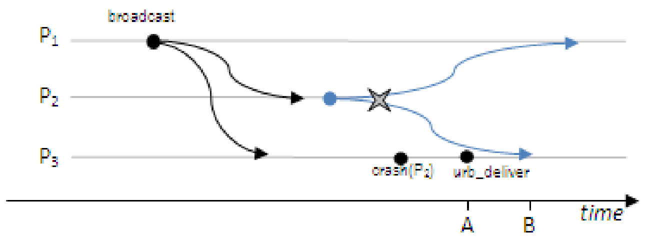


Figure 4.7: **Requirement of the *Deliver* set**

4.5.5 Erlang Implementation

The Erlang implementation of this algorithm, uses the same techniques as those presented in the algorithm of the *Regular Reliable* broadcast specification. The *sets* module is used to implement the *Delivered* and *Pending* sets. The *Ack* map is implemented using the Erlang *Dict* module. Note that in line 23 of listing 4.5, all sets in the *Ack* map are initialised to empty set. This initialisation is impossible to do in reality as it is impossible to initialize the map for all possible messages. However, in practice, if the entry in the *Ack* map being accessed is not found, then it is taken as if it contains an empty set.

4.5.5.1 Implementation Optimisations

As described in [14] and as hinted earlier, the storage requirements for this algorithm can be greatly reduced. The algorithm given thus far does not eliminate any messages from any of the *Ack*, *Pending* or *Delivered* structures. However, it can be noted that once a message can be delivered, then it can be removed from the *Pending* set. Moreover its associated set from the *Ack* map can also be freed. This tweak will prevent the *Pending* and *Ack* sets from growing indefinitely.

However, the message cannot be removed from the *Delivered* set. The necessity of the *Delivered* set was explained in section 4.5.4. Given that now messages in the *Pending* and *Ack* will be removed immediately as all correct processes acknowledge, then identical messages can only be received if a node crashed (one particular scenario is the run in figure 4.7). But problems can be easily prevented if messages are only allowed from correct processes. Hence, when a message is received, first the state of the sender is checked (in the *Correct* set), and the message will be "processed" only if it comes from a correct process; otherwise it will be dropped. This would now curb the need to keep a *Delivered* set, whilst still preventing messages to be delivered multiple times.

4.5.6 Evaluation

The algorithm given here meets the *Uniform Reliable Broadcast* specification. The *integrity* and *validity* properties are satisfied since these are provided by the *Best Effort Broadcast.* It satisfies the *Uniform Agreement* property because every process only delivers the message once it receives it from all correct processes - which means that there cannot be faulty processes which deliver messages which are not delivered by correct processes.

In the best case (when there are no failures), the algorithm takes two communication steps (one to send to all and one to receive from all), and exchanges $|\Pi|^2$ messages. The worst case occurs when all senders crash in sequence, each managing to send the message to just one participant. This would require $|\Pi| + 1$ steps.

4.6 Causal Order Broadcast

In all the broadcast specifications outlined so far, the focus was on the guaranteeing properties within a single broadcast operation and not on its possible interaction with other broadcasts already taking place. In particular, due to the asynchrony of the network, a process could perform two consecutive broadcasts with the receivers delivering the message from the second broadcast before delivering the one preceding it. In certain systems, such reordering may cause inconsistencies and hence must be prevented. The reason is because such a delivery is said to break the *causal* order of events - the second broadcast, may be an effect of the first one, however the delivery of the messages violate such an ordering.

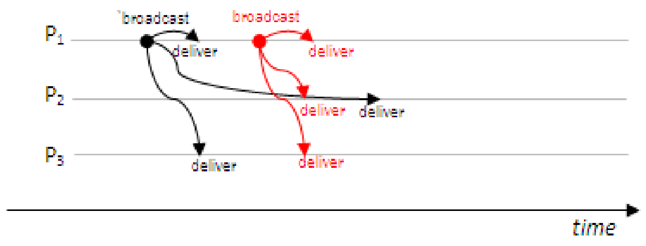


Figure 4.8: **An example illustrating violation of causal ordering with regular reliable broadcast**

Figure 4.8 gives illustrates such a scenario (with *regular reliable broadcast*) where the causal ordering is broken. Process P_1 issues two consecutive reliable broadcasts but process P_2 delivers the message of the first broadcast before the first. If say, the two broadcasts are associated with an operation which is non commutative, process P_2 will be in a different state than the other two processes.

The example illustrated, is just one case of causal ordering. In fact, the causal order relation $m_1 \rightarrow m_2$, indicating that message m_1 could have caused the submission of m_2 is valid in any of these cases [14]:

1. m_1 and m_2 were broadcast by the same process with m_1 being broadcast before m_2 (refer figure 4.9a)

2. m_1 was broadcast by process P_i and m_2 by process P_j after the delivery of m_1 (refer figure 4.9b)

3. $\exists m'$ such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$ (refer figure 4.9c)

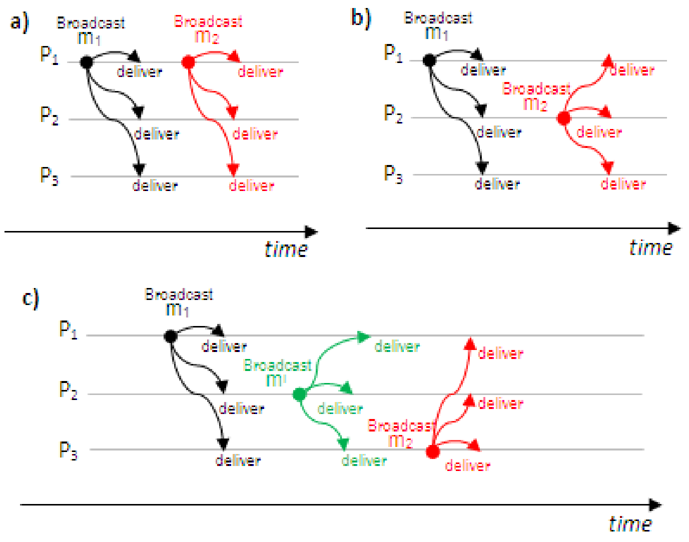


Figure 4.9: **Examples of Causal Ordering**

The *causal order broadcast* will ensure that the delivery of messages conforms with their causal order relations. Two messages which are not causally related, are said to be concurrent. The *causal order* broadcast does not define any delivery constraints on concurrent messages.

4.6.1 Specification

The *causal order broadcast* specification studied here guarantees the same properties as *regular reliable broadcast*, together with a *Causal Delivery* property. Its properties are:

- **Causal Delivery**: No process delivers a message m_j, unless every other message m_i, such that $m_1 - > m_j$, is delivered.

- **Agreement**: All correct processes deliver the same set of broadcast messages.

- **Validity**: If a correct process p broadcasts message m, this eventually gets delivered by all correct processes in the system.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.

4.6.2 Algorithm

```
-module(causal_order_broadcast).

-behaviour(regular_reliable_broadcast).

%% State Definition
-state(
        Delivered,   %% Set of messages which were delivered
        Past,        %% Set of pairs of delivered messages and sender PID
      ).

%% Event Handlers
upon event init() ->
        Delivered = ∅,
        Past = ∅.

upon event broadcast(Msg) ->
        trigger regular_reliable_broadcast:broadcast({data, Past, Msg}),
        Past = Past ∪ { {self(), Msg} }.

upon event rrb_deliver(From, {data, P, Msg, C }) ->
        if Msg ∉ Delivered ->
           ∀{Sender, M}∈ P·
               if M ∉ Delivered ->
                   trigger co_deliver(Sender, M),
                   Delivered = Delivered ∪ {M},
                   Past = Past ∪ { {Sender, M} }
           trigger co_deliver(From, M),
           Delivered = Delivered ∪ {Msg},
           Past = Past ∪ {{From, Msg}}.
```

Figure 4.10: **The causal order broadcast algorithm**

The listing in figure 4.10, gives the *no-waiting algorithm* for *causal order broadcast* specification. The idea behind the algorithm is very simple. Every process which needs to broadcast, always broadcasts a set of all messages already broadcast and delivered, together with the message to be broadcast. When a receiver receives the message and the set, it first checks whether any of the messages in the set are undelivered, if so it first delivers them. Finally, it delivers the message itself.

Internally the algorithm stores two sets *Delivered* and *Past*. The *Delivered* set is used to store all messages which have been delivered. The *Past* set holds pairs of messages and sender PID's.

When a message is to be broadcast, the *Past* set is reliably broadcast, together with the message as a single tuple (line 17). When this message is delivered by the receivers, first these check whether the message has already been delivered (line 21 onwards). Following this, all messages inside the received *Past* set are checked with the messages which were delivered. Any messages not yet delivered will be delivered immediately (line 24). This is why in line 21, all messages are checked to determine whether they had been previously delivered or not. Finally, the message itself is also delivered (line 27). When a message is delivered, it is included in the *Past* message and *Delivered* set.

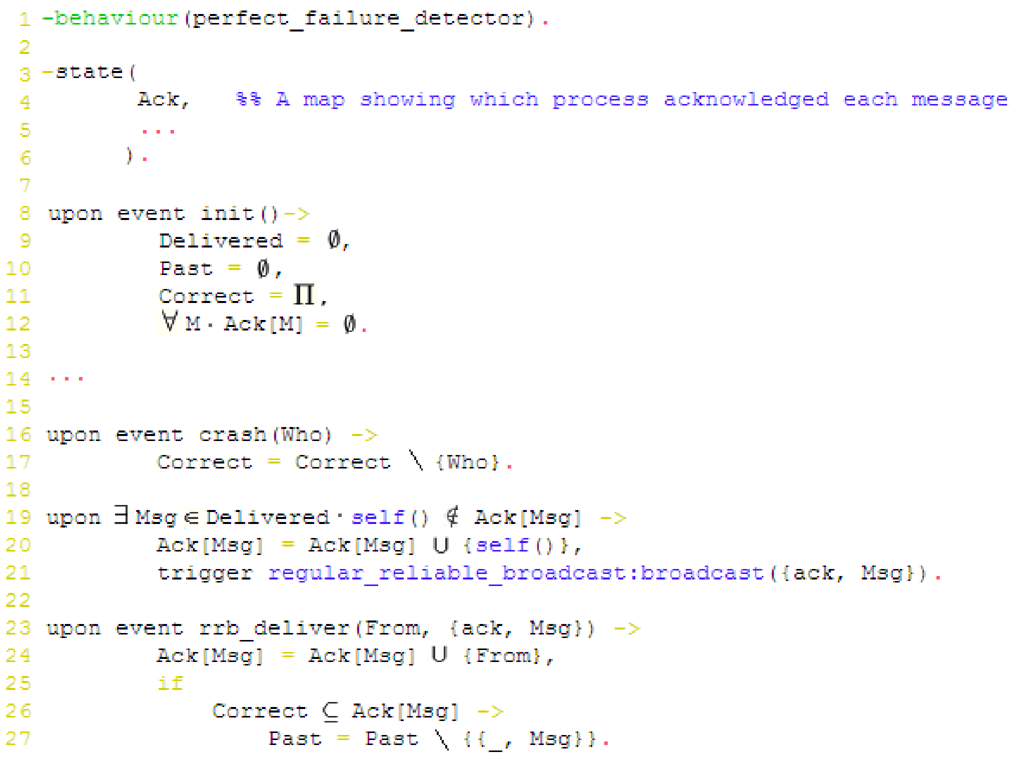
One should note, that in line 22, messages are taken out of the *Past* set and delivered. The order in which these messages are delivered is crucial. Messages should be delivered in the same order in which they were added to the *Past* set by the sender.

Note also, despite that a broadcast also sends the message to the sender process itself, the algorithm immediately adds an entry into the *Past* set for that message. This is done because in the time window between the submission and delivery (by the same process), the process may submit another message, and hence the previously sent message needs to be present in the *Past* set.

4.6.2.1 Pruning the Past set

The main concern with the algorithm is the indefinite nature of the *Past* set. Furthermore, this set is transferred on the network, with every broadcast - posing an ever growing message size as time progresses. This problem, presses to devise means to limit the size of this set.

The listing in figure 4.11, gives a method to "garbage collect" the *Past* set. The idea is to have every process to acknowledge to all other processes as soon as it delivers a message. When acknowledgements for a message, are received from all other correct processes, that particular message can be removed from the *Past* set. This way, the size of the *Past* is no longer indefinite, but rather decreases as all processes acknowledge delivery.



```
-behaviour(perfect_failure_detector).

-state(
       Ack,   %% A map showing which process acknowledged each message
       ...
      ).

upon event init()->
        Delivered = ∅,
        Past = ∅,
        Correct = Π,
        ∀ M · Ack[M] = ∅.

...

upon event crash(Who) ->
        Correct = Correct ∖ {Who}.

upon ∃ Msg ∈ Delivered · self() ∉ Ack[Msg] ->
        Ack[Msg] = Ack[Msg] ∪ {self()},
        trigger regular_reliable_broadcast:broadcast({ack, Msg}).

upon event rrb_deliver(From, {ack, Msg}) ->
        Ack[Msg] = Ack[Msg] ∪ {From},
        if
            Correct ⊆ Ack[Msg] ->
                Past = Past ∖ {{_, Msg}}.
```

Figure 4.11: **Garbage collection algorithm for the Past set**

This mechanism keeps a mapping between every message and the processes which acknowledged it (*Ack*). As soon as a message is delivered, but not acknowledged, an acknowledgement is reliably broadcast to all processes (lines 19-21). When this message is received, its sender is recorded in the *Ack* map for that message, and if all correct processes have acknowledged that message, then it is removed from the *Past* set (lines 23-27).

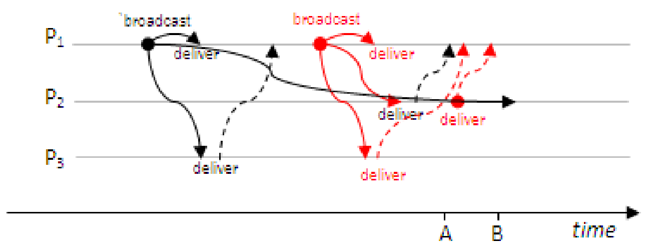


Figure 4.12: **Run of the causal order broadcast algorithm**

Figure 4.12 gives a run of the causal order algorithm. Basically, it re-illustrates the same example given earlier in this chapter, but using the *causal order* broadcast. The dashed lines represent acknowledges from processes. For clarity, only the acknowledges to process P_1 are shown. Process P_1 broadcasts two messages. Note that the first message arrives at process P_2 after the second message. However, since the second message's past set, contains the first message, process P_2 first delivers the first message and then the second message. At point A, P_1 receives acknowledgments for the first message from all other participants, and hence can remove the first message from its *Past* set. At point B, all acknowledges for the second messages, are received by P_1, and hence it can remove this message from its *Past* set.

4.6.3 Erlang Implementation

The *no-waiting* algorithm for *causal order* broadcast was implemented in Erlang using the framework outlined in this project. The distributed garbage collection scheme for the *Past* set was also implemented. The *sets* module was used to implement the *Delivered* and *Past* sets. The *Ack* map was implemented with the *Dict* module.

It was mentioned that messages should be delivered from the *Past* set in the same order in which they were added. This was achieved by having every process keeping a counter of the messages entered in its own *Past* set. Moreover, when a message is to be added to this set, the counter is added together with the message as a single tuple. This set is sent when a *causal order* broadcast is made. The receiver will then convert the received *Past* set to a list, and sort messages in the list in ascending order, according to this number. Finally, messages are recursively read from the list, and the original message is delivered.

One other thing to note is that the algorithm outlined here, does not allow identical messages to be broadcast from the same process. However note, that due to the technique just outline for ensuring the ordered delivery of messages in the *Past* set, no two messages broadcast from the same process can be identical. This is because, in the least, the count for the last message, will be greater than any of the previous messages.

Finally, note that the initialisation of the *Ack* map, requires that an empty set will be mapped with every possible message (figure 4.10, line 12). However, in practice, since a *Dict* is used, initially every message will be mapped with nothing (not empty set). Hence, when an entry for a message is to be entered for the first time (by checking whether it is currently mapped with nothing), first the empty set is mapped.

4.6.4 Implementation Optimisations

As noted, the *Delivered* set will hold a copy of all message delivered. As done in the implementation of previous algorithms, a hash of the message is kept, instead of the message itself. This relieves some of the memory requirement of the algorithm. Moreover, as soon as a message can be removed from the *Past* set, its equivalent set in the *Ack* set is removed as well.

The problem of the *Past* set has been tackled with the use of the garbage collection mechanism outlined. However, the *Delivered* set, still has indefinite memory requirements. A tentative solution to this problem is suggested for future work (see section 9).

4.6.5 Evaluation

The *causal order* broadcast guarantees that every message delivered obeys the causal ordering relation, apart from guaranteeing the same properties as *regular reliable* broadcast. The *agreement*, *validity* and *integrity* properties are guaranteed because of the underlying use of a *regular reliable* broadcast abstraction. The *causal delivery* property is guaranteed thanks to the *Past* set, which enables any causally related message which has not yet been delivered, to be delivered.

In terms of performance evaluation, the basic algorithm exchanges the same amount of messages as the *regular reliable* broadcast algorithm - ie has a best case of $|\Pi|$ (in a failure free run) and a worst case of $|\Pi|^2$ message exchanges. However, with the garbage collection scheme $|\Pi|^2$ message acknowledges need to be exchanged for every message broadcast.

4.7 Conclusion

In this chapter, a number of *Reliable Broadcast* specifications were investigated. First, the *best effort Broadcast* was outlined. This is the weakest broadcast abstraction and only guarantees correctness if the sender does not crash whilst broadcasting. Then the *regular reliable broadcast* was studied. This specification guarantees that all correct processes agree on the set of delivered messages. Following this, the *uniform reliable broadcast* was investigated. This broadcast abstraction guarantees that no faulty process delivers a message which a correct process does not. Finally, the *causal order broadcast* was outlined. This reliable broadcast specification guarantees that all messages delivered, obey the causal order relation.

Established algorithms for each specification, were studied and implemented in Erlang, using the implementation framework developed for this project. Insights about the implementation, together with tweaks and optimisations taken, were also outlined.

Chapter 5

Consensus

5.1 Introduction

In this chapter, another class of agreement algorithms will be investigated: Consensus. Consensus deals with having a system of processes which attempt to decide on using a value, from various proposals. Each process may propose a value, however, it is up to the consensus algorithm to decide on which value all the processes should choose.

Unless proper care is taken, situations may arise which cause processes to choose different values. In the light of process failures and network asynchrony, algorithms need to be studied to tackle the consensus problem reliably.

In this chapter, various consensus specifications will be outlined. Established algorithms for these specifications will investigated, and finally also implemented in Erlang.

5.2 Interface

Below is the generic interface for all consensus algorithms implemented in this chapter:

- **External Events Handled**
  - **init()**
    Initializes the algorithm.

  - **propose(Value)**
    Assigns *Value* as the proposal of the calling process.

- **Callbacks Expected**

  - **decide(Decision)**
    Indicates that the consensus algorithm terminated and decide with value *Decision*

  Note that certain algorithms require a *Decision Function* as a parameter to *init*. This *decision function* is used to choose a value from a set of proposals. In order to guarantee agreement, this decision function should be identical for all processes.

5.3 A Slight change

It might be noted, that the standard interface outlined, does not give way to two or more instances of consensus to be in execution. This is because, there is no way by which to distinguish, whether a *propose* is intended for one consensus instance or the other. For this reason, in practice, all calls and callbacks to consensus have another argument - the consensus identifier. This argument will always come first in the argument list.

Hence if a process needs to propose to a three different instances of consensus, it should simply trigger propose three times, each time with a different consensus identifier. In the implementation framework, the *Main* process takes care to check whether a consensus *gen_server* for the particular identifier is started. If it is not started, it will start another instance for that consensus identifier. From this point on, all calls with that particular consensus identifier, are directed to the existing consensus *gen_server* process. This simple scheme makes it possible to have multiple instances of consensus running at the same time.

5.4 Regular Consensus

5.4.1 Overview

The *Regular Consensus* specification provides the basic mechanism for distributed process agreement. Processes propose different values and one of these values is globally chosen (decided) and returned to all processes.

5.4.2 Specification

The regular consensus specification ensures the same properties as a *regular reliable broadcast*, but to a consensus scenario. The properties which are guaranteed by *regular consensus* are:

- **Termination**: There exists a time at which all correct processes decide some value.
- **Validity**: A decided value must be a proposed value.
- **Integrity**: Processes decide only once.
- **Agreement**: All correct processes agree on the same value.

5.4.3 Flooding Algorithm

In this project an asynchronous message flooding algorithm, proposed in [14], was implemented. This algorithm is based on the simpler *floodset* consensus solving algorithm. The *floodset* algorithm assumes a synchronous communication model, hence its correctness is not guaranteed in an asynchronous system [1]. Moreover, the *floodset* algorithm assumes an upperbound on the number of failures and does not utilize a failure detector. Despite all these problems, the *floodset* algorithm still sets the basis for the *flooding consensus* algorithm implemented here. For this reason, it will be briefly outlined. Following this, the changes done to achieve an asynchronous algorithm will also be explained.

5.4.3.1 Simple algorithms based on Reliable Broadcast Protocols

In the previous chapters, various reliable broadcast protocols were outlined. At this point, one might ask whether a reliable broadcast algorithm can be used to implement the *Regular* consensus specification. It may appear that a simple solution can be provided by having processes which broadcast their proposals using some fault tolerant broadcast such as *regular reliable*

[1]Due to the forcing failure detector used here, in this case this would not be a problem. However, in general, synchronous algorithms cannot be used in asynchronous systems

broadcast. This broadcast ensures that all correct processes will see the same broadcast messages. Hence, if all processes simply use this protocol and broadcast their proposal to all, it is ensured that all processes will eventually get an identical set of proposals (and hence can decide on a value based on the same set of proposals), even when there are process failures. However, there is one subtlety which makes the use of such protocols inappropriate. This is because if a process crashes without managing to actually send its proposals to anyone, the other processes would not be able to tell whether a message from this process is underway or not. Despite that, the failure detector of these processes should eventually signal that this process has crashed, this would still not break the tie. Given that a message could actually be on its way on the network, the waiting processes would not know whether these can proceed or wait to receive something from the crashed process.

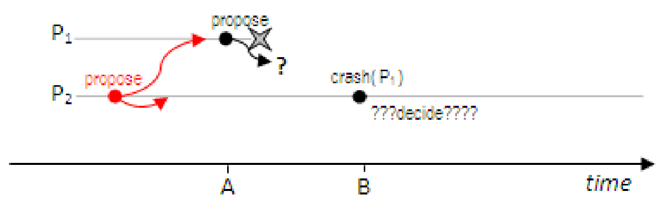


Figure 5.1: **Invalidity of Reliable Broadcast algorithms for implementation of consensus protocols**

This problem is illustrated in figure 5.1. Here the most basic case, with just two processes is considered. The two processes here are assumed to perform *regular* consensus using a reliable broadcast abstraction as explained above. Process P_1 is faulty and crashes at point A. However, from process P_2 point of view, it cannot determine whether this process has sent its proposal or not. Even though at point B, P_2's failure detector signals that process P_1 has crashed, it can make no assumption as to whether process P_1 has sent its proposals or not, and hence will not be able to decide.

At this point, one might attempt to "patch" this problem by ignoring messages received from crashed processes, however this would break the algorithm so badly, that it gives way for inconsistent decisions to be taken. This is because it might happen that a process detects a crashed process and ignores its future message, whereas another process might first receive the messages and then detect the node to have failed. Of course, since the first process has ignored messages the second did not, the processes will have different proposal sets on which to decide. An example of this run is shown in figure 5.2.

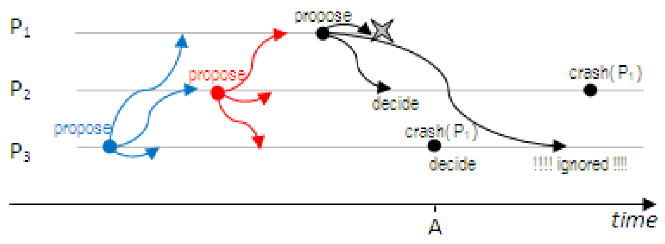


Figure 5.2: **Invalidity of implementing consensus algorithms using Reliable Broadcast Algorithms, even when ignoring messages from crashed processes**

In figure 5.2, three processes propose values by broadcasting their messages. Process P_1 fails shortly after finishing its broadcast. Process P_2 receives P_1's message, before actually detecting that this has failed. At this point it can decide because it has received all proposals. On the other hand, process P_3, detects that that P_1 failed (Point A) before receiving its proposal. Moreover, if this proposal will be received, it will be ignored, and hence P_3 can make a decision based on its current proposal set. Given that it has not seen proposals from process P_1, its proposal set will be different than that of P_2 and hence, it could decide differently.

For this reason, the reliable broadcast algorithms cannot be used directly to build consensus algorithms, but we will need to look at alternative algorithms to solve this problem.

5.4.3.2 The Floodset Algorithm

The idea behind the *floodset* algorithm is to ensure that all processes assemble an identical set of proposals and decide on that set. The algorithm consists of rounds. In each round all processes broadcast (*best-effort broadcast*), their proposal sets. The *floodset* algorithm also assumes upper bound

f on the number of process failures.

The number of rounds required by the floodset algorithm depends on f. If no failures occur, then just one round would be enough; in which every process waits for the proposal sets of the other processes, and then decides some value. If failures occur, these could cause processes to only broadcast their message to a subset of the participants. Such a situation will cause processes to have different proposal sets, and hence may possibly decide differently. The *floodset* algorithm overcomes this problem by performing $f + 1$ rounds. This way, at least one round is ensured not to contain any failures, and hence all proposal sets will be identical from this point on.

Listing 5.3 gives an adaptation of the floodset algorithm (given by Lynch in [19]). Note that the initialisation of the *floodset* algorithm requires two parameters: *Max_failures* and *Func*. *Max_failures* will initialize the maximum number of failures (referred to as f). *Func* is the decision function which will be applied to set of proposals, when all proposals have been gathered. These two parameters are held in the state, together with other data elements: *Proposals* set with the proposed values from the other processes, the current *Round* number and *Correct_this_round* set containing the PID's of processes from whom a proposal was received in the current round.

The algorithm here works by gathering proposals which are broadcast using the *best_effort_broadcast()* . The proposal sets received are stored together with the currently seen proposals in the handler of the *best effort broadcast* deliver event (line 27). Every time a Proposal set is received, its owner is marked as being correct by storing its PID in the *Correct_this_round* set (line 28). When this set is filled with all the processes (ie it is equal to Π), or the synchronous time limit for receiving messages has been exceeded (line 30), then the current round is over. As a reminder, note that the *floodset* algorithm is devised as a solution for consensus in synchronous systems, and hence there is a well defined deadline for receiving messages. Here, it is assumed that once this limit is exceeded, the *timedout receiving* proposition (line 30) will evaluate to true. At the end of the round, it should be checked whether this is the last round, (round $f + 1$), in which case the decision function should be applied to the proposal set, and the *decide()* event is callbacked (lines 33-34). If this is not the final round, then the *Correct_this_round* set should be emptied, *Round* number incremented and *Proposals* set broadcast.

```
-module( floodset).

-behaviour(best_effort_broadcast).

%% State Definition
-state(
        F,                      %% Maximum number of failures
        Proposals,              %% Set with all proposed values
        Round,                  %% Current Round Number
        Correct_this_round,     %% Set with Pids of senders for current round
        Func                    %% Decision Function
      ).

%% Event Handlers
upon event init(Max_failures, Function) ->
        F = Max_failures,
        Proposals = ∅,
        Round  = 1,
        Correct_this_round = ∅,
        Func = Function.

upon event propose( V ) ->
        Proposals = Proposals ∪ {V},
        trigger best_effort_broadcast:broadcast( Proposals ).

upon event beb_deliver(From, Proposals_rcvd) ->
        Proposals = Proposals ∪ Proposals_rcvd,
        Correct_this_round = Correct_this_round ∪ {From}.

upon ( Correct_this_round == Π ) ∨ (timedout receiving) ->
        if
            Round == F + 1 ->
                Decided = Func(Proposals),
                callback decide( Decided );
            Else ->
                Correct_this_round = ∅,
                Round = Round + 1,
                trigger best_effort_broadcast:broadcast( Proposals )
        end.
```

Figure 5.3: **The Floodset Algorithm**

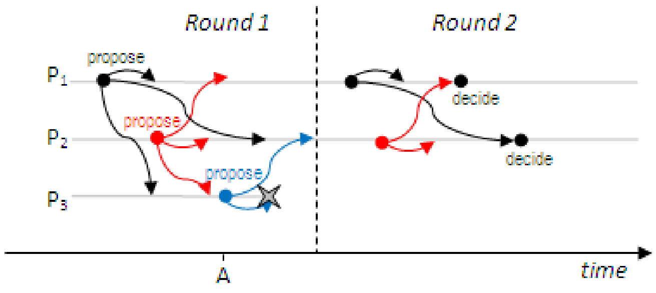


Figure 5.4: **A run of the floodset algorithm with one process failure**

Figure 5.4 gives a run of the floodset algorithm between three processes and with $f = 1$. Processes P_1 and P_2 broadcast successfully, however, process P_3 only sends its proposals to P_2 (Point A). Hence at the end of round 1, the processes do not have identical proposal sets. In round 2, which is the final round, P_3's proposal set is sent to P_1, by P_2 and hence all proposal sets are identical and the decision function can be applied.

The *floodset* algorithm solves consensus for any number of crash failures in a synchronous system. In particular, if $f = |\Pi| - 1$ then this algorithm will solve consensus for all possible failures in the system. However, its reliance on synchronous deadlines makes it unsuitable for usage in an asynchronous system. Moreover the lack of failure detection results in lots of message exchanges and communication steps (rounds). The following section will describe an algorithm, based on the original *floodset* algorithm, which tackles these problems and implements the *regular consensus* specification.

5.4.3.3 Algorithm

This section outlines the *flooding* algorithm for *Regular* consensus. This algorithm is based on the synchronous *floodset* algorithm, but addresses its shortcomings for asynchronous systems. This algorithm is also free from the inconsistency issues discussed when investigating the possibility of using *reliable broadcast* algorithms to implement consensus protocols (see section 5.4.3.1).

Listing 5.5 gives the *flooding* algorithm for *regular* consensus. This algorithm reduces the number of rounds required over the number of rounds required by the *floodset* algorithm. This is done by using an array *Cor-*

```
-module( flooding_consensus).

-behaviour(best_effort_broadcast).
-behaviour(perfect_failure_detector).

%% State Definition
-state(
        Correct,                %% Set of Correct Processes
        Decided,                %% Decision result
        Proposals,              %% Set with all proposed values
        Round,                  %% Current Round Number
        Correct_this_round,     %% An array with PID's of sender for all rounds
        Func                    %% Decision Function
      ).

%% Event Handlers

upon event init(Function) ->
        Correct = Π,
        Correct_this_round[0] = Π,
        Decided = ⊥,
        Round  = 1,
        Func = Function,
        ∀ I ∈ [1,|Π|] · Correct_this_round[I] = ∅,
                        Proposals[I] = ∅.


upon event propose( V ) ->
        Proposals[1] = Proposals[1] ∪ {V},
        trigger best_effort_broadcast:broadcast( {proposal_set, 1, Proposals[1]} ).

upon event beb_deliver(From, {proposal_set, R, Proposals_rcvd} ) ->
        Proposals[R] = Proposals[R] ∪ Proposals_rcvd,
        Correct_this_round[R] = Correct_this_round[R] ∪ {From}.

upon (Correct ⊆ Correct_this_round[Round] ∧ (Decided == ⊥ ) ->
        if
            Correct_this_round[Round] == Correct_this_round[Round-1] ->
                Decided = Func(Proposals[Round]),
                callback rc_decide( Decided ),
                trigger best_effort_broadcast:broadcast( {decided, Decided});

            Else ->
                Round = Round + 1,
                trigger best_effort_broadcast:broadcast( { proposal set, Round,
                                                   Proposals[Round - 1]} )
        end.

upon event beb_deliver(From, {decided, Value}) ∧ (From ∈ Correct) ∧ (Decided == ⊥) ->
        Decided = Value,
        callback rc_decide( Decided ),
        trigger best_effort_broadcast:broadcast( {decided, Decided} ).

upon event crash(Who) ->
        Correct = Correct \ {Who}.
```

Figure 5.5: **The Flooding Consensus Algorithm**

rect_this_round with indexes being the round numbers and the elements being sets with PID's of the processes whose proposal has been received. The *Proposals* set is also an array with the round numbers as indexes and elements being the proposals received for the corresponding round. When a value is proposed (lines 28 to 30), the value is added to the first *Proposals* set and broadcast. Note that when broadcasting (line 30), the message consists of three elements: an atom identifying the message type (*proposal_set*), the Round Number and the set of proposals. When this message is received (*beb_deliver* handler lines 32-34), the received proposals are inserted in the corresponding *Proposals* set for the received round number. The PID of the sender are also inserted in the *Correct_this_round* set for the received round number.

A round ends whenever proposals are received from all currently correct processes and there has not been a decision (line 36). If a failure has been detected (ie the correct processes for the round and one before do not match (line 38), then another round is required. The proposals seen so far, are broadcast so as to disseminate its *Proposals* and enable other processes to progress. It should be noted that all previous proposals will be "brought forward" in the new round - this is because when a process broadcasts its proposal set (line 45), this will also be received by this same process.

Note that unlike the *floodset* algorithm, there is no strict deadline for receiving the proposal messages. Instead, for each process, the algorithm either receives proposals or detects that it has crashed. The eventual delivery property of the link guarantees that if proposals are sent, these will be received, whereas the properties of the perfect failure detector guarantee that any crashed node will be detected. This guarantees that one of these two possibilities will hold for all the processes during a round, eventually terminating it.

If for the last two rounds, no failure is detected (ie there are two successive equal sets of PID's, in the *Correct_this_round* structure) then a decision can be made (lines 38-41). The decision is made by applying the decision function *Func* (which was passed as an argument to *init()*) on the final set of proposals.

After a process decides, it broadcasts this decision (line 41) to the other processes. This is done because the process which managed to decide, might have received proposals from a process which only managed to send its proposals to this process before crashing. Hence, these processes would detect the crashed process and not decide. However, in the following round, instead of receiving the proposals of the process which decided, these will receive the decision and hence trigger *decide()* without need for further rounds.

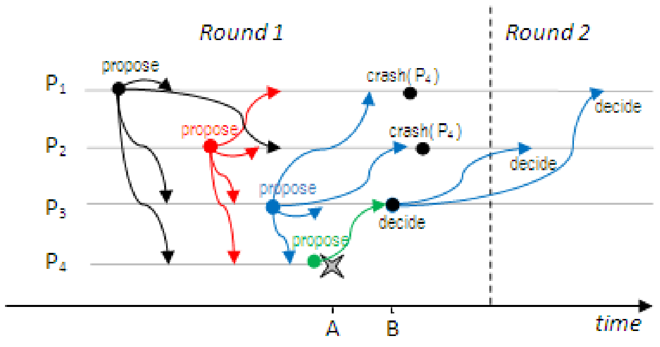


Figure 5.6: **A run of the Flooding Algorithm with one process failure**

Figure 5.6 shows a run of the *flooding* consensus algorithm between four processes. Here, the first three processes are correct and manage to broadcast their proposal. However, process P_4 crashes whilst broadcasting its proposals and only manages to send its proposal to P_3 (point A). As soon as this message is received by process P_3, this process would have received proposals from all correct processes. Moreover, since $Correct\_this\_round[0] = \Pi$, this process can now decide. At point B, P_3 decides and broadcasts its decision to the other processes. In the meantime, processes P_1 and P_2 detected the crash of P_4 and hence moved on to round 2 without being able to make a decision (Note that strictly speaking, these processes broadcast their proposals on the beginning of round 2, but this is irrelevant in this example and hence is left out for the sake of simplicity). In the second round, these processes receive the decision and hence decide immediately.

Note that it is not possible for two processes to decided differently. This is because for a process to decide, there must be two consecutive rounds without detecting a failure. Moreover, at the end of the first round, the proposals are broadcasted to all. Hence, even if the failure detectors of different processes do not detect the same failed processes, the proposal sets would still be exchanged at the end of the first round and hence if the two decide, they will decide the same.

5.4.3.4 Erlang Implementation

The implementation of this algorithm, in Erlang, uses lots of the modules and techniques already outlined in the implementation of previous algorithms such as the usage of the *sets* module, for the *Correct* set.

The *Correct_this_round* and *Proposals* arrays where implemented using Erlang's *Array* module. This provides a data structure with conventional array like access, which is much more convenient and efficient to use for this purpose than the conventional lists.

Another important feature demonstrated in this algorithm is the use of the decision function *Func*. This function should be passed as a parameter to *init()*, and then used when deciding. In Erlang, this can be implemented using Higher Order Functions in almost the same way as shown here.

5.4.3.5 Implementation Optimisations

As opposed to the reliable broadcast, every consensus algorithm has definite stopping point during its execution - after that a decision is reached, nothing will affect the decision. Thus, after that a decision is reached, all that needs to be kept is the *Decided* variable. The rest of the elements of the state can be given the value *null*, and Erlang's garbage collector will take care to free memory it previously occupied.

Note that such a technique could not be done with the reliable broadcast implementations, because all algorithms require data structures whose content was required to guarantee correctness. In this case however, just storing the decision (or a flag indicating that the consensus algorithm has dedided) is enough.

5.4.3.6 Evaluation

The *flooding algorithm* for *regular consensus* ensures that all correct processes will decide on the same value. It also guarantees all properties of consensus. *Termination* is guaranteed because each process will progress to the $|\Pi|$'th round, unless a decision is reached earlier. At this point, it will decided and hence terminate. *Validity* is satisfied because the *Proposals* is only populated with proposals received - and no message is erroneously created on its own. *Validity* is guaranteed because once *Decided* is assigned a decision, the predicate in line 48 will never evaluate to true, and hence the algorithm would not decide again. *Agreement* is guaranteed because in every round a process waits to receive the proposals from all processes. If a process decides, then this means that it has received proposals from all correct processes for two consecutive rounds, at which point it will broadcast its decision. If two or more processes decide, then their proposal sets should be identical because these would have received proposals from the same processes - and hence should have taken the same decision.

In terms of performance, the algorithm will terminate after just one round of execution, if no process crashes. Every process crash, will introduce another round of execution. With respect to the number of messages exchanged, in every round $|\Pi|^2$ proposal messages are sent. When a decision is reached, another $|\Pi|^2$ decided messages are transferred. In the worst case, the algorithm requires $|\Pi|$ rounds.

5.4.4 Hierarchical Algorithm

The previous section outlined a valid algorithm for *Regular Consensus.* Here another algorithms is presented, which despite having different characteristics, still satisfies the *Regular Consensus* specification.

The *Hierarchical* consensus algorithm, exchanges less messages than the *flooding* algorithm. However, the decision taken, does not depend on the collective proposals of the participants, but rather, on just the proposal of one participant. The algorithm guarantees that this proposal will be the decision taken by all processes, whatever the other processes may propose.

The algorithm works by sequentially ordering all processes. The first processes in the ordering will have a priority in attempting to impose their proposal as a decision - this can be seen as a hierarchy of processes, where the higher up the processes are, the more the chance they have of imposing their proposal on others. The algorithm works in rounds, where in each round a particular process broadcasts its current proposal and locally callbacks the *decide()* event. Initially the first process in the hierarchy broadcasts its proposal and decides. No other process decides in this round, but as soon as they receive the proposal from the current round leader, they change their internal proposal to the value received from the round leader. A process moves on to the next round, either when it receives the value proposed by the current round leader, or when it detects that the current round leader crashed.

The listing in figure 5.7 gives the *hierarchical* consensus algorithm. Internally every process keeps the current *Proposal*, as well as the rank of the process which proposed it (*Proposer*). Note that a way to rank processes is assumed, here it is assumed that a *rank(Pid)* function exists, which returns the rank of the passed PID. The algorithm also keeps an array with the round number as index, and a boolean flag as element, indicating whether the proposal has been received from the corresponding round leader.

At every round, the process whose rank is equal to the round number (line 31) is the round leader. The round progresses as soon as the leader process has a value to propose, or crashes. The first leader will send its proposed value to all other processes and decide (lines 32-34). Upon receiving this proposal, the receiver processes, whose rank is higher than that of the leader, *may* update their *Proposal* and *Proposer* values (lines 39-45). This will only happen if the receiver has not yet received a proposal from a node with a higher rank. If the received has already seen a proposal from a process with a higher rank, then it is implied that either the current sender has crashed and the next leader has taken over or that the two proposals are the same - in both cases the proposal received should be ignored.

If the proposal from the current round leader has been received, or the current round leader is detected to have crashed, then the algorithm moves on to the next round (lines 36-37). Note that at every round, the round leader will again broadcast its current proposal and callback the *decide()* event.

The reason for each process having to wait for "its" round to decide, rather than *decide()*ing immediately after it receives the proposal from the current round leader, is because of failures. This is further highlighted in the run given in figure 5.8.

The run in figure 5.8, shows three processes. Note that there is an assumed hierarchical ranking as follows: $rank(P_1) < rank(P_2) < rank(P_3)$. Hence the first round leader is P_1. This starts broadcasting its proposal but crashes shortly afterwards. Note that this proposal actually reaches P_3 but not P_2 (point A). At this point, P_3 adopts P_1 proposal and moves on to round 2, but does not callback the decide() event. In round 1, process P_2 detects that P_1 crashed, and so can move on to round 2 as well. In round 2, P_2

```
-module( hierarchical_consensus ).

-behaviour(best_effort_broadcast).
-behaviour(perfect_failure_detector).

%% State Definition
-state(
        Detected,  %% Set of ranks of failed processes
        Round,     %% Local round number
        Proposal,  %% The value which will be proposed to others
        Proposer,  %% Rank of process which proposed the current value
        Decided,   %% Flag indicating whether algorithm decide()ed
        Delivered  %% Array of flags showing whose proposals were delivered
      ).

%% Event Handlers
upon event init() ->
        Detected = ∅,
        Round = 1,
        Proposal = ⊥,
        Proposer = 0,
        Decided = false,
        ∀ I ∈ [1,|Π|] · Delivered[I] = false.

upon event propose( V ) ∧ (Proposal == ⊥ )  ->
        Proposal = V.

upon event crash(Who) ->
        Detected = Detected ∪ { rank(Who) }.

upon (Round == rank(self()) ∧ (Proposal ≠ ⊥ ) ∧ (Decided == false) ->
        Decided = true,
        callback rc_decide(Proposal),
        trigger best_effort_broadcast:broadcast({Round, Proposal}).

upon (Round ∈ Detected) ∨ (Delivered[Round] == true) ->
        Round = Round + 1.

upon event beb_deliver( From, {R, V} ) ->
        if
            (R < rank( self() )) ∧ (R > Proposer) ->
                Proposal = V,
                Proposer = R
        end,
        Delivered[R] = true.
```

Figure 5.7: **The Hierarchical Consensus Algorithm**

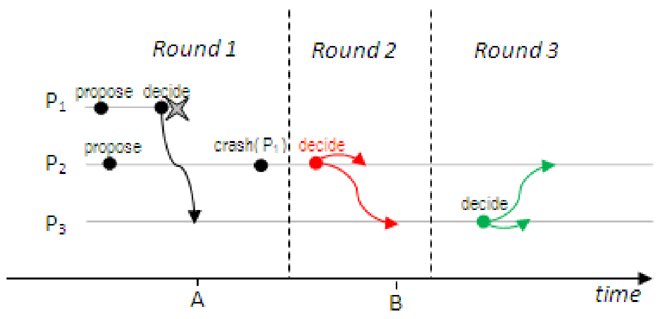


Figure 5.8: **A run of the Hierarchical Consensus algorithm with leader failure**

is the round leader. Since P_2 did not receive the proposal from P_1, it will *decide()* and broadcast, its own proposal as a decision. Process P_3 receives this proposal at point B, and hence it adopts this new proposal since it comes from a process with a higher ranking than its existing proposal. In round 3, process P_3 decides its current proposal, which is the one originating from P_2. Hence in this run, P_2's proposal is taken by all correct processes. This shows why all processes need to wait for their round in order to decide() and not decide() as soon as the proposal from the leader is received = as show with process P_3.

As can be seen from the run in figure 5.8, not all processes are required to propose a value (P_3 does not propose). Essentially, if there is no failure, only the first process in the hierarchy needs to propose, and all the rest will adopt its proposal. If there are failures, then only the originator of the proposal reaching the first non-faulty process in the hierarchy, needs to propose. Notice that the round leader, would propose it current proposal, which it may have adopted from previous round leader, and not the proposal which was triggered by the user of that process. Also note that if the round leader does not yet have a proposed value, then the algorithm will wait for that process to *propose()*.

5.4.4.1 Erlang Implementation

The hierarchical consensus algorithm implementation uses the Erlang modules outlined so far - the main ones being the *sets* module for the *Detected* set and *array* module for the Delivered array. All elements of the *Delivered* array can be easily initialized to false, given that the array is of fixed size.

One notable thing to mention is the implementation of the *rank()* function itself. In this work, the rank function is passed as a parameter to the *init* function, in order to allow the user to define his own ranking function. A simple practical implementation of the ranking function, would be to simply sort the node names given on startup (see appendix A), and give rank each node according to this ordering.

5.4.4.2 Implementation Optimisations

The hierarchical consensus algorithm, given here exchanges $|\Pi|$ messages at every round. However, one should note that when broadcasting the proposal, this can only affect the processes with higher ranks, since the other processes will simply ignore them. Hence, the leader should only send to the proposal to processes with higher ranks [14], and save on message exchanges.

5.4.4.3 Evaluation

The *hierarchical* consensus algorithm exchanges less messages than the *flooding* algorithm. However, the decision does not depend on the proposals of all the processes, but rather on the proposal of a single process, which imposes this value on all correct processes. In situations where it does make a difference if not all proposals are seen, then this algorithm should be preferred over the *flooding* algorithm for consensus.

The *hierarchical consensus* algorithm guarantees all properties of *regular consensus.* It guarantees *termination* because at every round, a process either receives the decision from the current round leader or else detects that this has crashed - either conditions are guaranteed by the *eventual delivery* of links and the *strong completeness* property of the failure detector. At this point it moves on to the following round and hence eventually the process will become a leader and the algorithm will terminate. *Validity* is guaranteed because the decision is always the proposal of the round leader. *Integrity* is satisfied because a *Decided* flag is kept to prevent a process from deciding multiple times. Finally, *agreement* is satisfied, because every process always adopts the decision of its leader and propose this decision - hence all correct processes will decide the same value.

In terms of performance evaluation, the algorithm requires $|\Pi|$ rounds for all processes to decide. Moreover, at each round $|\Pi|$ proposals are exchanged (assuming the process broadcasts its decision to all processes).

5.5 Uniform Consensus

5.5.1 Overview

The *Uniform consensus* specification, strengthens the *regular consensus* specification by ensuring decision agreement amongst all processes. The *regular consensus* specification allows processes to decide and crash, without being able to propagate their decision - causing the rest of the processes to possibly decide on a different value. The *uniform consensus* specification prevents this from happening - once a process decides, all processes are guaranteed to decide with that same value, even if there are process failures.

5.5.2 Specification

The *uniform consensus* specification aims to achieve properties similar to those which *uniform reliable broadcast* brings to *regular reliable broadcast.* In guarantees that a faulty process could not have taken a decision different from that taken by all correct processes.

Uniform consensus replaces the *agreement* property of *regular consensus,* with a stronger agreement (*uniform agreement*) property. The properties guaranteed by:

- **Termination**: There exists a time, at which all correct processes decide some value.
- **Validity**: A decided value must be a proposed value.
- **Integrity**: Processes decide only once.
- **Uniform Agreement**: All processes agree on the same value.

5.5.3 Algorithm

The listing in figure **??**, gives a uniform flooding consensus algorithm. This algorithm is based on the *floodset* algorithm for solving consensus in synchronous systems. In particular, it is based on the case where $f = |\Pi| - 1$,

```
-module( uniform_consensus ).

-behaviour(best_effort_broadcast).
-behaviour(perfect_failure_detector).

%% State Definition
-state(
        Correct,     %% Set of Correct Processes
        Decided,     %% Decision result
        Proposals,   %% Set with all proposed values
        Round,       %% Current Round Number
        Delivered,   %% An array with PID's of proposal sender for all rounds
        Func         %% Decision Function
      ).

%% Event Handlers
upon event init(Function) ->
        Correct = Π,
        Decided = ⊥,
        Proposals = ∅,
        Round = 1,
        Func = Function,
        ∀ I ∈ [1,|Π|] · Delivered[I] = ∅.

upon event propose( V ) ->
        Proposals = Proposals ∪ {V},
        trigger best_effort_broadcast:broadcast( {1, Proposals} ).

upon event beb_deliver(From, {R, Proposals_rcvd} ) ->
        Proposals = Proposals ∪ Proposals_rcvd,
        Delivered[R] = Delivered[R] ∪ {From}.

upon (Correct ⊆ Delivered[Round]) ∧ (Decided == ⊥) ->
        if
            Round == |Π| ->
                Decided = func(Proposals),
                callback uc_decide( Decided ),
            Else ->
                Round = Round + 1,
                trigger best_effort_broadcast:broadcast({Round, Proposals})
        end.

upon event crash( Who ) ->
        Correct = Correct \ {Who}.
```

Figure 5.9: **The Uniform Flooding consensus algorithm**

which would requires $|\Pi|$ rounds. Conceptually, it can be seen that for a decision to take place, then there can be a maximum of $|\Pi|-1$ failures (otherwise there would not be any process left alive to decide). Hence, letting $f = |\Pi| - 1$, would ensure that the *floodset* algorithm handles all possible failures. Moreover, all processes only decide during the final round, at which point it is guaranteed that all processes have an identical *Proposals* set.

At each round, processes exchange their internal proposal sets. After $|\Pi|$ rounds, all processes will have identical *Proposals* sets, because there should have been at least one round in which there are no failures and all *Proposals* sets were exchanged. Hence, at this point all processes will decide the same value - satisfying the *uniform agreement* property.

At each round of the *floodset* algorithm, every process either receives the proposal set from every other participant, or else the round times out and the algorithm moves on to the following round. In the algorithm for *uniform consensus*, a failure detector is used to determine whether to wait to receive the proposals from a process or not. Internally, a *Correct* set is kept with the PID's of all processes which the failure detector does not detect as having failed. Moreover, similar to the *floodset* algorithm, an array of sets (*Delivered*) is kept to store the PID's of processes from whom the proposals were received at every round (lines 29-31). A process moves on to the following round when it determines that it has received a proposal from all correct processes (line 33).

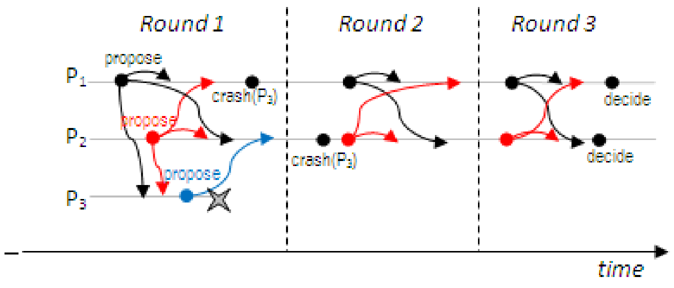


Figure 5.10: **A run of the uniform flooding consensus with one process failure**

Figure 5.10 gives a run of the algorithm for *uniform consensus* for three processes. Hence this algorithm requires three rounds to terminate. Process P_3 fails in the first round, after that its proposal is only received by P_2. In this round, process P_2 receives proposals from all processes and hence can move to the next round. Process P_1 does not receive a proposal from P_2, but detects that it has crashed, and hence can also move to the following round. In the second round, there is no failure and hence all correct processes manage to see the whole set of proposals. Nevertheless, the proposals sets are exchanged another time in round 3. Processes P_1 and P_2 decide in this final round and can be sure that no faulty process has decided a value different from their decided value.

5.5.4 Erlang Implementation

This algorithm was implemented using the framework outlined for implementing these algorithms in Erlang. As in other algorithms, all event handlers were implemented as *gen_server* cast handlers. The *sets* module was used for the implementation of the *Proposals* and *Correct* sets. The *Delivered* array was implemented using the Erlang/OTP *Array* module. It should also be noted that the decision function, which is passed as a parameter to the *init* event, was implemented as a higher order function.

5.5.5 Implementation Optimizations

As a simple memory optimisation, it is noted that the algorithm internally keeps a *Delivered* set with the PID's of processes at every round. However, as soon as a process moves on to the following round, it would no longer need this set, and hence could be discarded. Hence, at the beginning of every round, the entry for the *Delivered* set of the previous round is emptied (set to the atom null), and the Erlang garbage collector will take care of freeing the occupied memory. Moreover, after deciding, the same technique can be done to all the sets used internally by this algorithm.

5.5.6 Evaluation

The algorithm for *uniform consensus* ensures that if a process decides, all other processes will decide with that same value. This is precisely the *uniform agreement* property of uniform consensus. This property is satisfied because the algorithm ensures that there is a round in which there are no failures, hence all proposals sets become identical in that round, and hence all processes which decide would decide with the same value. Moreover, this property is only guaranteed with a *perfect failure detector*. Since the failure detector in this project is not purely perfect, it could be that a process incorrectly detects another as having failed, and hence does not wait to receive proposals from this supposedly failed process. Even though that in this project, once a process detects another one as having crashed, it sends it a *kill* message, this message can take a long time to be delivered. If this happens, the process which was supposed to have crashed, can still have time to decide on a different proposal set then the other process' proposal set.

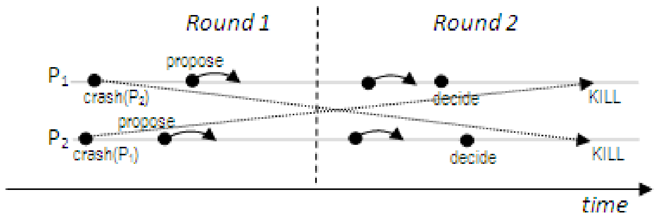


Figure 5.11: **Violation of uniform agreement due the weak accuracy of the failure detector**

The run in figure 5.11 indicates the violation of the *uniform agreement* property due to an inaccurate failure detector. Initially, processes P_1 and P_2 incorrectly detect one another as having crashed, and send *kill* messages, which take very long to be delivered (shown by the dashed lines). This causes both processes to move on to the following rounds before waiting for one another. Both processes reach the final round, and decide without having seen each others proposals. Hence, each decide on a different proposal sets, leading to different decisions. This simple example shows the true necessity of a *perfect* failure detector, because this would not have such accuracy problems.

The other properties of *uniform consensus* are also satisfied by this algorithm. *Termination* is guaranteed because all processes eventually progress from one round to the following because of the eventual delivery of messages or of the detection of processes which have crashed (failure detector *completeness*. *Validty* is guaranteed because proposal sets are only populated with actual proposals. Finally, *integrity* is guaranteed since is a specific check (line 33) to ensure that if the process has already decided, then it would not decide again. In terms of performance evaluation, the algorithm requires a fixed number of rounds. In fact, the best and worst cases of the algorithm are the same (it always requires $|\Pi|$ rounds. In each round, every process broadcasts to all other processes, hence $|\Pi|^2$ messages are exchanged in every round. This results in the algorithm exchanging $|\Pi|^3$ messages in total.

5.6 Total Order Broadcast

5.6.1 Overview

In the previous chapter, different specifications of *reliable broadcast* protocols were outlined. In this section, another *reliable broadcast* protocol will be outlined: *total order broadcast*[2]. As its name suggests, this *reliable broadcast* specification guarantees that all distributed processes deliver all broadcast messages in the same order. Note that such a protocol would be stronger than the *causal order* specification because it guarantees ordering amongst all messages, not just those which are causally related. Given such a guarantee, this protocol can be very useful when certain non-commutative operations are to be carried out - given a *total order broadcast* protocol these operations can be globally executed in the same order.

5.6.2 Specification

The *Total Order Broadcast* specification is similar to that of the *Regular Reliable Broadcast* but with a global message delivery ordering. In the *Regular Reliable Broadcast* there existed the possibility that due to the asynchrony of the network, different processes deliver messages in a different order. This situation can be seen in the run in figure 5.12 below. Here, processes P_1 and P_3 each start a *regular reliable broadcast.* Focusing on processes P_2 and P_4, note that due the asynchrony of the network, these processes deliver the messages in different orders. Note also that these broadcasts are not causally related (no other message is exchanged between either node to define such a relation), hence the use of a *causal order broadcast* protocol would not

[2]In certain texts, this protocol is referred to as *Atomic Broadcast*

resolve this problem.

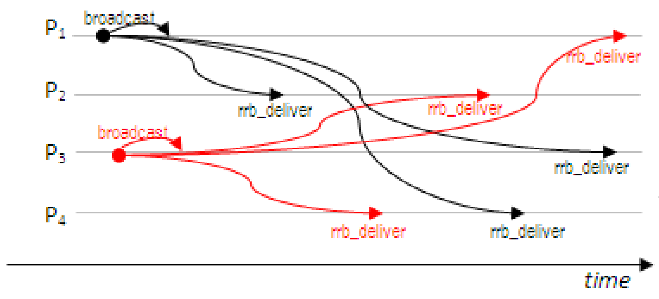


Figure 5.12: **Violation of total ordering with regular reliable broadcast due to the asynchrony of the network**

The *Total Order Broadcast* specification guarantees a *total order* property which would ensure the identical order of messages delivered by all processes. The properties satisfied by the *total order specification* are:

- **Total Order**: For any two messages m_1 and m_2, if a process delivers m_1 before m_2, then all other process deliver these messages in that same order.
- **Validity**: If a correct process p broadcasts message m, this eventually gets delivered by all correct processes in the system.
- **Integrity**: For any message m, every process delivers m at most once, and only if it was previously broadcast.
- **Agreement**: All correct processes deliver the same set of broadcast messages.

5.7 Algorithm

The listing in figure 5.13 gives a *regular consensus* based algorithm for the *total order broadcast* specification. This algorithm uses both the *regular reliable broadcast* and *regular consensus* behaviours. Conceptually the algorithm works by first reliably disseminating messages which need to be broadcast. Following this, every process proposes a set of messages, it has

```
-module( total_order_broadcast ).

-behaviour(regular_reliable_broadcast).
-behaviour(regular_consensus).

%% State Definition
-state(
        Unordered,          %% Set of Messages which have not been delivered
        Delivered,          %% Set of Messages which have been delivered
        Sequence_number,    %% An integer grouping all messages in the round
        Wait                %% A flag indicating there is a pending round
      ).

%% Event Handlers
upon event init( Order_function ) ->
        Unordered = ∅,
        Delivered = ∅,
        Sequence_number = 1,
        Wait = false,
        trigger regular_consensus:init(Order_function).

upon event broadcast(Msg) ->
        trigger regular_reliable_broadcast:broadcast(Msg).

upon event rrb_deliver( From, Msg ) ->
        if
             (Msg ∉ Delivered) ->
                 Unordered = Unordered ∪ { {From, Msg} }
        end.

upon (Unordered ≠ ∅) ∧ (Wait == false) ->
        Wait = true,
        trigger regular_consensus:propose({Sequence_number, Unordered}).

upon event rc_decide( {Sn, Decided} ) ->
        Delivered = Delivered ∪ Decided,
        Unordered = Unordered \ Decided,
        ∀{Sender, Msg} ∈ Decided ·
                callback torb_deliver(Sender, Msg),
        Sequence_number = Sequence_number + 1,
        Wait = false.
```

Figure 5.13: **Total Order Broadcast Algorithm**

seen, using a *regular reliable consensus* algorithm. The decision of this algorithm, is the set of messages which will be delivered by all processes in the current round. This set is identical for all processes and hence a deterministic ordering function on this set will always order the messages in the same way everywhere.

Internally, the algorithm keeps two sets *Unordered* and *Delivered*. The *Unordered* set is used to hold pending messages in every round, before proposing them using the regular consensus algorithm. Note that a *Wait* flag is used (lines 31 and 32) to prevent the algorithm from proposing a new set of messages whilst a consensus is taking place. However, during this time period, more messages might be broadcast and hence the need for the *Unordered* set.

When the broadcast event is triggered (line 22), this causes the message to be broadcast using the *regular reliable broadcast* specification (line 23). This is done so that the rest of the processes will have an undelivered message in their *Unordered* set and hence propose their *Unordered* set. The *Unordered* set might not be proposed immediately, as explained earlier, but the consensus proposal for the current round will eventually be made.

At this point, the necessity of executing a consensus might be unclear since messages are being broadcast reliably. However, it must be noted that these messages might arrive in different orders at different processes. Hence these must be ordered in some way or another. This cannot be done unless every process has an identical set on which to apply an ordering function. In order to achieve this set, at the end of the round every process proposes its set using *regular consensus*.

Note that the decision function of this consensus should return a set of messages from the set of all proposed messages. Here the decision function is passed as a parameter to the *init* event (line 20). A very simple implementation can be to simply return the set of all proposals, however if this is very large, one could consider to just return a subset of all the proposed messages, at a time.

As soon as this set is decided, the *rc_decide* event is triggered (line 35). All messages in this set should be included in the *Delivered* set and removed from the *Unordered* set to prevent them from being erroneously re-proposed again (lines 36-37). All messages in this set can now be delivered (lines 38-39). It should be noted that here messages are extracted from the *Decided* set using some deterministic order - hence in the same order at all processes. At this point, the algorithm enters a new round, in which the same operations are carried out again.

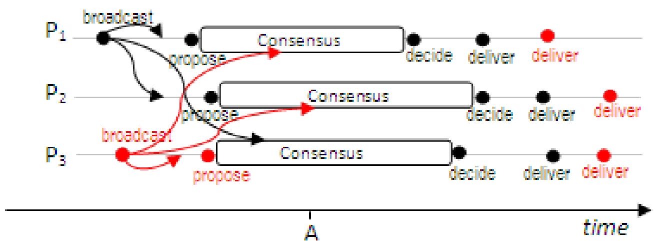


Figure 5.14: **A run of the consensus-based Total Order Broadcast Algorithm**

Figure 5.14 gives a run of a round of the *consensus-based total order broadcast* algorithm. Processes P_1 and P_3 broadcast messages. The delivery of these messages causes the processes to propose it as a value for consensus. Processes P_1 and P_2 propose P_1's message since these deliver this message first, whereas process P_3 proposes its own message. At this point, the processes execute a regular consensus algorithm. Note that processes deliver reliable broadcast messages, whilst executing their consensus (example P_2 at point A), but such messages do not trigger another proposal event.

When the execution of consensus is ready, and a set of messages is decided, the message can be deterministically *deliver*ed in the same order by all processes.

5.8 Erlang Implementation

Implementing this algorithm in Erlang, uses techniques explained for the implementation of algorithms outlined so far. This algorithm uses two behaviours, defined previously in this work - *regular reliable broadcast* and *regular consensus*. The implementation also uses Erlang/OTP sets module for the implementation of the *Unordered* and *Delivered* set.

The algorithm requires that messages are extracted from the *Decided* set in a deterministic order. In practice, this is done by converting the *Decided* set to a list and then sorting this list. In this implementation, messages are sorted in an ascending order depending on their content. This sorting is required because the sets *to_list/1* function does not guarantee a deterministic ordering.

5.9 Implementation Optimisations

The algorithm maintains a *Delivered* set of messages to avoid redelivering a message multiple time. This set however grows indefinitely and would require an ever growing amount of memory. In order to relieve this requirement, it suffices to only store hashes of messages. Hashes can be easily computing in Erlang by using the *crypto* module.

5.10 Evaluation

The *total order broadcast* specification provides a reliable broadcast algorithm with strict ordering guarantees - every process delivers messages in the same order. This *total order* property is guaranteed processes use a consensus abstraction to determine the set of messages to deliver at every round, and then use a deterministic function to deliver all messages in this set in the same order. This also guarantees *agreement* amongst processes on the messages delivered. *Validity* is guaranteed because every message is first disseminated to all other processes using a *regular reliable* broadcast (hence all correct processes will see this message). *Integrity* is guaranteed because processes only propose messages which they receive from on another.

With regards to performance, the algorithm exchanges exactly the same number of messages as a those exchange by a *reliable broadcast* algorithm and a *regular consensus* algorithm. Hence, in the best case (no failures), it would require $|\Pi| + |\Pi|^2$ message exchanges.

5.11 Conclusion

This chapter explored the Consensus class of agreement algorithms. First, the *regular reliable consensus* specification was outlined. Two algorithms were explained for this specification: the *flooding* algorithm and *hierarchical* algorithm. Each of these have different characteristics in terms of the way the decision is made, and also in terms of performance. Following this the *uniform consensus* specification was outlined. It guarantees that no faulty process decides with a decision different than that of all correct processes.

A reliable broadcast specification was also studied: *total order broadcast*. This broadcast specification guarantees a global ordering of the messages delivered by all processes. A consensus based algorithm for this reliable broadcast specification, was analysed.

All the algorithms presented in this chapter, were implemented in the implementation framework outlined for this project. A number of implementation considerations were mentioned, as well as some simple implementation optimisations.

Chapter 6

Atomic Commit

6.1 Introduction

The last agreement problem to be studied in this work involves is the Atomic Commit problem. The Atomic Commit problem deals in determining whether all processes can carry out a particular operation, or not. If so, all processes would carry out the operation and are said to have committed. If at least on process cannot commit, then the operation should be aborted by all processes.

This problem is typically encountered in a distributed database environment, where all processes are required to carry out an operation as a group [8]. If committed, this operation is seen as having been carried out by all processes together, and hence is seen as an *atomic* operation.

This chapter investigates different algorithms for this specification. Various considerations on popular synchronous implementations are outlined. Finally, an asynchronous algorithm is also studied and implemented in Erlang.

6.2 Interface

The generic interface of the atomic commit algorithms is given below:

- **External Events Handled**
  - **init()**
    Initializes the algorithm

    - **commit()**
      Starts a commit request.

- **Callbacks Expected**
    - **can_commit()** returns **boolean**
      This callback event is used to determine whether the current process can commit or not. Inside its handler, the process should check whether it can commit and return a boolean, where *true* indicates that it can commit and *false* indicates that it cannot.
    - **decide(** *Value* **)** where *Value*$\in \{0, 1\}$
      This callback marks the end of the commit algorithm. The decided value indicates the outcome of the commit. If $Value = 1$ then the commit action should be carried out, whereas if $Value = 0$ then the commit should be aborted.

6.3 Specification

An algorithm for the *Atomic Commit* specification should satisfy the following properties [14]:

- **Uniform Agreement**: All processes decide take the same decision on whether to commit or abort.
- **Integrity**: If a process decides to abort, it cannot decide to commit later on, and vice versa.
- **Abort Validity**: The operation should be aborted only if a process cannot commit or a process crashes.
- **Commit Validity**: The operation can be committed only if all processes agree to commit and no process crashes.
- **Termination**: All correct processes decide on whether to commit or abort.

6.4 Synchronous Approaches

In the context of synchronous systems, there are two common protocols for solving the atomic commit problem. These are the *Two Phase Commit* and *Three Phase Commit* protocols. The *Three phase commit* protocol is an improvement of the *Two Phase Commit* problems, since it resolves a blocking issue inherent in the latter protocol. For this reason, the *Two Phase commit* protocol is referred to as a *Blocking* protocol, whereas the *Three Phase commit* protocol is a *Non-blocking* protocol.

Both these algorithms are divided in two roles: the *Coordinator* role and the *Cohort* role. The *Coordinator* is the initiator of the commit request. It acts as a leader of the current commit action and orchestrates actions to the cohorts. The *Cohort*'s role is that of checking whether the commit request can be locally carried out or not, and indicating this to the coordinator. It also aborts or commits the action as requested by the coordinator.

6.4.1 Two Phase Commit Algorithm

As explained earlier, this algorithm can be separated into two roles. First, the *Coordinator* role is described. The Coordinator part of the *Two Phase commit* algorithm is given in the listing in figure 6.1.

```
-module(two_phase_commit ).

-behaviour(best_effort_broadcast).

%% State Definition
-state(
        Replied   %% Set of processes which can commit
      ).

%% Event Handlers
upon event init() ->
        Replied = ∅.

upon event commit() ->
        trigger best_effort_broadcast:broadcast(query_to_commit).

upon event received({reply_yes, From}) ->
        Replied = Replied ∪ {From}.

upon Replied == Π ->
        trigger best_effort_broadcast:broadcast(commit).

upon event received({reply_no, From}) ∨ (timedout phase 1) ->
        trigger best_effort_broadcast:broadcast(abort).
```

Figure 6.1: **The coordinator's role in the Two Phase Commit algorithm**

As its name suggests, the actions within the *Two Phase commit* algorithm can be divided into two consecutive phases. The first phase starts when the *commit* event is triggered (line 14). In this phase, a *query_to_commit* message is broadcast to all cohorts. This message indicates that a commit operation is underway and that the cohorts should check whether the commit operation can take place locally or not. The cohorts should then reply with a *reply_yes* or a *reply_no* message indicating that the cohort can carry out the operation or not respectively.

As soon as a *reply_yes* message is received, the PID of the sender of such message is included in an internal *Replied* set. If the coordinator receives a *reply_yes* message from all processes, then it can move to the second phase (lines 20-21). In this phase, the coordinator sends a *commit* message which instructs the cohorts to actually commit the operation.

On the other hand, if the coordinator receives a *reply_no* message (line 23), then the commit operation should be aborted. This is done by sending an *abort* message to the cohorts, instructing them to discard the commit operation.

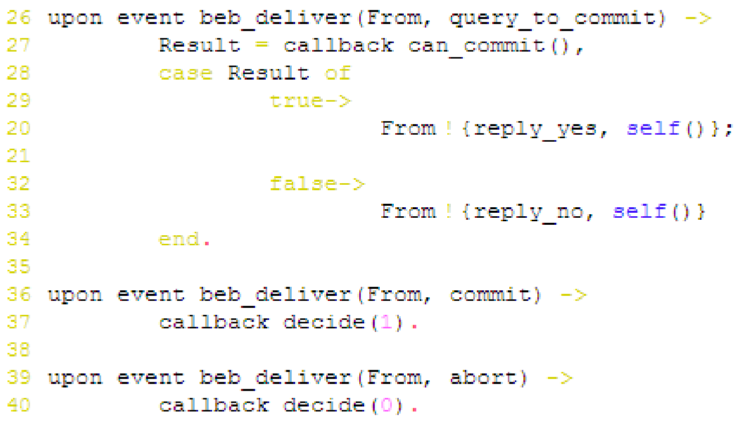
Note that on line 23, the abort message is sent also if phase 1 timesout. This means, that not all cohorts reply within a fixed time frame, after being sent the *can_commit* message. Such a timeout, in a synchronous system, means that a cohort crashed and hence the commit operation needs to be aborted.

The algorithm for the cohort role, corresponds with the messages sent by the algorithm of the coordinator. The Cohort part of the *Two Phase Commit* algorithm is given in the listing in figure 6.2.

Firstly, the algorithm handles the *can_commit* message from the coordinator. This message causes the cohort to callback the *can_commit* event (line 27). This event is a check, to ensure whether the commit operation can be done. If the *can_commit* event returns true (the cohort can commit), then a *reply_yes* message is sent, otherwise a *reply_no* message is sent (lines 28-34).

In the second phase, the algorithm callbacks *decide(1)* or *decide(0),* depending on whether a *commit* or *abort* messages were received from the coordinator, respectively.

Figure 6.3 gives a run of the two phase commit algorithm. Process P_1 is the coordinator of the commit operation. It starts by broadcasting the *commit* message to all processes including itself. The cohorts reply with a



```
upon event beb_deliver(From, query_to_commit) ->
        Result = callback can_commit(),
        case Result of
                true->
                        From ! {reply_yes, self()};

                false->
                        From ! {reply_no, self()}
        end.

upon event beb_deliver(From, commit) ->
        callback decide(1).

upon event beb_deliver(From, abort) ->
        callback decide(0).
```

Figure 6.2: **The cohort role of the two phase commit algorithm**

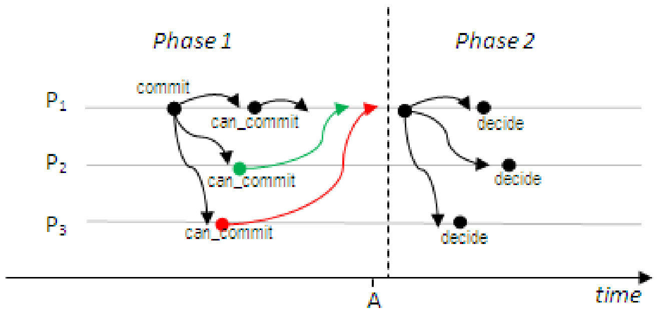


Figure 6.3: **A run of the Two Phase commit algorithm**

reply message to the coordinator. The coordinator receives replies from all cohorts at point A, and so the algorithm moves to the second phase. In this phase, assuming that all cohorts could commit, the coordinator broadcasts the *commit* message which trigger, the *decide* callback events at all cohorts.

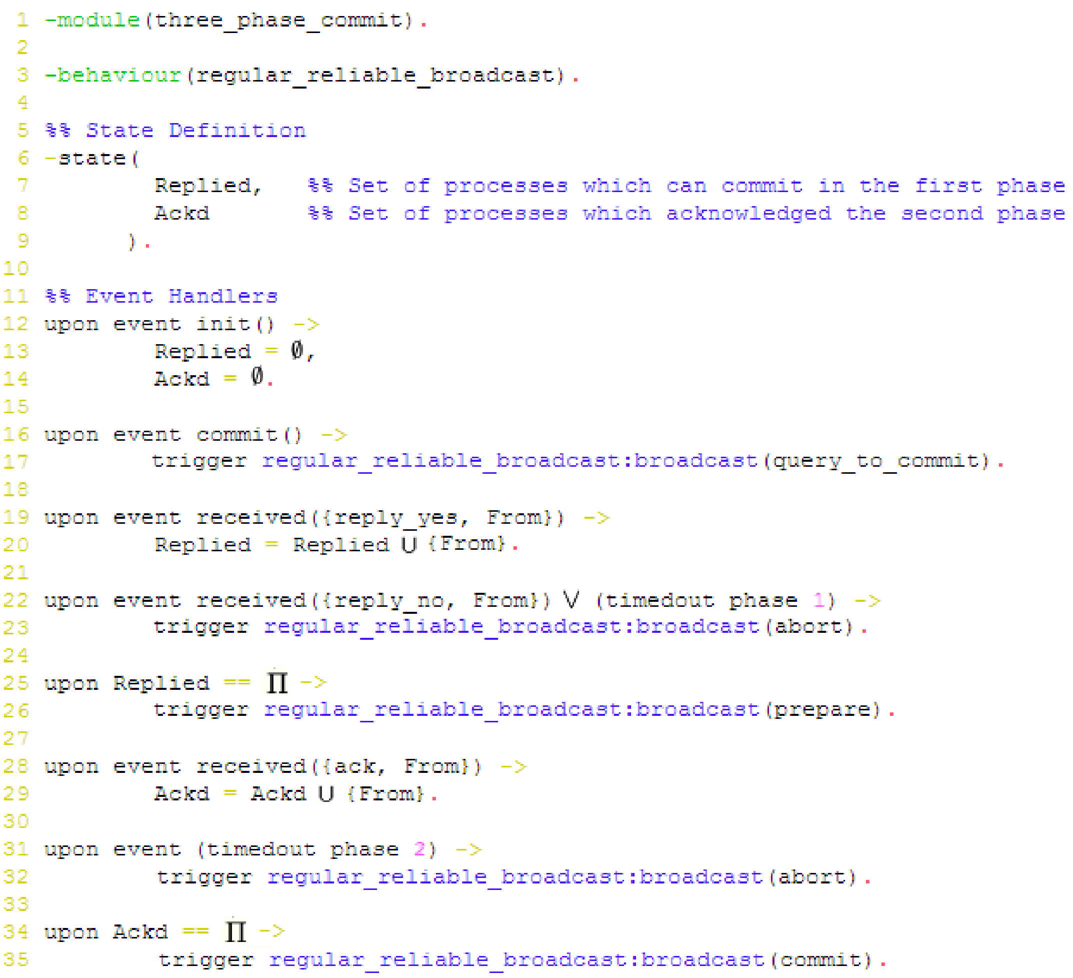
6.4.1.1 Blocking concerns

The use of the *best effort broadcast* raises concerns on the failure resiliency of the algorithm, in particular on the event of coordinator failure in the middle of this broadcast. It can be noticed, that despite the algorithm would not commit or abort the operation, it would block waiting for such messages from the coordinator. In particular, if a process receives a *can_commit* message and is ready to commit, but never receives the *commit* confirmation message from the coordinator, it cannot decide whether to commit or not. This happens because if it decides to commit, other processes might have received an *abort* message in the mean time. Conversely, if it decides to abort, then it could be that other processes did actually receive a *commit* message. This would leave the process waiting blocked without taking any action. The coordinator might recover in the meantime and be able to resolve the issue. If this does not happen, however, the cohorts will remain waiting, whilst possibly holding resources for ever.

It should also be noted, that if the *best effort broadcast* protocol is replaced by a reliable broadcast protocol, then there might still be cases where the cohorts remain blocked. In particular, if the coordinator crashes before sending a single *commit* or *abort* message, then all cohorts would remain waiting for these messages. Moreover, even if the cohorts are augmented with a failure detector, these would still not be able to avoid these blocking problems, because this cannot provide information as to at what point during its execution, the coordinator actually crashed.

6.4.2 Three Phase Commit Algorithm

The *Three Phase Commit* algorithm extends its *two phase* counterpart so as to solve the blocking problems. It also replaces the *best effort broadcast*s with *reliable broadcast*s as suggested in the previous section. The *three phase commit* algorithm provides well-defined actions for every possible situation, so that if blocking conditions similar to those of the *two phase commit* arise, then process would be able to determine what action to take.



```
-module(three_phase_commit).

-behaviour(regular_reliable_broadcast).

%% State Definition
-state(
        Replied,    %% Set of processes which can commit in the first phase
        Ackd        %% Set of processes which acknowledged the second phase
      ).

%% Event Handlers
upon event init() ->
        Replied = ∅,
        Ackd = ∅.

upon event commit() ->
        trigger regular_reliable_broadcast:broadcast(query_to_commit).

upon event received({reply_yes, From}) ->
        Replied = Replied ∪ {From}.

upon event received({reply_no, From}) ∨ (timedout phase 1) ->
        trigger regular_reliable_broadcast:broadcast(abort).

upon Replied == Π ->
        trigger regular_reliable_broadcast:broadcast(prepare).

upon event received({ack, From}) ->
        Ackd = Ackd ∪ {From}.

upon event (timedout phase 2) ->
        trigger regular_reliable_broadcast:broadcast(abort).

upon Ackd == Π ->
        trigger regular_reliable_broadcast:broadcast(commit).
```

Figure 6.4: **The coordinator role of the Three Phase commit algorithm**

The listing in figure 6.4, gives the coordinator part of the *three phase commit* algorithm. The first two phase is very much identical to that of the *two phase commit* algorithm except for the usage of the *regular reliable broadcast*. Also, another set *Ackd* is locally kept by the coordinator. This set is used to determine when the coordinator should move on to the third phase. When in the second phase, the coordinator sends out a *prepare* message.

The actual difference between the *two phase* and *three phase* commit algorithms is this *prepare* message. Consider a variant of the two phase commit algorithm which uses *regular reliable* broadcast instead of *best effort broadcast.* Such a variant would still be incorrect, even if there are strictly defined timing boundaries for the maximum time taken for the reply from the coordinator to be received. Note that if the coordinator fails in the middle of the broadcast, but manages to send *commit* messages to some of the cohorts, then these processes will now *relay* these messages to all other cohorts.

Since for this algorithm, a synchronous system is being assumed, a boundary for receiving a message through a relay process can be established. However, this still does not solve the problem. Considering the case where the coordinator manages to send the *commit* message to just one cohort process and then crashes, it could happen that this cohort also crashes before relaying the message to the other processes. However, it could have managed to commit the action before crashing. Since the only relay process has crashed, the rest of the processes would not be informed to commit, and would remain blocked. If these decide to abort the operation after that the operation timesout, there would not be agreement with the crashed process (since this actually committed). It is not difficult to see that this is the same problem solved by the *Uniform Broadcast* protocol. In fact, if a *uniform reliable broadcast* is used in the *two phase commit*, the blocking problem would be solved. Essentially, this is what the *three phase commit* is implicitly doing. The *prepare* phase serves to turn the broadcast of the *commit* message, into a uniform broadcast. All processes which receive a prepare message, will ”promise to commit” even if these do not receive any message from the coordinator in the third phase. On the other hand, if processes do not receive this *prepare* message, their default action will be to abort the operation.

The coordinator waits to receive acknowledges from all processes, and when this happens it moves to the third phase and sends out the *commit* message (lines 34-35). Note that the coordinator will send abort messages if there is a timeout (coordinator does not receive a reply or acknowledge from a cohort in the second and third phases) or in the case that it receives a negative reply (lines 22-23) in the first phase.

Note that in the second phase, cohorts can only send *ack* messages. Cohorts can only prevent the operation from taking place by sending a *no_reply* in the first phase. If the server does not receive an *ack* from some process, it times out in phase 2 and sends an *abort* message (lines 31-32). Hence, once the coordinator is in the second phase, the operation to be committed can only be *abort*ed if the coordinator does not receive *ack* from some cohort (is detected to have crashed by the coordinator). If the coordinator crashes without sending out *commit* messages, the operation will still be committed by all cohorts as soon as the second phase times out.

```
37 upon event rrb_deliver(From, query_to_commit) ->
38         Result = callback can_commit(),
39         case Result of
40                 true->
41                         From ! {reply_yes, self()};
42
43                 false->
44                         From ! {reply_no, self()}
45         end.
46
47 upon event rrb_deliver(From, prepare) ->
48         From ! {ack, self()}.
49
50 upon event rrb_deliver(From, commit) ∧ (timedout phase 2)   ->
51         callback decide(1).
52
53 upon event rrb_deliver(From, abort) ∧ (timedout phase 1) ->
54         callback decide(0).
```

Figure 6.5: **The cohort role of the Three phase commit algorithm**

The listing in figure 6.5, gives the cohort part of the *three phase commit* algorithm. There is minor difference from the *two phase commit* algorithm. The difference is the use of the *regular reliable* broadcast, as explained earlier. The cohort processes also send *ack* messages to the coordinator after receiving the *prepare* message.

Moreover, it should be noted that there are well defined actions to be taken when phase 1 and phase 2 timeout (the next expected message is not received from the coordinator within some time boundary). In case the cohort timeout while in phase 1 (no *prepare* or *abort* message is received from the coordinator), then the operation will be aborted by all cohorts. Conversely, if the cohorts timeout while in phase 2 (no *commit* or *abort* message is received from the coordinator), then the operation will committed by all cohorts. This is expressed using phase timeouts on lines 50 and lines 53.

A failure free run of the *three phase commit* algorithm is given in figure 6.6. In this run, process P_1 is the coordinator. It starts the commit algorithm by reliably broadcasting the *commit* message to all cohorts including itself. All cohorts reply with the result of their *can_commit* callback. In phase 2,

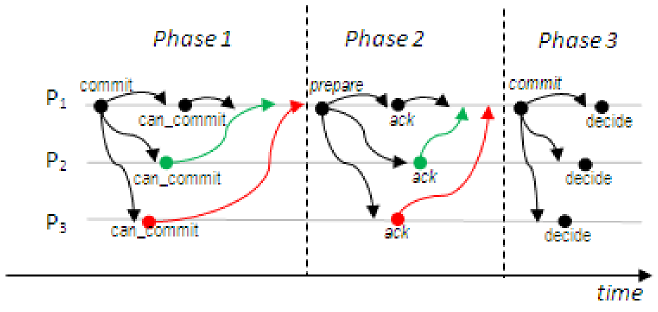


Figure 6.6: **A run of the Three Phase commit algorithm**

assuming all cohorts agreed to commit, the coordinator sends the *prepare* message to all cohorts. All cohorts reply with an *ack* message. At this point, the coordinator moves on to phase 3 and sends *commit* messages to all cohorts. As soon as the cohorts receive this message, a *decide(1)* event is triggered.

6.4.2.1 Erlang Implementation

Despite that in this project, the focus is on algorithm for the asynchronous model, the *Three Phase commit* algorithm was implemented due to its widespread usage. The algorithm utilizes the *sets* module for the internal *Replied* and *Ackd* sets.

A short clarification might be needed on the implementation of the phase timeouts. In the framework presented for the implementation of these algorithms, this can be done by keeping an integer variable in the state, whose value signifies the current phase of the algorithm. As soon as the algorithm enters a new phase, it spawns a process which waits for a fixed time and then casts a message to the local *gen_server* with the value of the phase in which it was triggered. The handler of this cast, would check whether the internal phase is the same as that received in the cast, and if so then the particular timeout action for that phase is carried out. Otherwise, no action is taken.

6.4.2.2 Application in Asynchronous Systems

As mentioned earlier, both the *two phase commit* and *three phase commit* algorithms are relevant for *synchronous* systems. It is rather easy to show that the *three phase commit* algorithm will incur problems in asynchronous systems. This is because the phase timeouts cannot be implemented reliably. As an example, in an asynchronous system, since network delivery times are unbounded, some of the cohorts can timeout during a phase (due to network delays), whereas others do not. If this happens in the last phase, when the coordinator sends *abort* messages, the cohorts which timeout will commit whereas those which manage to receive the *abort* message, will abort the operation.

Surprisingly enough, the *two phase commit* algorithm, achieves correctness even on an asynchronous system. If the cohorts do not impose any receive timeouts (as in the case presented in this project), then the cohorts might still block (if the coordinator crashes), but still do not break correctness since no arbitrary action is taken - none of the processes decides to commit or abort on its own. For this reason, a number of real world implementations make use of the *two phase commit* algorithm despite its blocking problems [8].

6.5 Asynchronous Consensus Based Algorithm

6.5.1 Overview

In the previous section, two algorithms tackling the commit problem in a synchronous environment, were investigated. In this section, an algorithm for asynchronous systems, based on the algorithms in the previous section, is outlined. Similar to the *three phase commit* algorithm, this algorithm is also non-blocking.

The idea behind the algorithm is very similar to a *two phase commit* algorithm, which utilizes a *regular reliable* broadcast for the first phase and a *uniform broadcast* for the second phase. This idea has already been outlined in the previous section, and it was noted that this would be identical to a *three phase commit*. However, this would not suffice for an asynchronous system because if the coordinator crashes in the second phase before sending *abort* or *commit* messages, the cohorts cannot determine whether they will ever receive any of these messages, since in such a system there cannot be a phase timeout. Even when their failure detector determines that the coordinator crashed, these processes cannot take any action (abort or commit) because any of these messages can be on its way on the network, or relayed from some other process.

This problem can be solved by using a *uniform consensus* in the second phase. The idea is that at this stage every process will propose 0 (if it cannot commit) or 1 (if it can commit), to the *uniform consensus.* The decision function of this *uniform consensus* will decide 1, if and only if 0 is not proposed. Once there is a decision, this will indicate whether the processes should commit (decision is 1) or abort (decision is 0).

However, it can be noted that this way, a cohort can crash and fail to propose a value to the underlying consensus, and the consensus would still decide 1 (commit). This is because when the *uniform consensus* algorithm (and any other consensus algorithm) detects a participant failure, it simply does not wait to receive a proposal from that particular participant but still progresses to achieve a decision.

In order to resolve this issue, it should be ensured that all cohorts have a value to propose (indicating whether they can commit or not). This can be done by having cohorts broadcast this proposal to one another before actually proposing it to the *uniform consensus* abstraction. This way, every process first waits to receive a proposal from all other cohorts before proposing to the *consensus.* Moreover, if while waiting for proposals, a cohort detects that another cohort crashed, it would immediately propose 0 to ensure that the commit is aborted.

Now that every cohort is waiting to receive a proposal from all other cohorts, the requirement of doing the *uniform consensus* might be a bit unclear since every cohort would already have all proposals. However, it suffices to notice that whilst one cohort can receive a proposal from all, another cohort might detect that that one cohort failed and not wait for its proposal. Should the decision be taken at this point, the former cohort will commit whilst the latter will abort, violating agreement. This indicates that issuing a *uniform consensus* is still required.

6.5.2 Algorithm

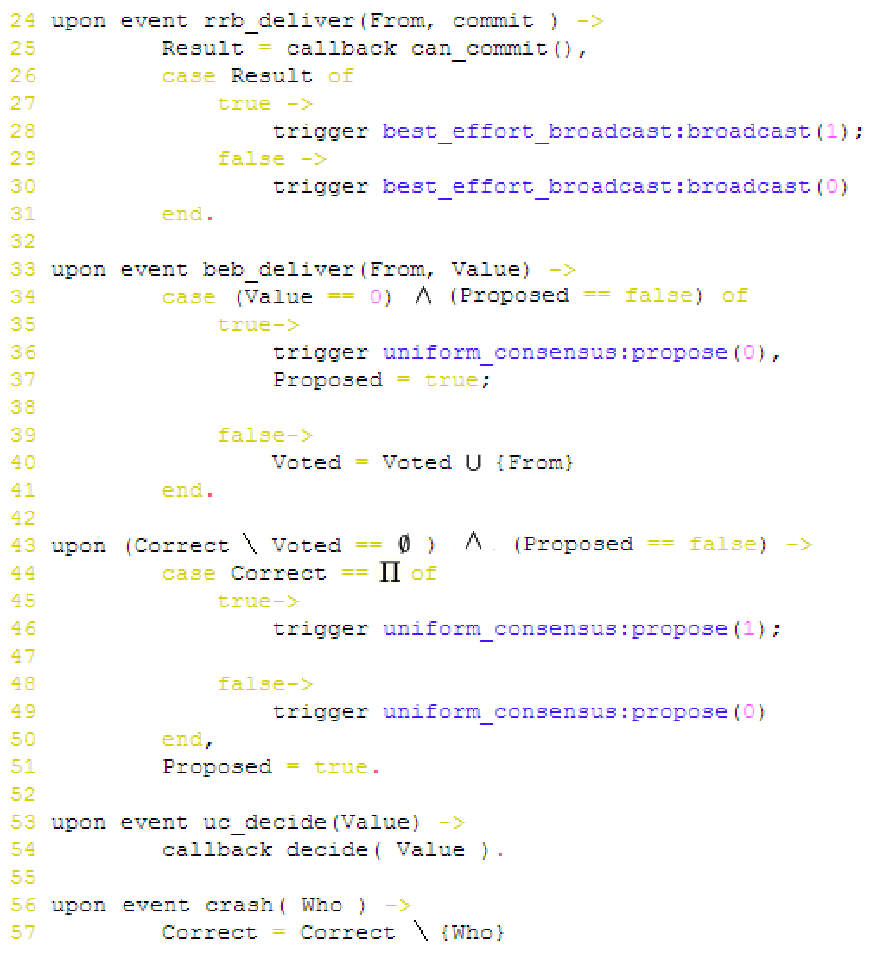
The asynchronous algorithm is divided into the coordinator and cohort roles. Nevertheless, the roles are rather blurred because the coordinator no longer has the role of receiving and orchestrating actions to the cohorts, rather the cohorts act in a choreographic and non-blocking manner. In fact, the only distinction between a cohort and a coordinator is that the latter starts the algorithm (performs the first phase of the *three phase commit*) but does not wait to receive from the cohort - from this point on its role is identical to that of a cohort.

```
-module( consensus_nbac ).

-behaviour(best_effort_broadcast).
-behaviour(regular_reliable_broadcast).
-behaviour(uniform_consensus).
-behaviour(perfect_failure_detector).

%% State Definition
-state(
        Correct,  %% Set of Correct Processes
        Voted,    %% Set of processes whose commit proposal was received
        Proposed, %% Flag indicating whether current process has proposed
      ).

%% Event Handlers
upon event init() ->
        Correct = ∅,
        Voted = ∅,
        Proposed = false,
        uniform_consensus:init( Min ).

upon event commit() ->
        trigger regular_reliable_broadcast:broadcast(commit).
```

Figure 6.7: **The coordinator's role in the consensus based non-blocking atomic commit algorithm**

The listing in figure 6.7 gives the coordinator role in this algorithm. Note that this listing also gives the state definition and initialisation of for the whole algorithm. Internally the algorithm keeps two sets, *Voted* and *Correct*, both used by the cohorts. It also keeps a flag indicating whether the cohort has proposed or not. Note also that the *uniform consensus* algorithm is passed the *Min* function which returns the smallest element of a set - this will make the consensus always decide 0 if it is proposed, otherwise 1.

The only code directly related with the role of the coordinator is the handler of the external *commit* event (lines 22-23). This simply performs a reliable broadcast to start the commit request, by sending a *commit* message.



```
24 upon event rrb_deliver(From, commit ) ->
25         Result = callback can_commit(),
26         case Result of
27             true ->
28                 trigger best_effort_broadcast:broadcast(1);
29             false ->
30                 trigger best_effort_broadcast:broadcast(0)
31         end.
32
33 upon event beb_deliver(From, Value) ->
34         case (Value == 0) ∧ (Proposed == false) of
35             true->
36                 trigger uniform_consensus:propose(0),
37                 Proposed = true;
38
39             false->
40                 Voted = Voted U {From}
41         end.
42
43 upon (Correct \ Voted == ∅ )  ∧ (Proposed == false) ->
44         case Correct == Π of
45             true->
46                 trigger uniform_consensus:propose(1);
47
48             false->
49                 trigger uniform_consensus:propose(0)
50         end,
51         Proposed = true.
52
53 upon event uc_decide(Value) ->
54         callback decide( Value ).
55
56 upon event crash( Who ) ->
57         Correct = Correct \ {Who}
```

Figure 6.8: **The cohort's role in the consensus based non-blocking atomic commit algorithm**

The listing in figure 6.8 gives the cohort role of this algorithm. Lines 24-31 give the handler of the *commit* message which is reliably broadcast by the coordinator. This causes the node to callback the *can_commit* event. The *can_commit* callback should check whether the commit operation can take place and return a boolean indicating this result as explained in section 6.2. After that the *can_commit* callback returns, its result should be broadcast to all other participants, to show the other processes that it has a proposal.

All cohorts will wait to receive these proposals from each other. As soon as this proposal is received, its sender is added to the *Voted* set (line 39). If the proposal received is 0, indicating a request to abort, then the process does not need to wait for further proposals but can immediately propose 0 to the *uniform consensus* - because the commit should be aborted (lines 35-37).

If a proposal to commit (with value 1) is received from all correct processes, and no process has been detected to have failed, then the process will deduce that the operation can be committed, and propose 1 to the *uniform consensus* abstraction. Otherwise, if some process is detected to have failed, then it will propose 0 to the underlying *uniform consensus* (lines 43-51).

When the *uniform consensus* decides with a value, that value is the decision of the commit. This value is returned directly by the *decide* callback of the commit operation.

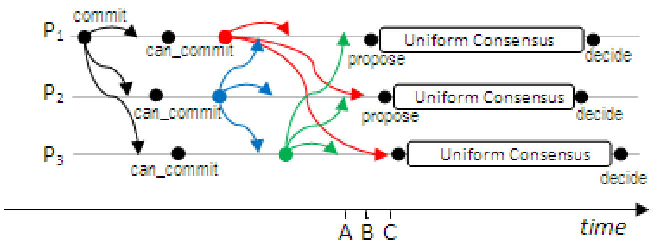


Figure 6.9: **A run of the non-blocking consensus based algorithm of the Atomic Commit specification**

Figure 6.9 gives a run of the consensus based non-blocking atomic commit algorithm. This run consists of three processes, where P_1 is the coordinator. Initially P_1 starts starts a commit request by reliably broadcasting the *commit* message to all processes. When this message is delivered, each process callbacks the *can_commit* event and broadcasts its result through a *best effort broadcast* abstraction. As soon as each process receives these results from all processes, it will propose a value to the *uniform consensus* abstraction. This happens at points A, B and C for processes P_1, P_2 and P_3 respectively. At this point, the processes propose their value to the *Uniform Consensus* abstraction, which eventually decides whether the operation should be committed or aborted.

6.5.3 Erlang Implementation

The consensus based algorithm for non-blocking atomic commit was implemented in Erlang using the techniques outlined throughout this project. The framework outlined for implementing these event based algorithms, was used. The *sets* module was used to implement the *Correct* and *Voted* sets. The algorithms previously implemented were reused trough the use of Erlang behaviours.

6.5.4 Evaluation

The *consensus-based non-blocking atomic commit* algorithm provides a decentralized approach towards the non-blocking atomic commit specification. It satisfies the properties of the *non-blocking atomic commit specification* and guarantees that either all processes commit or abort the operation. It guarantees *uniform agreement* due to the underlying use of the *uniform consensus* abstraction.It guarantees *abort-validity* because all processes first wait to receive the result of the *can_commit* callback from each other. It proposes 0 as soon as it detects a failure or receives 0. The usage of the *Min* as a decision function ensures that if 0 is proposed, it will be the decided value. This decision function also guarantees *commit-validity. Termination* is guaranteed because whilst waiting, eventually processes are either detected as having failed (due to the *completeness* property of the failure detector) or their proposal is received. The *uniform consensus* abstraction also guarantees *termination* hence its usage would not violate the specified *termination* property. Finally, *integrity* is guaranteed also due to the integrity property of the *uniform consensus* abstraction.

In terms of performance evaluation, the algorithm can be split into three parts. First, the transmission of the *commit* message requires the usage of a *regular reliable* broadcast algorithm. In the best case, this requires $|\Pi|$ message exchanges and $|\Pi|^2$ in the worst case. Then each process broadcasts using *best effort broadcast.* This requires $|\Pi|^2$ message exchanges in all. Finally, the *uniform consensus* algorithm is used which requires $|\Pi|^3$ message exchanges. Hence, despite providing some strong guarantees, this algorithm is rather expensive in terms of message exchanges. Due to the initial broadcasts, it also requires at least two more steps than the $|\Pi|$ rounds required by the *uniform consensus.*

6.6 Conclusion

In this chapter, the Atomic Commit specification was investigated. First an analysis of the algorithms tackling this problem in an asynchronous environment, was done. First the *two phase commit* algorithm was studied, and its problems were outlined. Following this the *three phase commit* algorithms was also investigated. Finally, an asynchronous algorithm for Atomic Commit was studied. This algorithm is based on the *uniform consensus* specification.

These algorithms were implemented using the implementation framework outlined for this project. The intricacies as well as problems of some of these algorithms, were outlined. Moreover, any special implementation techniques were also described in this chapter.

Chapter 7

Testing and Case-Study

7.1 Introduction

The last three chapters outlined three different classes of distributed agreements, and assessed various specifications for each class. Algorithms for each of these specifications were described and implemented using the implementation framework outlined earlier. These implemented algorithms form a suite of Erlang modules, in the form of behaviours, which are readily accessible for any distributed application requiring these algorithms.

This chapter will study the testing strategy taken for this project. Testing consists of different phases, each of which will be explained here. In the second part of this chapter, a simple test scenario is outlined. The overall layout of this case-study and, in particular, the usage of the suite of protocols implemented here, is explained. This usage is then evaluated in the following chapter.

7.2 Testing

Throughout the development of any application, testing is a crucial activity for ensuring the quality and correctness of the system being developed. Ideally, testing is well planned and a systematic approach towards system testing is devised. This section will outline the testing approach followed and explain how it was carried out during the implementation of this project.

7.2.1 Test Strategy

In this project, a bottom-up testing approach was followed. Tests were carried out alongside the programming itself. A bottom up strategy starts with testing the individual components of a system and then test at higher levels of abstraction. This approach was taken, so as to be able to carry out the testing of the system in parallel with the coding of the algorithms.

In the context of this project, three levels of testing were performed. These are:

1. **Function Testing**: The testing of individual functions on their own
2. **Module Testing**: Testing a group of functions which together form a module
3. **System Testing**: Testing the distributed system together.

In order to carry out these tests, different tools and techniques were used. The following sections explain these techniques in more detail and present examples of how this testing was carried out.

7.2.2 Function Testing

Function testing encapsulates all tests which are done to assess the robustness of individual functions. In general, sometimes this involves testing a group of two or three functions together due to their cohesive nature, however, these functions together do not achieve some practical distinct task which deserves its own module. Rather, function testing should ensure that every function gives the desired results with all possibilities of input.

In this project, function testing was done in two ways. Firstly, the most trivial form of function testing are tests performed "by hand" immediately after writing a function. Such ad-hoc tests have low coverage and are not really structured, however, these do test that the function works at least works for certain forms of data. Such tests are carried out by directly invoking the function from the shell and testing it with different parameters and their expected outcome.

The second form of function testing was done using Quickcheck. Quickcheck is a tool devised to test Erlang code. It provides different tools for different types of tests. For function testing the generator tool (*eqc_gen*) was used.

This is a tool which automatically generates a large number of testcases, and ensures that the function obeys a particular property.

For example, an *eqc_gen* with a property for testing the conversion of a list to a set, might look like:

```
?FORALL({Values}),
        {list(char())},
        sets:from_list(Values) == sets:from_list(reverse(Values)).
```

This example, creates a generator of string *Values* which comes up with various different strings, focusing on boundary cases ([?]). For each case, it checks that the conversion of a string to a set of characters, will give the same set as when the reversed string is converted.

Quickcheck will attempt to find a counter example for which this property does not hold. When a counter example is found, it will be reported to the user. However, Quickcheck does not stop here. It will attempt to minimize the counter example, so that the user will be able to track where the error lies.

The test specified above, was actually devised for this project. Surprisingly enough, this property is not guaranteed by the *sets* module - ordering affects set equality. Quickcheck provided the following counter example:

```
sets:from_list([a,q])  /= sets:from_list(reverese[a,q])
```

A question about this problem was asked on the well known trapexit.com Erlang forum, and the workaround suggested was used. It turns out that the *sets* module uses a balancing structure which is affected by the order in which elements are added. Hence, equality cannot be checked using the == operator, but rather through the use of other function calls provided by the *sets* module.

The *eqc_gen* test might seem limited to functions about which one can express some form of property. However, to test a function f, one can use a property like $f(Values) == f(Values)$ to ensure that the function does not trigger an exception for all the *Values* generated by this Quickcheck tool. This way, all tests will still run, and if an exception is triggered, it will be reported.

Throughout testing, another tool which was widely used was the Erlang debugger. The Erlang development suite comes with a fully flushed debugger which provides all functionality typically offered by such debuggers, such as single stepping through code, breakpoints and variable inspection. Though this is not a testing technique, this debugger did help a lot tracking down problematic code after function testing, and hence deserved a mention!

7.2.3 Module Testing

A group of function which together perform some specific task, are usually grouped into a module. In this project, every algorithm was written in a separate module, hence, throughout this project, module testing actually tested the functionality of a complete algorithm.

For this form of testing, Quickcheck was again rather crucial. However this time, another tool offer by Quickcheck, was used: the *eqc_statem* tool. This tool tests Erlang processes which internally maintain a state - as is being done by the *gen_server* of every algorithm.

With this tool, a simple state machine is defined to express the behaviour of the process. For example, here a state machine to model the behaviour of the *gen_server* process handling *regular reliable broadcasts* is given.

First, the state of the process will be defined. This state need not be the complete state stored by the process being tested, but just the part which needs to be tested. For this example the state will consist of simply the *Correct* set, which holds the PID's of all correct processes. The state machine will invoke the *broadcast* and *crash* functions with different interleavings, whilst ensuring the correctness of the process.

The state definition and initial states are given below:

```
-record(state, {correct}).
...
      initial_state() ->
          #state{correct=node_utils:get_all_peers() }.
```

This code will initialise the first *Correct* set as having a list with the PID's of all nodes. Following this, the state transitions are defined:

```
    next_state(S, _, {call, r_rb, broadcast,[Data]}) ->
        S;

    next_state(S, _, {call, r_rb, crash, [Pid]} ) ->
```

```
        S#state{ correct = lists:delete(Pid, S#state.correct) }.
```

This *next_state* function defines two transitions, one which takes place when the *broadcast* function is invoked, whilst the other takes place when the *crash* function is invoked. The first clause of the *next_state* function defines the effect of the *broadcast* function call on the internal state of *eqc_statem*. The first parameter, *S*, is the current state. The second parameter is not needed (it holds the return value of the call in the third parameter). The third parameter is a tuple indicating that a call is to be made. The tuple consists of the atom *call*, the module, function name and a list of arguments to the function. Thus, this transition is taken when *r_rb:broadcast(Data)* is called. Note that this function simply returns the state passed, indicating that this call does not affect the internal testing state.

The second clause gives the state transition to be taken when a call to *crash* is made. This call does affect the internal state - if a call to crash is made, the PID of the process which crashed is removed from the internal list of PID's.

At this point, the internal state and how different calls will affect this state, have been defined. What remains is to create a test case generator. This is done by specifying what commands (functions) *eqc_statem* can invoke. These commands are specified by defining the *command* function. A sample *command* function definition is given below:

```
command{S)->
oneof( [{call, r_rb, broadcast, [random_string()]},
{call, r_rb, crash, [random_element(S#state.correct)]}]).
```

The *command* function indicates that the issued test commands should be one of those specified in the list passed to the *oneof* function. This list specifies the two commands which can be invoked: *broadcast* and *crash*. Note that in this definition, generators for the parameters are defined. The *random_string* function should generate random string which are passed as parameters to the *broadcast* function. The *crash* function is passed a random element from the list of PID's in the state.

These snippets are an example of the usage of the *eqc_statem* tool for module testing in this project. Quickcheck's *eqc_statem* tool provides an effective means by which to test the stateful nature of the various algorithms implemented in this project. Whenever a particular sequence of calls triggers and exception, this is reported. Quickcheck will also attempt to ”shrink” this sequence of calls, further simplifying the task of tracing the error itself.

7.2.4 System Testing

System testing attempts to verify the correctness of the complete suite of algorithms implemented here. Though the module tests did at times involve a number of modules, since certain modules cannot operate without others, system testing tests the complete system of modules altogether.

The system of processes used for the algorithms here were first tested in a localised environment (on the same machine). In Erlang, code can be written in a way so that is requires minimal change when deployed on a localized or distributed environment. This can be achieved because Erlang's PID's hold information about the node where the process is located. If the *global* module is used, almost no code change would be required, because processes can be reference by using globally (anywhere in the whole distributed system) registered names.

Hence, following this localized approach, the system was tested in real distributed environment. The deployment in a distributed system, makes the system prone to different forms of failures which are not present in a localised environment. Throughout testing, since crash-failures were assumed, a node failure could be simulated by simply closing the Erlang shell for the system of processes on that node. Nevertheless, with such an approach, certain conditions could be very difficult to be simulated. For example, the algorithm for *regular reliable broadcast* should ensure, that if a sender crashed after broadcasting to just one of its peers, that peer should be able to relay the message to all other processes, when it detects that the sender crashed.

Crashing the sender exactly after sending to only one other process is rather difficult to do . Thus, for the purpose of testing, a number of delays were inserted into such points in code. For example, a delay was inserted after sending a message to a process. This gives time to close the shell of the sender, exactly after having sent to just one process.

Finally, testing and debugging a system which is composed of remotely located machines, can require a rather strenuous effort to keep hold of the data being output on the different locations. However, the *node_diag* module, provides a means by which to start distributed nodes from one shell (refer appendix A). This is achieved through the use of Erlang's *rpc* module. A particular characteristic of Erlang, is that a process which is created remotely, will have its standard output automatically redirected to the remote shell from where it was created (from where the rpc was issued). This provides a centralized way to "debug" distributed processes. This way, the debug data output by remote processes will all be sent to the shell which started them (which is referred to as the "monitor shell"). When outputting debug data, every process precedes this data with the name of the node from were it is originating, so that one can keep track from where the debug data on the monitor shell, is originating.

7.2.5 Conclusion

This section outlined the testing approach taken throughout this project. Moreover, the various tools and techniques used, which helped in testing and debugging, were outlined. The next section will outline the design of a real-world application for the algorithms implemented. Apart from serving as a test case for a number of the algorithms implemented, it will help to first handedly assess the benefits achieved from utilizing the implemented distributed protocol suite.

7.3 Case-Study

7.3.1 Overview

In the Background chapter, various examples usages of the agreement algorithms implemented here, were presented. Here, the basic semantics of the serverless distributed file system *p2pfs* were informally described. In this section, this example will be studied in detail, and an implementation making use of the protocol suite implemented for this project, is described.

7.3.2 Description of the p2pfs filesystem

p2pfs is a distributed file system with fault tolerant characteristics. *p2pfs* operates in a peer-to-peer basis, where every distributed node coordinates with others to ensure that the local view of the filesystem is consistent with that of the other nodes. This file system is assumed to operate in a system model identical to that assumed for the development of the agreement algorithms implemented in this project. It works in an asynchronous environment and assumes fail-stop node failures - this implementation will focus on the graceful degradation as different nodes fail.

For simplicity's sake, this file system is assumed to consist of just one root directory with no subdirectories. Files can be created, accessed and deleted from within this directory and all these changes should be reflected on all participating nodes.

A filesystem, typically handles a number of calls which are issued by the operating system. The filesystem implemented here will only handle a small subset of these calls. The calls handled by the filesystem implemented here are:

1. **open**
   This is a call to open a file. It is issued by a process which needs to access a file, before performing any reading or writing of the file.

2. **read**
   This call reads data from the file.

3. **write**
   This call writes data to the file. All data written should be reflected at all distributed nodes.

4. **close**
   Closes the file, and terminates the current access to a file.

5. **unlink**
   Deletes a file from the filesystem. The file should be deleted from all nodes, if not currently open by some node.

6. **mount**
   Mounts the file system at a particular directory (called *operating directory*). If before mounting, this directory contains existing files, these will be sent to all other nodes.

7.3.3 Assumptions and Considerations

Prior to implementation, a number of issues regarding the semantics of the proposed filesystem, had to be taken into consideration. Moreover, to simplify this test-case, a number of assumptions also had to be made.

Firstly, one should note that when *mount*ing the file system, different nodes may have files with the same name already present in their operating directory. According to the semantics outlined above, these files should be sent to all other nodes. However, since there cannot be two files with the same name in the same directory, only one version of the file is to be kept. However, care should be taken to ensure that all nodes keep the same version of the file. So as to ensure this, it is assumed that the most recent version of the file should be kept. Note that though the clocks of the nodes are not assumed to be synchronized, this would still ensure that the same version of the file is kept by all nodes. In the rare case that two files have the same timestamp, the file whose md5 hash is greater than the other, is kept.

In practice, first a call to *mount* is made, specifying the directory where the file system should be mounted. Following this, any of the other operations may be requested. However, it is assumed that the filesystem is first mounted at all nodes, before any of the other operations are issued.

It should also be noted that due to the nature of the test-case, a number of race conditions could arise. For example, if a node issues a read on a file, does some other work, and then issues another read on the same file, it could be that data was written to the file in the meantime. This would lead to the second call to be reading from a different version of the file. This situation also happens in a localized file system, if two processes open the same file and save at different times, these would be overwriting eachother's work. This can be solved by using file locks, however for this scenario, this would result in distributed file locks. These are beyond the scope of this work, and are not implemented.

Finally, it should be noted that the operating system may issue two or more concurrent calls to the file system. Since these two calls could be possibly operating on the same file, if a write occurs whiilst a read is taking place, this could cause the read to return inconsistent data (data from two versions). For this reason, though the operating system may issue concurrent calls, these calls are serialized by the filesystem and handled one after the other. This would ensure that one read, would always return data from one consistent version of the file.

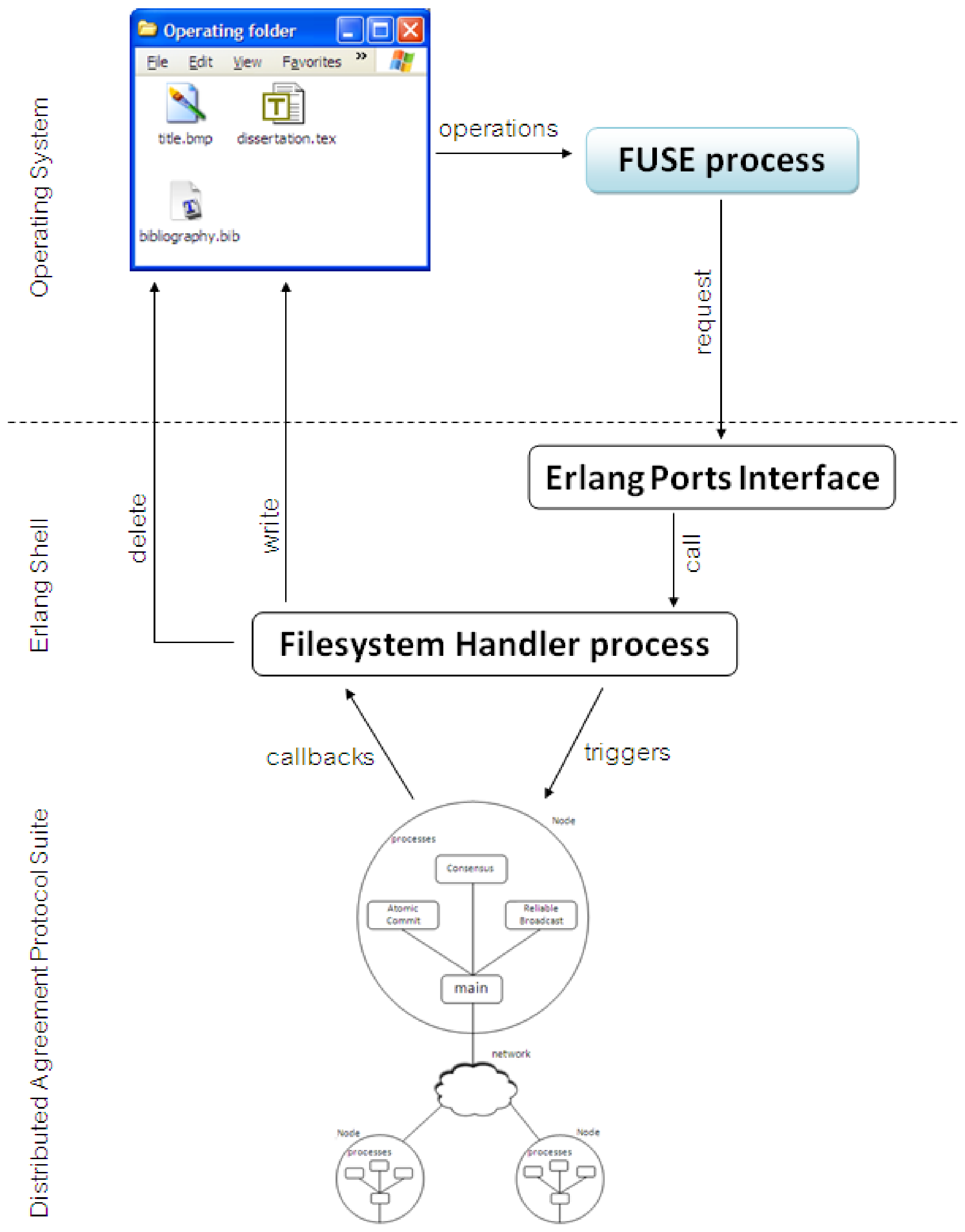


Figure 7.1: **Organisational Layout of the Test Case**

7.4 Implementation

The organisational layout for the implementation of the proposed test case, is given in figure 7.1. It shows the interaction of different processes and components, all of which will be explained in more detail in this section.

At the topmost level, is the Operating System. This is the "User's View" of the filesystem, where files are created, accessed and deleted using application software such as text editors, image editors and so on. Whenever each of these software applications access any of the files in the filesystem, this triggers a system call which typically handles the call being requested (such as reading and writing). Our testcase application needs to be able to capture such requests and handles them - example when a write is done, this is sent to all other nodes. However, these calls are handled by the kernel level and hence it is typically very difficult to program a filesystem from scratch.

For this purpose, FUSE was used. FUSE (File System in Userspace) intercepts the file system operations and lets them be handled by specially programmed application (the *FUSE process*). In this test-case, this is exactly what is needed, since operations such as writing to a file, will be intercepted and broadcast to all other nodes.

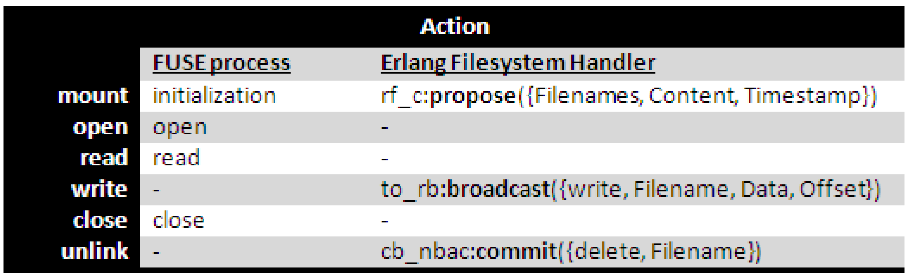
Hence the *FUSE process* will have handlers for the file system calls specified. The problem is that the *FUSE process* has to be written in C, not in Erlang. However, Erlang has a mechanism by which to interface to external programs. This is called the *Ports Interface*, which allows data to be sent and received to and from an external program. The external application needs to encode this data in a special way (which is done by C libraries which come with Erlang). The *Ports Interface* process will then decode this data and take the necessary actions. In this case, the external application with which the Erlang code is going to be interfacing with, is the *FUSE Process*. The *FUSE Process* will send information about the operation requested by the Operating System, to the *Ports Interface process.*

The *Ports Interface* process will decode this information, and pass the requested call to the *Filesystem Handler* Erlang process. The *Filesystem Handler* is a process which receives external calls and handles them sequentially. It is based on the *gen_server* behaviour, with which was used extensively in this project. Hence, for example, when the *Port Interface* makes a call to the *Filesystem Handler* process, to indicate that a write was made, the latter will broadcast this write to the other nodes.

the *Filesystem Handler* process will make direct use of the distributed agreement protocol suite implemented for this project. It will trigger events and handle the callback events of the distributed agreement protocol suite. As an example, when the operating system writes to a file, a broadcast event should be triggered. On the other hand, when some other node writes to a file, this node will receive a write message from that other node through the *deliver* callback of the broadcast mechanism being used.

The callbacks will indicate that the current node should update its view of the filesystem. In this filesystem, such updates can be reduced to deleting a file and writing data to a file. Both the deletion and writing data to a local file, can be handled from within an Erlang shell, through the use of the *files* OTP module.

In reality, not all file system operations require interaction with other nodes. Some of the operations only require a local action. These operations can be handled solely by the *fuse process* process. The table in figure 7.2, gives the actions required to be taken by the *FUSE process* and *filesystem handler*, for all the filesystem calls.



	Action	
	FUSE process	Erlang Filesystem Handler
mount	initialization	rf_c:**propose**({Filenames, Content, Timestamp})
open	open	-
read	read	-
write	-	to_rb:**broadcast**({write, Filename, Data, Offset})
close	close	-
unlink	-	cb_nbac:**commit**({delete, Filename})

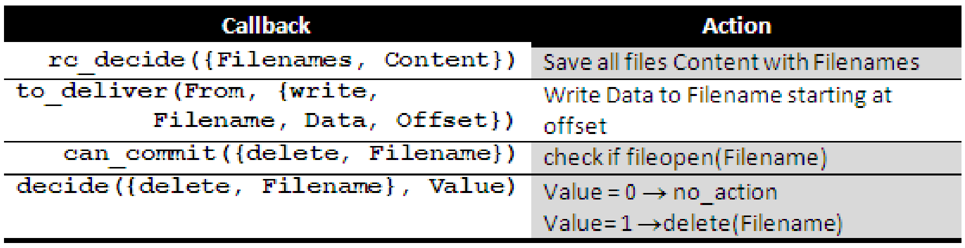
Figure 7.2: **Actions to be taken for all filesystem operations**

As can be seen from the table in figure 7.2, the *mount* operation requires the *FUSE process* to perform some initialisation to start the file system. This is a standard initialisation required by all FUSE filesystems. As mentioned, the *FUSE process* will notify the *filesystem handler* about this mount. Moreover, it has been outlined, that initially any files already present in the *Operating Directory*, need to be sent to all other nodes. Moreover, there was the issue of identical filenames, which needed to be solved by keeping the versions which have the latest timestamp. Hence, the *filesystem handler* needs to use a *regular flooding consensus* in order to decide which files to use. Thus, as an action for the mount call, this process proposes all the filenames, file contents and timestamps (as three lists). A flooding consensus is required, because all files need to be seen before taking a decision. The decision function should choose the files which have the latest timestamp.

As can be seen in figure 7.2, the *open*, *read* and *close* events do not require any action to be taken by the *filesystem handler* process. In fact, these can invoke the usual calls for carrying these operations locally.

As has been mentioned, the *write* call needs to be handled by some form of reliable broadcast. It could be noted, that unless a total ordering is ensured, different nodes will possibly end with different versions of the file, after that all operations have completed. Hence, a *total order broadcast* algorithm will be used for broadcasting the filename and data which needs to be written at a particular offset. When this message is delivered by a remote node, it will write this data to the specified file, at the specified offset.

Finally the last operation is the file deletion or the *unlink* call. This requires to first check that the file is not open by all nodes, and if so, deleted from all nodes. If however, the file is open at some node, the delete operation should not be carried out by any node. Such an operation can be easily solved through an atomic commit. Hence, the *filesystem handler* issues an call to a *consensus based non-blocking atomic commit*. The *can_commit* callback event will check whether the file is open at that node, and if the file is not open, then the file will be deleted.



Callback	Action
`rc_decide({Filenames, Content})`	Save all files Content with Filenames
`to_deliver(From, {write, Filename, Data, Offset})`	Write Data to Filename starting at offset
`can_commit({delete, Filename})`	check if fileopen(Filename)

`decide({delete, Filename}, Value)`	Value = 0 → no_action
	Value=1 →delete(Filename)

Figure 7.3: **Action taken for callbacks triggered by protocol suite**

The table in figure 7.3 gives the actions to be carried out for every callback event received by the *filesystem handler* process. The first callback, the *rc_decide* is invoked when the run of *regular consensus* chooses a list of initial filenames and their content. These files should be written into the operating folder.

The *to_deliver* callback indicates that some other node has issued a write on a file. Hence the action for this callback is to write the data, at the specified offest into the requested file.

The last two callbacks, *can_commit* and *decide* deal with the *consensus based commit* issued to determine whether a file can be deleted or not. The handler of the *can_commit* callback, should check whether the file is open by the local node and return a boolean indicating whether it is open or not. The *decide* callback will indicate whether the file should be deleted - a value of 1 indicates that the file should be deleted.

This way, the test case scenario can be built. This case study involves various components, but these all revolve around the distributed agreement protocol suite implemented in this project.

7.5 Conclusion

This chapter presented different ways for assessing the distributed agreement protocol suite. In the first part, different types of testing done throughout development, were outlined. In the second part, a real world test environment, involving a serverless distributed filesystem, was described and implemented.

Chapter 8

Evaluation

8.1 Overview

This chapters attempts to assess various aspects of this project. The implementation of the testcase scenario, gives an insight on the applicability of the distributed suite of agreement protocols. This chapter attempts to outline most of these considerations. The benefits and challenges faced are evaluated and the applicability and usability of the distributed agreement protocols is investigated. Their actual benefit for distributed programming is also analysed.

8.2 Benefits Achieved

The distributed agreement protocol suite, played a major role in the development of the filesystem. Approaching the development of the filesystem, with this library at hand, provides various advantages to having to build everything from scratch.

8.2.1 Separation of Concerns

This project involved analyzing commonly encountered distributed computing problems. These problem are widely known and have already been studied extensively. The solutions to these problems, however, are rather intricate and require a deep understanding of the difficulties which arise when attempting to give solutions to these problems. Primarily, alternative approached to solving distributed agreement problems are analysed.

This project pinpoints these problems by giving counter-examples of runs which would cause the ultimate goal not be reached (example not achieving agreement in a run of consensus) and leave the system in an inconsistent state. It could be noted, that these intricacies are sometimes rather subtle to note, because of the complicate sequence of events which trigger them, yet when they occur they could cause different kinds of problems. Doing distributed programming requires a clear understanding of the application domain and of the intricacies associated with distributed systems. The former will change from application to application, but the latter are common to all forms of distributed systems. The protocol suite implemented in this work, relieves the programmer from having to deal with these distributed system caveats. Whilst the programmer concentrates on the application logic, he would utilize the adequate algorithms from the suite, which unseeingly handle the intricacies incurred. At run time, the algorithms in the implemented protocol suite, act as orchestrators for the application code triggering parts of the users code at designated points in the algorithm.

8.2.2 Shorten development time

Utilizing this library, first of all helped developing this application in fraction of the time, had it not been available. Coming up with fault tolerant code, is rather hard since one generally needs to cater for a large number of possible interleavings. This makes coding such systems rather hard and non-straight forward. Having this protocol suite readily available for the development of the filesystem, helped in modelling the system in terms of the functionality provided by this suite. This relieves having to devise the distributed agreement algorithms themselves, but rather the programmer focuses on writing interfacing codes (or wrappers) for triggering events in the existing algorithms and handling events these callback.

8.2.3 Structured Code

This suite of distributed protocols also provides fault tolerant guarantees. Attempting to achieve these fault tolerant characteristics without this suite, would have required us to constantly issuing to write checks to ensure that a consistent state has been maintained after every operation. Undoubtedly, there is a high possibility of these checks not being correct, in that these do not cater for all the exceptional cases. But apart from this, the code would lack clarity and be less structured. This is because, "traditional" techniques for checking whether a call has succeeded would need to taken.

For example, the typical approach is to wrap every call which attempts to receive from the network, with timeouts. Moreover, all functions which attempt to send, will need to incorporate some logic to ensure that this sending completed successfully. Having all this code, interspersed with the application logic code, makes the latter less and less easy to follow.

In the code written for the filesystem testcase, no concern had to be expressed about node failures. On the contrary, the distributed algorithms in the suite handle failures on their own. For example, the application code always issues a broadcast in the same way, despite the number of failed or correct nodes - it is the internal code of the broadcast algorithm, which handles sending to just the correct nodes.

8.2.4 Agreement Algorithm Abstractions

Moreover, the concept of abstraction applies when using these algorithms. A programmer using these algorithm does not need to know how these work internally, and all the intricacies these try to address. However, had the filesystem been built from scratch, the programmer would have needed to understand these problems in their totality, and make sure that all effects are catered for. However, with this suite of algorithms, it suffices for the programmer to simply know the basic semantics of the algorithm and its interface, in order to produce a correct piece of code.

8.2.5 Code Reuse

The suite of algorithms aids in putting code reuse to practice. When developing a distributed application with such a suite of algorithm, reusing the existing algorithm helps in reducing the size of the code and making it more maintainable. Code reuse is also a key aspect within the coding of the suite itself - most algorithms make use of one another, and various Erlang/OTP module are also used.

8.2.6 Simplifies testing

The development of code with this suite, also helped reduce on testing. Testing distributed code also poses a challenge and requires careful planning and analysis. Using a library which has been already rigorously tested, definitely gives an advantage over writing all distributed agreement algorithms from scratch, and having to test these algorithms as well. Such a suite will help reduce the amount of bugs and hence also reduce the time taken to track down and fix these bugs.

8.2.7 Conclusion

Finally, it should be noted that thought the programmer does not need the internal details, he need to be very well acquainted with the properties of these algorithms. Moreover, just because a reliable algorithm is used, it does not mean that the application using it will automatically inherit all its properties. The actions done by the application code itself, also play a great role in the final properties of the application. However, ultimately, it should be noted that given that care is taken to choose the right abstraction, and that all interactions with other node, is made through these abstractions, then the final code will be much easier to write, whilst providing the correctness ensured by the underlying algorithm.

8.3 Assessing the Implementation Framework

In this project, an implementation framework was devised - in which all agreement algorithms were then implemented. This framework provides a common ground for developing all agreement algorithm in Erlang. Here the effectiveness of this framework is evaluated, with respect to implementing the standard algorithm into this framework.

8.3.1 Bridging Theory and Practice

This framework provides a standard way by which to implement algorithm which are rather theoretical. For example, these algorithms expect the existence of a perfect failure detector to guarantee correctness. Part of this framework, consisted of extending Erlang's built-in failure detection mechanism to match the properties of the failure detector expected here, as much as possible. Moreover, through the use of the *gen_server*, it provides ways by which to give event-like semantics to all calls - where function are handled sequentially. Another example, is of the standard way to implement the *predicate triggered* events.

8.3.2 Code Reuse

This framework makes the implemented algorithms easily reusable. Each algorithm has a well defined interface by which its functionality can be accessed. The callbacks of every algorithms are specified in the form of Erlang behaviours, hence the user is allowed to input "custom" code into the algorithm, whilst reusing the existing code of the algorithm itself. Moreover, internally these algorithms also make use of one another through this technique of behaviours.

8.3.3 Structured and Maintainable Code

All algorithms implemented have a similar structure. This makes it easier to understand and follow the code. Moreover, internally these algorithms are modelled in the form of *gen_server* behaviours. Apart that, the *gen_server* provided part of the implementation semantics required, it also is a very commonly used and well documented Erlang behaviour. Hence this makes the code more structured and more maintainable.

8.4 Evaluating the usage of Erlang

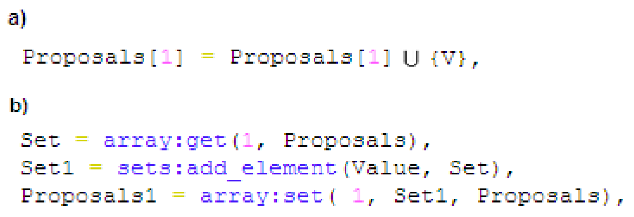
Having implementing this suite of algorithms in Erlang, the applicability of Erlang can be evaluated from a better standpoint. In this section, a "post-mortem" analysis of the relevance of Erlang and the benefits it provided, is presented.

Firstly the Erlang's support for both localized (concurrent) and distributed programming should be mentioned. This helped structure the testing of this project. The algorithms implemented, were first tested locally and then eventually in a distributed environment, with minimal code change.

This project really availed itself of the OTP libraries which come with Erlang. As explained, all the algorithms used various of built in modules for some data structure - such as the *sets*, *Array* and *Dict*. Moreover, Erlang also provides various libraries which facilitate distributed programming such as the *global* module (for distributed process name registration). Furthermore, the implementation of the algorithms in this project, also make extensive use of the *gen_server* behaviour. Of course, similar libraries could be written (or used if they already exist) in other programming languages. However, given that all these are standard modules in Erlang, these would help in making the code more maintainable and facilitates future modifications.

Certain aspects of the Erlang syntax were found very helpful during the implementation of this project. For example, Erlang abstracts from the "complexities" associated with distributed communication such as byte ordering, marshalling and . In fact, the communication constructs are as simple as those found in pseudo-code, or in process calculi.

One aspect which was not really felt, was the significance of the single assignment, with regards to producing correct code. It is claimed that this helps tracking down bugs, because only a single action could have assigned a value to a variable [4]. However, this particular benefits was not really felt. Furthermore, this made the code rather obscure at time. For example, figure 8.1a gives a line taken from the flooding consensus algorithm, as presented here. Its actual implementation in erlang (figure 8.1b), takes three lines, and is much less readable.



a)

```
Proposals[1] = Proposals[1] U {V},
```

b)

```
Set = array:get(1, Proposals),
Set1 = sets:add_element(Value, Set),
Proposals1 = array:set( 1, Set1, Proposals),
```

Figure 8.1: **a) line 29 of the flooding consensus algorithm. b) Its actual implementation in Erlang**

Erlang's pattern matching feature, was used extensively. For example, all *handle_casts* make use of pattern matching for to choose which handler to invoke for each particular message type. Together with pattern matching, guards were also occasionally used. These two constructs really did help improving the readability of code.

Another effective feature of functional programming used throughout this project are Higher Order Functions. These were used for the decision function of consensus algorithms.

Finally, the lack of strict typing in Erlang, was also seen as an advantage in itself. This is because it provides a means by which to write generic functions which are not bound to a particular data type. So for example, once a broadcast function was written, this would work with any form of data be it numeric, a list or even a high order function.

8.5 How practical are the algorithms?

All the algorithms implemented here are based on theoretical algorithms - each utilizing some various forms of mathematical constructs which might require extra work to be implemented using the tools provided in conventional programming languages. With Erlang being a functional language, this work is reduced significantly since it provides native and library constructs to implement mathematical constructs such as sets, maps and the notion of higher order functions. It also provides theory-like abstractions for distributed communication such as simple communication primitives, and process labelling. All this reduces the leap required from theory to practice.

Nevertheless, in theory usually one tends to prioritize correctness before performance requirements concerns. Whilst it is true that Premature optimization is the root of all evil (Donald Knuth), a correct algorithm which requires an unreasonable amount of memory to operate, exchanges enough messages to hog most networks and takes forever to complete, is very much likely to be scrapped in real life.

8.5.1 Network Requirement

Here it is important to distinguish between worst case scenarios and normal or real world scenarios. Starting from broadcast algorithms, we see that most algorithms require from $|\Pi|$ to $|\Pi|^2$ message exchanges. The lazy broadcast algorithm for Regular Reliable Broadcast, has a best case which is the lower bound for broadcast algorithms, $|\Pi|$. Its worst case scenario is when all process fail in sequence (very rare in practice) and would now require $|\Pi|^2$ message exchanges. It is noted the properties of the broadcast algorithm are strengthened, the amount of messages required increase. This is, in fact, what happens in the case of the All-Ack algorithm for the Uniform Broadcast abstraction which suddenly has an average of $|\Pi|$ message exchanges the price for strengthening the agreement from just between correct process to all participants (both correct and faulty).

Similar behaviour can be seen also with the consensus algorithms. In the case of Regular Consensus, three algorithms were presented, each having their own characteristics and network requirements. The flooding algorithm is the most network demanding (requiring $2|\Pi|^2|$ message exchanges in a run without failures), but in turn, is the only one in which the decision depends on all proposals. The Hierarchical consensus algorithm, reduces the number of messages exchanges of the flooding consensus algorithm by half, however the decision is imposed by one of the participants without considering all other proposals first.

The algorithm for the total order broadcast uses an existing regular consensus algorithm and a regular reliable broadcast algorithm. Thus, consensus-based algorithm for the total order broadcast exchanges all the messages exchanged by these underlying algorithms, as well as its own set of messages (which turn out to be $|\Pi|$). Similarly, the consensus based algorithm for the non-blocking atomic commit, utilizes a uniform consensus algorithm. Hence, this algorithm ultimately exchanges all the messages exchanged by the uniform consensus algorithm, as well as another $|\Pi^2|$ messages. This highlights the importance, that in practice one should choose the just right protocol, and not use anything stronger than required. This is because, whilst not affecting correctness, it would introduce unnecessary overheads, which apart from increasing the network requirement, also affect the runtime performance of the algorithm.

So far, the primary network requirement concern has been the amount of message exchanges. The actual size of the message was always dependent, for the most part, on the size of the data being broadcasted by the user. However, the Causal Order Broadcast algorithm also leaves one pondering about the actual size of the messages exchanged. Without any optimisations, as time progressed, this algorithm transferred message of ever increasing sizes. In this case, these ever growing messages were catered for, by using a garbage collection scheme to start pruning content of the messages exchanged. However, as noted, this is mechanism introduces new message exchanges which could increase the load on the network. Moreover, under conditions of heavy network load network delays would cause this mechanism to slow down and hence larger messages could be relayed on the network worsening the delays!

8.5.2 Memory Requirement

In the presentation of the algorithms given here, a lot of concern was expressed regarding the memory requirement of certain algorithms. It is noted, that in the implementation of these algorithms, as soon as some structure is no longer needed, it is removed from the recursive state (held by the gen_server behaviour). This would help relieve certain algorithms from the amount of memory required. However, there are cases where it is not possible to free certain structure in a way that these remain consuming memory for ever. An example of such structures are sets holding all the messages which were delivered (such as the *Delivered* set in the regular reliable broadcast). It is not possible to do away with these structures, even when the algorithm terminates, due to the asynchrony of the system.

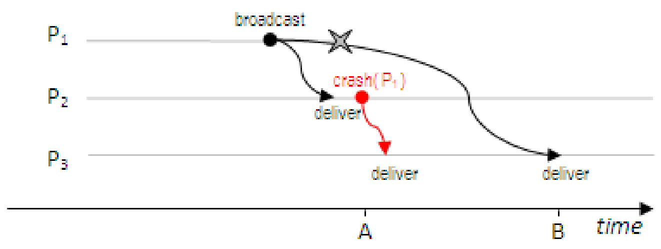


Figure 8.2: **Incorrect double delivery, due to freeing the internal *Deliver* set**

Consider the run of the lazy algorithm for regular reliable broadcast in figure 8.2. Here, process P_1 manages to send a message to P_2 and P_3, but crashes shortly afterwards. Process P_2 receives and delivers this message. At point A, it also detects that P_1 crashed, and hence relays this message to P_3. Process P_3 receives this message and delivers it. However, at point B, the original message from process P_1 arrives and unless there is a set with all delivered messages, P_3 would end up delivering the same message again, as shown. Such structures pose a limit on the practical side of the algorithm. In order to solve this problem, one would require to devise higher algorithms which would ensure that the removal of elements from such structures, would not result in inconsistent behaviour such as multiple deliveries. Due to time constraints, in this project this problem could not be solved to its totality, however a possible solution is mentioned as part of the future work.

8.6 Assessing the failure detector implemented

In this project, we chose to work on top of Erlangs native failure detection mechanism, to build a perfect failure detector. The field of failure detectors is a research area on its own there are various algorithms and important results which would need to be studied before implementing a failure detector. For this work, it was decided to keep focus on the algorithms being investigated and analyse these algorithms in future work. After all, the role of the failure detector abstraction is this to help separate process monitoring from the algorithm itself. However, most of the algorithms implemented here require a perfect failure detector in order to guarantee correctness. This means that the failure detector should satisfy both the Completeness and Accuracy properties (see section). The Erlang failure detection mechanism utilizes a sheer heartbeat mechanism if a heartbeat is not received from a process, then that process is taken as crashed. This approach is rather problematic, because network delays could cause the failure detector to think that a process has crashed, when in fact it would still be running.

At this point, a simple procedure to attempt achieving a perfect failure detector was taken: as soon as a failure detector suspects a process, that process is sent a message which forces it to crash. This would strengthen the accuracy, since a suspected process would now truly crash. Note that since the process will actually crash after that it is detected, it would still continue its operation until it receives the message to crash possibly even sending out further messages. However, since the system is asynchronous, and the failure detector is also asynchronous, such behaviour could be allowed similar to having message reordering, a failure detector can always be fast enough to detect a failure, before all pending messages are received.

However such a perfect detector is not truly perfect. There could be cases where it could break the underlying algorithm. A particular example was mentioned in section 5.5.6. This happens because the process which is detected as having failed, carries on in its execution, and ends up taking conflicting actions which violate the properties of the algorithm.

This approach of constructing a Perfect Failure Detector is in reality just a work around - since a failure detector operating on a busy network, would cause it to kill other processes due to the network delays. In such networks, it would be difficult to build a perfect failure detector, and usually one uses some form of adaptable timeout mechanism to construct eventually perfect failure detectors. However most algorithms described in this work, only work with perfect failure detectors. Using a weaker failure detector, generally breaks the algorithm. However, there might be cases where with some minor changes, the algorithm would still work fine with weaker failure detectors. Example, in the case of lazy reliable broadcast algorithm, suspecting a process would cause all messages received from that process to be rebroadcast. Hence as long as this broadcast also attempts to send to the suspected processes, an eventually perfect failure detector can be used. Nevertheless, such a failure detector could also cause the algorithm not to reach its goal, example a consensus algorithm would not reach agreement. In fact, it can be seen that the flooding consensus algorithm might not decide if processes are suspected as crashed, but have not really crashed. The same happens with the hierarchical consensus algorithm.

8.7 Conclusion

This chapter evaluated different aspects dealing with this project. Primarily, the applicability of the implemented suite of protocols was assessed in relation to its usage in the implementation of the test case scenario. Erlang as a language of implementation was evaluated as well, and its promised strengths, were assessed. The established algorithms were evaluated in terms of their performance, and finally, insight was given regarding the failure detector used throughout this project.

Chapter 9

Conclusion and Future Work

No matter the amount of work that is done, there is almost always room for improvement, In this section various possible improvements for this project are outlined. Following this, the relevance of the work done is highlighted, and then finally, this work is concluded.

9.1 Alternative Failure Detector

In this project, a Perfect Failure Detector was assumed throughout. Most algorithms only work if such a failure detector is available. Moreover, a very simple approach to attempt to achieve a Failure Detector, was taken.

In reality, failure detectors are an area of research on their own. Failure detector algorithm are proposed in various literature, and these should be taken into consideration for future work.

Moreover, weaker failure detectors, such as the Eventually Perfect failure detector, should be studied. These might be more practical to build in the real world, however, the algorithms for the Agreement problems presented here, are much more complicated to build with such a failure detector.[6]

9.2 Approaching the indefinite growing of the state

Some of the algorithms outlined in this project, were more focused on achieving correctness, rather than seeking a practical implementation. In fact, some of the algorithms store all messages received - resulting in internal structures which grow indefinitely and require infinite memory. In this work, different kinds of optimisations were attempted, such as keeping hashes of messages instead of the messages themselves. However, this does not really solve the problem and further investigation of this aspect needs to be studied.

As an example, consider the algorithm for *regular reliable broadcast.* This algorithm required a *Delivered* sets which holds all messages which were delivered, and prevents messages from being incorrectly delivered multiple times. For future work, a simple scheme is proposed in which all processes acknowledge that they have received the message. When all processes have received the message, then the local copy of the message would not be needed any more and could be deleted. However, there could be situations where this message which was deleted from the *Delivered* set, is received again, and would be erroneously delivered again. Such a situation is shown for the *regular reliable broadcast* algorithm in figure 9.1.

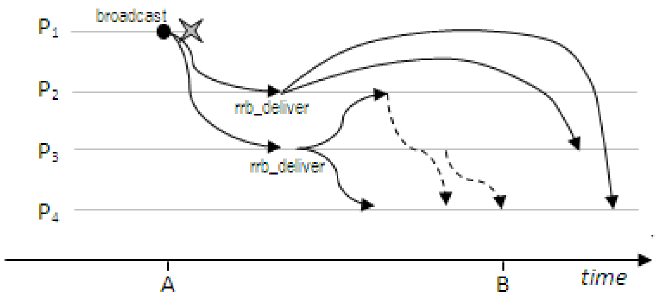


Figure 9.1: **Need for the Deliver set**

Here, the dashed lines indicate acknowledges for the message which was initially broadcast by P_1. For the sake of clarity, only the acknowledges directed towards P_3 are shown. In this run, process P_3 receives acknowledges from all other processes at point A. Hence it may remove this message from its internal *Delivered set.* However, this same message is received again, shortly afterwards, and hence would result in incorrectly redelivering this message.

For future work, we propose to assess to use of timestamps (such as Lamport timestamps) to be able to determine that this message has already been delivered. The idea is that every process keeps a timestamp for every node, and the message received is only delivered if it has a later timestamp. This method, however, would require that message sent by the same sender, do not get reordered in the network - however such a property can be easily satisfied by a transmission control protocol.

Hence here, we suggest the study of this idea even further, so as to possibly come up with a solution for these ever growing sets.

9.3 Different Failure Model

In this project, only *crash failures* were taken into consideration. A process was not expected, or even allowed, to recover once it had crashed. Doing so, it might break the algorithm being executed.

However, there is a lot of study on agreement algorithms with *transient* or even *byzantine* failures. Once again, the algorithms for these failure models, are much more complicated than those for the *crash failure* model. However, as part of the future work, it would be interesting to investigate these failure models as well.

9.4 Analyse further specifications and agreement problems

The work done in this project, by no means covers all specifications or studies all of the agreement algorithms. Various variations or even completely different problems exist. For example, in this project, no probabilistic algorithm was studied. However, in very large network, one usually has to resort to such algorithms because of the massive amount of messages, and computation rounds which would be otherwise required.

9.5 Relevance of work done

The work done in this project, studies the way a system of processes can cooperate together, to solve different classes of agreement problems, in the presence of failures and unreliability. In a time, where most applications are moving away from a centralized approach to a more distributed environment, guaranteeing correctness in a distributed system is always becoming more of a priority.

The work presented here, provides a suite of algorithms through which correctness can be achieved without the need for central coordinators. This entails a degree of fault tolerant operation, because nodes can fail, and the system could still remain in operation.

Nowadays, various technologies work on this principle, such as cloud computing, decentralized distributed databases and peer to peer technologies. Hence such a suite immediately has strong applicability for the development of such systems. Moreover, it could be used to explore the possibility of decentralising existing system, so as to achieve fault tolerant characteristics as well being able to guarantee a reliable distributed service.

9.6 Conclusion

In this project, the problems associated with distributed programming were analysed. In particular, the distributed agreement problems were studied in detail. Erlang was chosen as the implementation language, and a common framework for these algorithms, was developed using this language.

A number of distributed agreement specifications were then studied in details. Following this, algorithms for each of these specifications were also investigated and actually implemented in the framework developed in Erlang. This provided a suite of reusable protocols which tackle these agreement problems.

The relevance of this suite was then assessed with a real world scenario. Using this suite, a decentralized file system was implemented. The benefits achieved from such a suite, were then evaluated.

Appendix A

Using the protocol suite

Here the details on how to use the protocol suite developed in this project, are outlined.

A.1 Prerequisites:

- Erlang/OTP framework
- Open SSL: required by Erlang's Crypto Module, which is used in this project.
- Protocol Suite (found in the code directory)

A.2 Setting Up

Make sure that the following files are set up:

1. Erlang hosts file: This file is located in the home directory and is name .hosts.erlang. The domain name of all machines with which the system needs to connect should be put here
2. Erlang cookie file: This file is located in the home directory under the name .erlang.cookie. This should contain a random string. The content of this file should be the same on all nodes

A.3 Starting a shell

In the command prompt write the following.

```
erl -sname NODENAME -peers PEER1 PEER2 PEER3
```

A.3.1 Explanation

This starts the Erlang shell and assigns it the name NODENAME. Here the name of the shell of all peers should also be specified in the form of arguments to the -peers switch. Note that if the shells are not on the same domain, then =name should be used instead of -sname, and all names should be given as fully qualified domain names.

A.4 Initializing the Suite of protocols

In each Erlang Shell, write

```
node_utils:start_node().
```

Alternatively, if a shell is to be used as the monitor node (used for debugging, all output is directed to that shell), the following should be entered to initialize all Erlang shells:

```
node_utils:start_all_nodes().
```

Bibliography

[1] Gregory R. Andrews. Distributed programming languages. In *ACM 82: Proceedings of the ACM '82 conference*, pages 113–117, New York, NY, USA, 1982. ACM.

[2] Joe Armstrong. *Programming Erlang - Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[3] Joe Armstrong, Bjarne Dcker, Thomas Lindgren, and Hkan Millroth. Erlang whitepaper. `http://ftp.sunet.se/pub/lang/erlang/white_paper.html`.

[4] Joe Armstrong and To Helen. Making reliable distributed systems in the presence of software errors, 2003.

[5] Joe Armstrong and Steve Vinoski. The road that we didn't go down. `http://armstrongonsoftware.blogspot.com/2008/05/road-we-didnt-go-down.html`.

[6] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *PODC '92: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 147–158, New York, NY, USA, 1992. ACM.

[7] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[8] Bernadette Charron-Bost. Agreement problems in fault-tolerant distributed systems. In *SOFSEM*, pages 10–32, 2001.

[9] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[10] D. Farina. A case for erlang. `http://metalinguist.wordpress.com/2007/07/04/case-for-er`

[11] Michael J. Fischer, Nancy A. Lynch, and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186, 1981.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[13] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[14] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] Maurice Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[16] S. Krishnaprasad. Concurrent/distributed programming illustrated using the dining philosophers problem. *J. Comput. Small Coll.*, 18(4):104–110, 2003.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[19] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[20] Nilsson Niclas. The multicore crisis: Scala vs erlang. `http://www.infoq.com/news/2008/06/scala-vs-erlang`.

[21] Klaus Renzel. Error handling for business information systems - a pattern language. 1996.

[22] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.