Supporting Fault-Tolerance in a Distributed Multi-Process Environment

Aman Sagar Kohli, BSc.

School of Computer Applications

Dublin City University

Supervisor: Dr. David Sinclair

September 1999

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of MSc. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

[illegible] 24/9/99.

Aman S Kohli. ID Number: **95970843**. June 1999.

Contents

1 Introduction **1**

1.1 Roadmap 7

1.2 Presentation Notes 8

2 Literature Review **9**

2.1 Defining Failure 9

2.2 Approaches To Handling Faults 12

2.3 Atomic Transactions 14

2.4 FT-Linda 15

2.5 The SPROC Approach 19

2.6 Configuration Management 23

2.7 Reconfiguration 24

2.8 Identifying Failed Processes Using Time 25

2.9 Achieving Consensus 29

2.10 Conclusion 30

3 Supporting Fault Tolerance **33**

3.1 Fault Tolerance Implementation Strategies 33

3.2 Process Model 35

3.2.1 Distributed Shared Memory Approaches 35

3.2.2 Communicating Processes 39

3.3 Fault Tolerance – Detection and Recovery 40

3.4 Selecting a Process Model 42

3.5 Proposed Process Model 45

3.6 Fault Tolerance Through Language Extension 46
3.6.1 Avalon/C++ 47
3.6.2 FT-Linda 52
3.6.3 Possible Enhancements to FT-Linda 54
3.7 Architecture of Selected System 54
3.7.1 Fortifying the Network 57
3.7.2 Detecting Failure 58
3.8 The Implementation Model 58

4 **A Language and Environment for Providing Fault-Tolerant Distributed Systems** **61**
4.1 Language Overview 62
4.1.1 Language Description 63
4.1.2 Control Statements 65
4.1.3 Distributed Operations 66
4.1.4 Compilation 67
4.2 The Simulation Environment 67
4.3 Simulation Machine Instruction Set 68
4.3.1 Standard Instructions 69
4.3.2 Executive Interface Instructions 69
4.4 The Runtime System 72
4.4.1 Major Components 73
4.4.2 Transmission and Reception Context Creation 78
4.4.3 Identifying Faulty Processes 81
4.4.4 Achieving Consensus 84
4.5 Dealing With Faults 85
4.5.1 Summary of Supported Features 87
4.6 The Simulation Machine 88
4.6.1 User Interface 89
4.6.2 Measuring Effectiveness 93

5 Results **94**

5.1 Introduction 94

5.2 The Test Case 95

5.3 The Results 96

5.3.1 Dealing With Errors 96

5.3.2 The Effect of Errors and Replicas – Performance Degradation of the System 100

5.3.3 Replica Overhead 102

5.4 Conclusions 103

6 Conclusions **104**

6.1 Future Extensions 106

6.2 Conclusion 109

A ftdl Language Description and Compilation Environment **111**

A.1 The ftdl Language 112

A.1.1 Lexical Analysis 112

A.1.2 ftdl Grammar 113

A.2 Compilation 118

A.3 End Remarks 118

List of Figures

1.1 Distributed Application Example 4

2.1 Linda example 17
2.2 FT-Linda example (user error) 18
2.3 FT-Linda example – rewritten 19
2.4 The Combining Condition 24
2.5 Distributed Executive main loop 28

3.1 Linda Example 41
3.2 Identifying deterministic/non-deterministic data modifications . 43
3.3 recoverable class declaration 50
3.4 Avalon/C++ CustomerTable Example 51
3.5 `atomic` class 52
3.6 `subatomic` class 52
3.7 Traditional distributed application architecture 56

4.1 A Protocol Declaration in ftdl 62
4.2 Mapping of an operation which does not have a return a value . 67
4.3 The Runtime context component 73
4.4 The Context Manager 75
4.5 Request identifier composition 75
4.6 Runtime Context Local Variable Interaction 80
4.7 State transitions of the runtime 82
4.8 Consenus Alogrithm 85
4.9 ftdl Distributed Architecture 87
4.10 Environment Initialisation 89

4.11 Replica Cell 90
4.12 Main Simulator Screen 92

5.1 Eleven Test 97
5.2 Completion vs. Probability of Failure 98
5.3 Average Invocation Time vs. Probability of Failure 101
5.4 Replica Cost 102

A.1 EBNF Description For Protocol Declaration 114
A.2 EBNF Description For Protocol Implementation 115
A.3 Statements in ftdl 115
A.4 Expressions in ftdl 116
A.5 Control Statements in ftdl 117
A.6 Operation Invocation and Reception Statements in ftdl 117
A.7 Copy Approach To Sharing Protocol Declarations 119

List of Tables

4.1 Primitive types supported by ftdl 63
4.2 Supported operations 64
4.3 Simulation Machine Standard Instruction Set 70

5.1 Replica Count, Failure Probability and Average Invocation Time 99
5.2 Replica Cost at 0% Failure Probability 103

A.1 Reserved Words in ftdl 112
A.2 Lexical Definitions 113

Abstract

This thesis is about designing a distributed system that *transparently* [6] supports fault tolerance. In order to provide this, a lightweight language, the *Fault Tolerant Distributed Language*(ftdl), has been developed which supports the essential features required to enable fault tolerance. This language is the user interface into the runtime fault tolerant distributed architecture. By adopting a hybrid approach based on existing work in distributed systems, a model for distributed fault tolerant computation has been constructed based on distributed shared memory and communicating processes. The effectiveness of the model in the face of failure is measured. The manner in which the model deals with failures, the degradation of the system in the face of failures and the overhead associated with the fault tolerant components is explored. Overall, the model has successfully shown the requirements to build a transparent fault tolerant system.

Acknowledgements

I would like to thank David Sinclair for his helpful guidance and support over the last 3 years. Without it, it is very unlikely that I would have reached this stage.
My colleagues in IONA have also helped in a variety of ways; from giving me the time to work on the project to help in proof-reading the thesis.
Ciaran McHale and James Power have provided guidance and help in the production of this thesis through their knowledge of LaTeX.
Particularly I would like to thank Adrian Trenaman for his patient and constructive comments while writing the thesis.

Chapter 1

Introduction

A distributed system consists of multiple autonomous computers interacting with each other. This collection of computers can be referred to as a *network* [23]. Tanenbaum [23] states that two computers are interconnected if they are able to share information. The connection medium can be anything; for example a cable, a system bus or microwaves. Autonomous means that one computer cannot directly control another, for example it cannot make the other computer start or stop. The computers interact with each other by transmitting messages over their connection medium. Every computer represents a node on the network, and the links correspond to the connection medium joining all the computers together. This definition excludes the classification of multi-processor computers as representing a distributed system.

The work presented in this thesis attempts to provide an environment where distributed systems can be developed that can tolerate arbitrary failures up to the number of processes replicated. To this end, a distributed language and runtime environment has been developed, known as the *Fault Tolerant Distributed Language* (ftdl). The ftdl system provides a simple language, runtime support system and simulator. The language is lightweight, but serves to provide the minimal support necessary for providing fault tolerance at the language level. The runtime system manages the inter-node communication as well as incorporating the fault detection mechanism. A simulator has been developed on which the ftdl environment can exist.

A simple way to describe a failure is to describe it as "something has gone wrong", that is the failed component did not produce the result it was supposed to. Note, a failure is different to a *"bug"*. A bug is a defect in a component, whereas a failure is something that occurs unexpectedly which causes the component not to behave as was expected. For example, consider a simple computer program that is supposed to add two numbers and output the result. In a "buggy" implementation, say, it multiplies the results instead of adding them. So an input of 2+3 results in 6 and not 5. An example of a failure in this scenario would be the termination of the program without printing a result during a particular execution of the program. A "bug" is thus a programatic error, whereas a fault is an error which occurs during execution.

The ftdl system attempts to provide solutions for fault tolerance against *fail stop* and *Byzantine* failures. A fail stop is a failure that is marked by a change of state – usually from a *running* state to a *stopped* state. This failure can be detected by other processors. Byzantine failures are more awkward to deal with. Failures of this type are difficult, and in some cases, impossible to detect. When Byzantine failure occurs it appears as if something else has failed, it is as if the failing entity has "covered" its tracks.

Distributed applications are built on distributed systems. A distributed application runs on two or more nodes of a distributed system; a non-distributed application, on the other hand, only runs on a single node. A distributed application can be separated into the following layers:

1. User code
2. Distributed environment executive
3. Network layer interface

The user code layer is the code written by the programmer to deal with the application being implemented. The distributed executive is the component that controls the communication between the remote computers. It provides the interface between the application code on a particular node on the network and the code on another node, via the network. Some environments provide language libraries which the application programmer uses to provide the distributed capability to the program. BSD `sockets` [20] are an example of this.

The distributed executive may need to need to translate the data received from the remote computers into a format understood by the user code, similarly the executive may need to translate data to be passed to the remote computer from the user code into a form that is understood by the network layer. This is done by an interface to the distributed executive. In a system like CORBA [25], this interface is the generated stubs from the specification, expressed in IDL. The ORB runtime converts these structures, which the user code uses in a language dependent way, into a format understood by other CORBA applications, that is, the data is converted into an implementation language independent form.

Some environments, however, implement their distributed environment through a language, as SR [4] shows. Here, the interface to the executive is built into the language itself in terms of constructs and expressions. An example of this is the **in** statement, described by Andrews et al. in *"An Overview of the SR Language and Implementation"* [4].

One form of the **in** statement takes the form:

```
in communication_endpoint
and synchronisation_expression by scheduling_expression →
body of in
ni
```

This blocks the currently executing process until it receives data on the communication endpoint specified and the synchronisation expression is satisfied. The data is delivered in the order specified by the scheduling expression. This allows the programmer to explicitly control how to deliver the data. For example, Andrews et al. describe a bounded buffer with an insert operation specified as follows:

in *insert(item)* **and** *count* < *size*

This is telling the distributed executive that the current process is waiting for data which is meant to be delivered to the *insert* communication endpoint. This process will execute the body of the **in** statement only if the value of the local variable *count* is less than the value of the local *size*. This expression is evaluated *every* time the executive considers allowing execution to continue for the blocked process.

When interfacing to the network layer, the executive implements the communication subsystem. This defines the format of messages that are transferred between nodes of the system.

As well as this task, other communication derived tasks must be performed: such as routing messages, managing message timeouts, retransmissions, - dealing with new connections and removing old ones. These may be built on top of the underlying network protocols.

Figure 1.1 Distributed Application Example

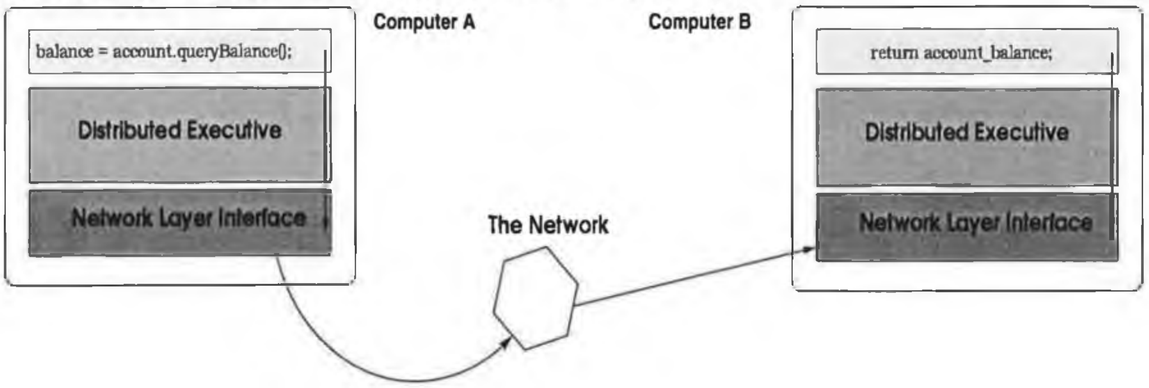


Figure 1.1 presents an example of querying a remote bank for the balance of a bank account. At *Computer A*, the user code is calling the `queryBalance`

operation on an `account` object, the result of which will be stored in the variable `balance`. The `account` object, however, is actually a remote object, i.e. it lives on another machine. For the purposes of this example, let the `queryBalance()` request initiate at *Computer A* and it will be routed through the network to the where the account object exists – *Computer B*.

The `queryBalance` operation, is thus a remote invocation, which can be decomposed into the following steps:

1. **Distributed Executive.**

   (a) The translation of the statement `account.queryBalance()` is made by the interface to the distributed executive. This takes all the information known about the current call – the operation name, target object and location of the object – and puts the information into a format to be used by the distributed executive. This translation can be expressed as:

   > **send** *queryBalance* **operation** to *Jack's Account* **at** *"First Savings Corp., Main St."*.

   The destination for the request will be *Computer B*.

   (b) The distributed executive translates this data into something that the network layer interface can use.

2. **Network Interface Layer.** Takes the data from distributed executive, attaches network addressing information (which allows the network layer to determine to which machine the message is going) and then converts this data into a form which can be written onto the network.

In practice, a distributed system will be concerned with routing messages from computers to computers. Typically applications will be written on a multi-tasking operating system so the applications are more concerned with messages going from a process running on one computer to another process on another computer. The term process refers to what the host operating system regards as a process. The application code executes in the context of an operating system process.

Due to the complexities of a distributed system, there are many type of failure which can occur: hardware, software and network. An example of a hardware would be the power supply of a computer (i.e. a node) short circuiting, resulting in the computer to stop functioning. An example of software failure would be an incorrect result being returned from a function. A network failure occurs when the medium connecting the computers in a distributed system fails. Each of these component failures occur with varying probabilities and the ability to detect failures also varies. Software failures can be impossible to detect, where is it can be quite easy to detect hardware or network failure. Bearing the above restrictions in mind, the work presented here is concerned with *general* failures and the ability to design and implement a system which can handle failure.

One of the advantages of a distributed system is that it can facilitate cooperation without centralised control, but also can provide higher reliability and availability. It can be designed to allow the masking of failures, thus increasing its availability. Many applications are naturally distributed, for example a network of automated teller machines. Distributed environments can be used to aid the efficient use of resources through load balancing.

These advantages come with some drawbacks. As there are many more components – processors, network connections, inter-processor connections, etc. – a higher probability of failure exists over single processor based solutions. If a single component in the distributed system fails, this should not mean that the entire distributed application must stop functioning. In order to ensure this does not happen, a distributed application must be able to continue function in presence of failures, that is it must be *fault-tolerant*. Many distributed languages have failed to fully integrate fault tolerance into their environments, leaving fault tolerance to be implemented in an *ad-hoc* and inconsistent manner on various systems by the application developer. By making fault-tolerance an application level concern, it requires that the application developer must focus on fault tolerance as well as the application she is attempting to develop. It also requires that the application developer be an expert in fault-tolerance techniques. These goals cannot always be met so to build truly robust fault-tolerant systems this critical feature must not lay in the application layer, but in distributed executive layer.

In order to ensure an effective, transparent fault tolerant distributed system, all the layers of the distributed environment must be modified to provide fault tolerance. If this does not occur, the amount of fault tolerance that can be provided to the application decreases. For example, one of the guarantees required for the network layer is the ability to bound message transmission times between two processes on separate computers. Clearly this can only be done with certain types of networks [23], such as fixed time-slice networks but with other types, like wide-area networks or Ethernet, this is not possible. The consequence on fault tolerance in this case would be an algorithm that cannot accurately use time to determine if a process has failed, so the overall ability to provide full fault tolerance has been reduced considerably.

1.1 Roadmap

This thesis presents the work in three major parts. The first reviews the current literature on providing fault tolerance to distributed systems. This is done by presenting a literature review, starting in Chapter 2. The next chapter describes the various ways in which synchronisation can be achieved in a fault tolerant manner. Chapter 3 also examines providing language extensions to provide existing systems with fault tolerant capabilities. The chapter concludes by describing the process model and approach adopted by the ftdl system. Chapter 4 describes the ftdl system components and its simulation environment. The effectiveness of the model is presented in the results chapter, Chapter 5. Finally, Chapter 6 presents the concluding remarks.

1.2 Presentation Notes

Many of the structure diagrams presented in this thesis use the *Unified Modelling Language* [10] notation. These are primarily found in Chapters 3 and 4.

When presenting examples from the ftdl language, a courier small caps font is used: an `FTDL LANGUAGE EXAMPLE`. Reserved words in ftdl are presented in a sans serifed bold font: **the reserved typeface**. This same font is used when presenting the machine op-codes of the simulation machine. The op-codes of the machine are presented as three character instructions. Examples from other programming languages are presented in a simple courier font: `some other language font`.

Chapter 2

Literature Review

There are a number of approaches to providing fault-tolerance to distributed systems. This chapter identifies and examines some of those currently in existence. Some of the algorithms only apply to distributed shared memory systems, others to database systems and some to distributed, inter-acting processes. Before selecting an approach to take, the type of distributed system needs to be determined. By examining the different systems, a set of requirements can be derived for constructing a system that tolerates failure.

2.1 Defining Failure

One of the methods for implementing fault tolerance is the use of replicas. The processes that make up the distributed system are replicated on other processing nodes. A replica of a process is copy of that process that does not exist on the same node as the process being replicated. Typically a replica is another process running on another machine, executing the same program that is being replicated. Once a fault is detected, the faulty process is stopped and one of the replicas becomes the 'main' process. All commands issued to the main process are also propagated to the other replicas, thus ensuring at any time that all the replicas have the same *state* – that is all the process local variables of the replicas have the same value. This is the method explained by Schneider [18].

This method can be used by a message passing architecture as well as distributed shared memory (DSM) systems [5]. For DSM systems, it is the memory that is replicated, not the process. All changes made to shared memory are propagated to its replicas.

The ideas central to the issue of fault tolerance can be considered as: replica management, fault detection, recovery and reconfiguration. Replica management can be thought of as deciding which processes to replicate, how to replicate them, how to propagate changes to these copies and where to keep the replicas. Replication has a time and space overhead associated with it, so one of the issues when designing a fault tolerant system is: are all processes that make up the distributed application replicated or only a select few? Another issue that must be addressed is the form taken by replicas, some systems store replicas as duplicate processes, as indicated by Bakken and Schlichting [5], others store to reliable secondary storage. A design decision needs to be made for which alternative is chosen.

A fairly loose definition, given by Bal *et al.*, of a fault tolerant distributed system is:

> *"A system is fault tolerant if it still continues to function properly in the face of processor crashes, allowing distributed programs to continue their execution and allowing users to keep on using the system." [6]*

This definition is lacking as it does not fully define failure, they are only concerned with *total failure (fail stop)* of computers (nodes) on network. It does not deal with partial or Byzantine failures. These are much harder to deal with but are an important issue in distributed system implementation.

The above definition does not take into account the issue of network or other failure categories. As distributed systems are comprised of many components, it is better to deal with the term *failing system components*. Let us take the above term to mean a component or process in the system, i.e. a processor is a node on the network (a processing unit), not the physical processor itself. In this manner a network failure becomes another failure category for a distributed system.

How can process (i.e. a system component) failure be accurately detected? A process that is detected as failed, due to some time constraint, may be executing on a machine that may be busy, or the communications channel may have slowed down due to extra network traffic. Has the process failed or is it slow to respond [14]? This leads to a faulty process being defined [1] as:

> *"a faulty process is one who's behaviour is no longer consistent with its specification." [18]*

The failure types can be categorised as *fail stop* or *Byzantine* [18]. Fail stop is defined as a state change. This state allows other processes to detect it has failed. A Byzantine failure, on the other hand, can cause malicious, undectable behaviour [15]. A component which has failed through a Byzantine failure can lead the failure detection mechanisms believe another component has failed. These class of failures are very good at masking the actual component which is faulty.

A major issue raised with faulty processes is that of the recovery of the failed entity. The system not only needs to take into account failed processes that restart but also deal with processes that have restarted during or after recovery mechanisms that have taken place.

Though this definition is concerned primarily with faulty processes, it can be applied to the other components in a distributed system. It is vital to take into account the different components which may fail. These may be the process itself (software failure), the computer on which it is running (hardware failure) or the network (network failure). A faulty component is one that has

[1]This definition, though not rigorous, can be broadly interpreted as meaning that a faulty process is one that has stopped working.

stopped behaving the way it is expected to. Applying this definition to the network component, the network has failed when it cannot route a message from one node to another.

2.2 Approaches To Handling Faults

Currently, the options language designers have taken for handling fault tolerance are identified in *Programming Languages for Distributed Systems* [6]:

Ignore. Ignore the fact that faults exist, i.e. nothing will inhibit or stop the processes. However, a program executing on several computers has a higher chance of failing than when running on one.

When a process tries to communicate with one that has failed, it may be blocked forever or get a communication error and terminate. Bal *et al.* [6] note that since processor crashes are rare, a language that ignores faults is not a major problem.

This may be true when the only failing component is a CPU, but if the failed entity is another component of the distributed system, this manner of 'handling' faults is not entirely acceptable. A solution applicable to a distributed system must be found.

Programmer. Let the programmer handle faults. The language run-time system will return an error code (or raise an exception) to the concerned (distributed) processes that want to interact with the failed process. It is then up to the programmer to handle faults.

This approach is adequate if failed processes have no side-effects. For example, if the process that fails just returns the balance in a bank account, another process can be invoked by the language run-time system. However, if the process must, say, obtain a database record lock and fails before releasing it, starting a new process will not solve the problem. This later process has a side-effect of obtaining a resource. A mechanism is required to ensure an action runs to completion or not at all.

Language Support. One construct that aids a programmer in defining system components as fault tolerant is the use of atomic instructions. This support is supplied through the run-time system and the language grammar itself.

FT-Linda [5] is an enhancement to the `Linda` [2] programming system that provides fault-tolerance. Here, the concept of an atomic guarded statement is added to the language. The purpose of this is to indicate to the runtime environment which tuple spaces are required for the transaction; once these are secured, the body of the statement then executes as atomic transactions. This is discussed in the section *Atomic Transactions* in section 2.3.

Transparent Fault-Tolerance. e.g. fault tolerant message passing. This has been suggested by Borg *et al.* [9]. For each process an inactive backup process is created at another node. All messages to the primary process are sent to the backup. The backup process records all the messages sent by the primary. If the primary process fails, the backup becomes active and starts repeating the primary's computations. During normal computations the primary and backup processes are synchronised.

This scheme is obviously limited to a single failure system. However the principle of messages being sent to a primary process and its backup can be expanded to allow multiple replicas. This is further discussed below.

A combination of the above approaches would lead to a solution to the fault-tolerance problem. Ideally the solution would provide both language support and transparent fault tolerance. FT-Linda [5] and Avalon/C++ [11] are both examples of existing languages augmented to support fault-tolerance.

2.3 Atomic Transactions

This section examines *transactions*. Though normally associated with the database field, this area is quite important for distributed systems. From this perspective, a transaction can be regard as an operation on the shared entity. For distributed share memory systems, this would be any operation which takes place on the shared memory. For message passing systems this would be any message sent to another process.

If an operation (transaction), when viewed externally, has no intermediate states, it is said to be *indivisible*.

If all objects participating in a transaction can be restored to their initial state after a fault so that the transaction has no effect at all, it is said to have the property of *recoverability*. To achieve this property, all operations are applied to a *copy* of the object which was intended to receive the transaction. This copy is known as a *version*; if the transaction fails, the version is discarded. On success, the version is committed to stable storage. This stable storage has a high resilience to faults and is accessible by all processes. Obviously there is a overhead in time and space for copying an object, applying changes to it and if the transactions do not fail, re-applying/copying the version as the new object. The versioning and locking of objects can be dealt with by the language runtime system.

A group of operations is atomic if it has the properties of indivisibility and recoverability [6]. This is an important definition as it allows multiple operations to be grouped together as if they were one atomic transaction, provided the group has the properties of indivisibility and recoverability.

Atomic transactions are applied to all functioning replicas or to none of them *and* are executed without interleaving at each replica. This guarantees that all replicas will execute the same set of instructions and the instructions in a transactional set will be executed *in the same order.*

This guarantee allows the principle of process replication to work. Here, we use the premise that if you send the same instructions, in the same order to all the replicas of a process, running at different nodes, the state of all the replicas will be the same – i.e. consistent. If this condition did not hold and if some replicas did not get the transaction, or they were executed in different orders, the replicas would have different states. If one of these conditions did not hold then the system would be in an inconsistent faulty state as no replica could be used as a replacement and its results could not be used in further interactions. This only holds true for transactions that are deterministic.

2.4 FT-Linda

The concurrent language Linda uses shared memory in the form of a tuple space(TS) as the entity which shared between different threads of execution. This is an associative access data store, i.e. data is addressable by content and not by address. Tuples placed within a TS are immutable: once the data has been placed in it cannot be changed (unless the specific remove request is applied). TS's also have the inherit properties of temporal and spatial decoupling. Spatial decoupling means that a process does not have to know which processes it is communicating with. Temporal decoupling allows communicating processes to exist at different times - the processes do not have to be concurrently active. These abstractions help enable Linda to be extended to provide fault tolerance.

Three fundamental operations exist for tuple spaces: `out`, `in` and `rd`. `out` is an asynchronous operation that deposits tuples into the TS; `in` extracts tuples based on a template (it can also be used for synchronisation); `rd` is like `in` but does not remove the tuple from the TS. Asynchronous versions of `in` and `rd` exist; they are `inp` and `rdp` respectively.

Linda [2] is a language that lends itself quite easily to transparent fault tolerance. As all of the current state of the system is maintained in controlled access tuple spaces, the issue of what to replicate and when to log changes is simplified. FT-Linda [5] extends the basic Linda language and runtime environment to facilitate an effective fault-tolerant system using a DSM architecture.

These extensions have been designed to allow the convenient implementation of fault tolerant systems and the cost of execution kept to a minimum [5].

The values of volatile memory can written to stable memory (e.g. to disk). Alternatively, if the values are to be shared across multiple nodes, the values can be replicated onto multiple nodes.

A number of issues are involved when using a distributed shared memory (DSM) model to provide fault tolerance. The tuple space of Linda is essentially a DSM component. A number of these issues are examined in the paper *Distributed Shared Memory: A Survey of Issues and Algorithms* [17] and these should be taken into account when a DSM system is designed. The *structure* and *granularity* must be defined. The *structure* describes the semantics of the bytes in memory – does a particular memory location refer to an 'integer', a structured type or a particular programming language's 'object'?

Granularity describes the size of the unit of sharing; for example, this can be a byte, word, page or data structure. The larger the shared memory segment, the less "paging" overhead occurs; however, having large shared "pages" may cause contention between different processes.

The next issue with designing a DSM system is that of the *coherence semantics* of the memory. This addresses the issue of when local nodes update their local copies of the shared memory.

Fortunately, the Linda language itself can help in providing solutions for the implementation of the DSM. The structure and granularity issues are addressed by the tuple space concept of Linda.

By this use of stable tuple spaces and atomically executing actions, fault tolerance can be achieved in Linda. If commands are deterministic and are executed atomically with respect to concurrent access, the state variables of each replica will remain consistent [5].

Bakken and Schlichting [5] describe another way in which to ensure fault tolerance in Linda programs is the "bag of tasks" approach [5]. Here, the tuple space is filled with tasks that must be carried out; when a process wants to execute a task, it marks it as "in progress" space and also places a breakdown of what needs to be done. It will then begin to execute these sub-tasks. If at any time failure occurs, the work can easily be carried on.

There are some restrictions placed on FT-Linda. The guard of atomic guarded statements (AGS) may block, the body cannot. This is in place to prevent deadlock. All tuple space operations must be part of the AGS.

The first restriction is quite awkward from a programming perspective. In the body of the AGS, all the `read` and `in` (and their variants) tuple space operations must reference existing tuples, if not an exception is raised and the program terminated. Thus the programmer must ensure that all references to tuple spaces are placed as part of the guard in the AGS. This adds an implicit overhead onto the runtime system. For example, the programmer must insert a `rd` into the guard for a tuple that it is not concerned with at the current execution point. Then while doing computation in the body, the `rd` is reissued, the result may different from the guard's `rd` because previous operations in the body may have changed the value.This adds unnecessary network communications and programmer burden. This is one of the problems of adding a feature onto a language that does not intrinsically support it.

Figure 2.1 Linda example

$\mathbf{in}(firstVal, ?val1)$
$\mathbf{in}(nextVal, ?val2)$
$\mathbf{out}(result, \mathbf{plus}(val1, val2))$

Figure 2.1 shows an example of a simple `Linda` program fragment which reads two values from the tuple space, adds them together and deposits the result back into the tuple space. The statement: $\mathbf{in}(firstVal, ?val1)$ reads the value from the tuple with key $firstVal$ into the local variable $val1$. Similarly, $val2$ obtains its value from the tuple with the key $nextVal$. The values are added together and deposited into the tuple space into the record with key $result$. In order to implement this in FT-Linda, the statements must be rewritten in AGS form. The guard of the AGS is formed by performing the two **in** statements. The body of the AGS would be the **out** statement as in Figure 2.1. This approach requires the programmer to think about what parts of the program must be made fault tolerant. As this is done manually, some error may occur because the AGS structure is invalid.

Figure 2.2 FT-Linda example (user error)

$$
\begin{aligned}
&\langle\ \mathbf{in}(firstVal, ?val1) \Longrightarrow \\
&\qquad\qquad \mathbf{in}(nextVal, ?val2) \\
&\qquad\qquad \mathbf{out}(result, \mathbf{plus}(val1, val2))\ \rangle
\end{aligned}
$$

Figure 2.2 shows an *incorrect* approach to implementing the above Linda code as FT-Linda. An AGS is denoted between the angle brackets (between the '$\langle$' and '$\rangle$'); the *guard* occurs before the '$\Longrightarrow$'; the *body* comprises the statements from the '$\Longrightarrow$' to the '$\rangle$'. The body contains an **in** statement, which causes the Linda runtime to block the process until the requested tuple is present; however, with FT-Linda, this will cause a runtime error, as the body is not allowed to block. The approach taken by FT-Linda requires the programmer to separate parts of the application as being non-fault tolerant and those being fault tolerant. It is open to user error which can produce inconsistent and non-fault tolerant systems. The programmer is also required to adjust the normal flow of statements and "protect" against faults explicitly, by dividing the program into AGS's. A correct approach is shown in Figure 2.3. This requires issuing two AGS's, one of which contains a null body to ensure that the `in`operation is fault-tolerant. Clearly this is intrusive for the programmer.

Figure 2.3 FT-Linda example – rewritten

$\langle$ **in**$(firstVal, ?val1) \Longrightarrow$; /* *null body* */ $\rangle$
$\langle$ **in**(nextVal, ?val2) $\Longrightarrow$
                                        out$(result,$ **plus**$(val1, val2))$ $\rangle$

2.5 The SPROC Approach

Following on from the above primitives for distributing memory, a higher level construct can be used to model fault recovery. This is the approach taken by FT-Linda [5] based on the model proposed by Schneider [18].

Schneider identifies, three types of objects which exist in a distributed system – the client, server and output device. A *client* requests a service from the server. The *server* provides this service. The *output* (or result) from the server may go to the output device (which could be the client). [2]

Though Schneider only considers a client-server approach to distributed computing, his model for fault tolerance can be applied to other distributed systems as well. Consider the *client* as the *issuer* of a request to a *rproc* (a *r*equest *p*rocssor) – the *server* in Schneider's model.

Schneider [18] proposes a *state machine* approach to fault-tolerance. In this model, a state machine is a process with state, an sproc. In the model, the rprocs are replicated and co-ordination is provided between the interaction of issuers and the rproc replicas. The issuer only deals with one rproc, but the propagation of its invocations must be directed to all the replicas of that rproc.

Though Schneider refers to his model as the state machine approach to fault tolerance, do not that this not a true state machine from a Computer Science perspective. This is a process that at a particular point in time has 'state'. This state is the value of its instance (local) variables. Hence we refer to this

[2]These are relative terms, referring only to the type of transaction occurring.

model as the SPROC model.

One can think of a SPROC as a rproc, in that it has issuers (clients). The relationship between a SPROC and its issuers can be summarised as: a SPROC implements procedures and a issuer's requests are procedure calls. The issuer making a request is not blocked while that request is processed as its output (result) can be sent anywhere.

The outputs of a SPROC are completely determined by the sequence of requests it processes – they are independent of time and any other activity in the system.

A SPROC consists of state variables and commands (requests invoked by issuers). The state variables encode the SPROC 's state; the commands transform its state. A command must be *deterministic* and *atomic*. This leads to an immediate limitation to the type of objects that can be considered as valid SPROC 's. How can replication be provided for non-deterministic rprocs?

This limitation can be overcome by a number of different approaches. One such approach is to re-engineer the rproc so it only reacts to external events, thus all internal, general, events are deterministic. The portion of the rproc that causes this non-determinism, is made an issuer of the rproc. Thus it just invokes on the exposed services. These can easily be propagated to the other replicas. An exposed service is an interface element which client can invoke. An example of this is a communication channel in Ada or an operation in SR.

Another approach requires the language to recognise the non-deterministic portions. This is not always possible semantically, but perhaps a language syntax can be introduced to indicate non-deterministic portions.

The following assumptions can be made about SPROC requests: Requests are processed in a *first-in first-out* manner, i.e. the order in which they were issued. If a request r made to a SPROC s by issuer c causes r' to be made by c' to s then r is processed before r'. These two rules are quite obvious for SPROC 's that exist in a non-replicated environment, where one issuer issues requests to a SPROC it is obvious that the request r will be processed by the SPROC before *r'*. However when replicas are involved, we need to ensure that this holds as each replica may be processing requests at different rates than any of its peers. A situation could arise where a *r'* request could be received before a *r*. That is why it is important that order is preserved. The assumptions thus ensure that *r* will be processed before *r'*. In other words, these assumptions do not imply that a SPROC will process requests in the order received, but the relative ordering of the messages will be preserved.

A system can be defined as being t-fault tolerant if the following holds true:

> *"a system is t-fault tolerant if it satisfies its specification provided no more than t of the software components become faulty." [18]*

One of the ways in which to achieve this type of fault tolerance is through the use of replication. For this scheme to operate correctly, *replica co-ordination* must be used.

Replica co-ordination ensures all replicas receive and process the same sequence of requests. There are a number of ways to implement this: *agreement and order*is one method. They allow requests into an *ensemble* (a set of replicas for a particular SPROC) to be fully disseminated.

Agreement ensures that every non-faulty SPROC replica receives every request. And *order* ensures that non-faulty SPROC replicas process the requests received in the same relative order.

Agreement governs the behaviour of an issuer interacting with SPROC replicas; ordering governs the behaviour of SPROC replicas with respect to replica co-ordination [18].

In order to satisfy agreement, a transmitting process sends out a value, this value must be the same as all other non-faulty processes in the transmitter's replica set. If this transmitter is non-faulty, then the value used by the recipient processes is the one agreed on above.

In the context of a client-server invocation, a client issues a request. This request will contain parameter values. The client send the request to the server, which deals with it. Since we are operating in a replicated environment, n client replicas each issue the request to the m server replicas. At a given server replica process, m_i, it receives n request, from the client replica set. To satisfy agreement, all the messages, from *non-faulty* replicas, must contain the same values for the parameters and operation identifier.

Order is satisfied by assigning a *unique identifier (uid)* to a request and have each replica process the requests in a total ordering relation. The replica process the request with the smallest *uid*.

More details about the issues replica co-ordination, agreement and order are described in more detail in Schneider [18].

Schneider's model deals primarily with SPROC replication, however this does not insure against faulty issuers or faulty output devices. It is not always possible to replicate issuers for a variety of reasons. The application semantics may not support it or there may not be enough resources. If this is the case, the system must be designed in such a way as to minimise effects from a faulty issuer. This may involve designing the SPROC 's that interact with these issuers to only allow certain commands that can be performed on it. These may be simple commands that do not give access to major state space areas to the clients – e.g. restrict the amount of memory that an issuer can write into. Another approach is to build tests into the commands to ensure the validity of the issuer. This has obvious performance drawbacks.

In the cases where the issuer can be replicated, the SPROC can be modified to deal with all its replicas [18]. This however is quite intrusive and does not integrate with the generally clean model of Schneider's model which has the goal of *transparent* fault tolerance for all system components.

Dealing with faulty outputs requires the establishment of a *voter*. One must first make the distinction between outputs being used inside the system or outside.

If the outputs from a SPROC are going to be used outside of the system then a critical voter is used. It waits for $t + 1$ votes (fail stop) or $2t + 1$ (Byzantine

Failures). A majority is chosen if all the votes do not agree. The voter is external to the system.

When the outputs are used internally to the system, the voter is part of the output device (say the client). It is faulty when its host process is. The voter adopts a different strategy outputs used internally by the system. The voter waits for $t + 1$ identical votes from different replicas for Byzantine Failures; for fail stop, execution can proceed when any output is received.

Faults are detected by *configurators*. Each configurator is responsible for the detection of failure or the repair of the object it manages. One configurator exists for each object in the system. For fail stop failures, the configurator needs only to check the failure detection mechanism of its managed object. However, with Byzantine Failures, it is not always possible for the detection of failures. A greater degree of fault tolerance can be achieved by reconfiguration.

2.6 Configuration Management

According to Schneider [18] the state of a set of rprocs, issuers, output devices and all their replicas must be maintained. This is done typically in a system repository SPROC . Whenever an object (rproc, issuer, output device or a replica) is added or removed, that changed is mirrored in the relevant set.

These sets are usually changed when an object is replaced by the fault detection and recovery mechanism. The values of the different sets must be given to the issuers and output devices. This may be done on a demand basis, where the issuer polls the system SPROC for changes. Alternatively, on every request sent to the issuer or output device, the system attaches state-change information. This scheme requires periodic communication with the issuer/output device.

2.7 Reconfiguration

Another parameter to be taken into account for an active system is the number of failures that can be tolerated before the t fault tolerant condition is invalidated.

One way to ensure the more than t faults can be tolerated is to allow the replacement of faulty processes. The runtime system of a language must reconfigure the set of objects to accommodate failure. This may involve removing the faulty process and replacing it with a correctly functioning one. Replacing a process requires the replaced process to have the same state as the failed one before it can become an active member of the current configuration. Thus one can have a system continually running that is fault tolerant by replacing the faulty objects. This leads to the **Combining Condition** [18] shown in Figure 2.4.

Figure 2.4 The Combining Condition

$$P(\tau) - F(\tau) > enuf \quad \forall \quad \tau >= 0$$

$$enuf = \begin{cases} \frac{P(\tau)}{2} & \text{if Byzantine Failures are expected} \\ 0 & \text{if fail stop} \end{cases}$$

where:

P(τ)	represents total number of processors at time τ that are executing replicas of some SPROC of interest
F(τ)	the number of faulty replicas

As the number of failures increase, F(τ) increases, causing $P(\tau) - F(\tau)$ to decrease. Under Byzantine failure conditions, removing a faulty process from the replica set decreases $enuf$ without further decreasing $P(\tau) - F(\tau)$. However, it is not always possible to identify Byzantine failures.

Under fail stop conditions, increasing the number of non-faulty process is the only way to ensure the condition remains valid. This is done by dynamically creating new process as others fail. As this has the effect of increasing $P(\tau)$, this keeps $P(\tau) - F(\tau)$ from falling below 0.

This condition must be valid at any time during the life of the system in order for the system to remain non-faulty and produce the correct output.

To ensure this, under Byzantine conditions, removing a faulty process does not invalidate $P(\tau) - F(\tau)$. Under fail stop, just increasing the number of non-faulty processes helps to ensure the Combining Condition holds.

When replacing a faulty object, it must added to the ensemble of replicas before the condition is invalidated. Maintaining these last two constraints will ensure a greater than τ fault tolerant system. These constraints govern the rates at which failures and repairs can occur.

Performance issues may be another motivation for the removal of faulty processes. With some systems, the number of messages sent is proportional to the number of SPROC replicas that must agree on its contents. This agreement protocol may be dependent on the number of faulty processes.

The Combining Condition is one of the rules which must be kept valid during the lifetime of a fault-tolerant distributed application. It governs the number of replicas which must be active to ensure the application keeps going.

2.8 Identifying Failed Processes Using Time

One of the approaches taken when identifying a failed process is to use timeouts. If a replica does not reply to a request by the time the timeout has expired, it is marked as faulty.

The problem with timeouts is that a process can fail without doing anything. In the paper *"Using Time Instead of Timeout for Fault-Tolerant Distributed Systems"*, Lamport [14] presents an alternative approach for identifying faulty processes. The paper presents a distributed executive effecting the runtime environment of a process, as well as network constraints.

Lamport [14] argues that each component and event involved in a distributed system must have an upper bound time applied to it. Thus the time a remote invocation takes comprises of the transmission time and remote processing time. It must be possible to calculate these times, so to apply Lamport's ideas, each layer of a distributed system must be able to perform tasks in a timely, predictable manner. This allows reliable timing information to be extracted.

In his system, a process processes events. When an event occurs, it must be processed, an output and reply generated. The constant δ is defined as:

> δ is the time taken by process i to handle the event, and send the reply to process j.

Thus δ is comprised of :

1. The time needed to process an event and generate a message;
2. The time needed to transmit the message across the communication medium

One important requirement is that a process can determine the immediate source of any message it receives.

Using this approach, failure occurs when node i does not receive a response from j after 2δ seconds, this gives the system sufficient time (δ) to process the event at i and then at the receiving node j.

Since this system is dealing with absolute time, the clocks of the different nodes need to be periodically synchronised. At any time the clocks of non-faulty processes must differ by at most ϵ seconds.

Using the above assumptions, the following proposition can be defined:

> A message arrives at node j from i at time $T + \delta + \epsilon$, where T is the time when process i has received the event.

Assuming a nonfaulty communication link between i and j, and provided j is nonfaulty, then the message received by j will be the one sent by i. This message will have a timestamp of T, that is it arrived at i at time T. Node j receives the message after i has processed it and after it has been transmitted by the network, this takes a time of δ seconds.

One of the properties required for a network in a fault-tolerant environment is the ability to support broadcast. This is the mechanism used by process to communicate with their replicas. Lamport initially presents the processes in his fault-tolerant distributed environment as receiving all events [14]. When a message is sent by node i, it is broadcast to *all* the nodes in the environment. This approach allows all the nodes to respond to the event and thus keep their local state up-to-date.

The following condition is imposed on the broadcast mechanism of the underlying network: If process i broadcasts a message at time T on its clock then:

1. If i is nonfaulty then every nonfaulty process j receives the message by time $T + \Delta$ on its clock, for some constant Δ;
2. If j and j' are nonfaulty then both of them receives the same message by time $T + \Delta$ on their clock, or neither of them receives a message at time $T + \Delta$ on their clock.

A major constraint requires that for any node i wishing to communicate with node j, there exists at least one path in the network of nodes for this to be accomplished. The greater the number of paths that exist, the more tolerant the network. Under fail stop conditions, to tolerate f failures, there must exist $f + 1$ paths. For arbitrary failures $2f + 1$ paths are required.

Another important constraint that Lamport has placed on the network is that of bounded delivery times. It can be deduced from the network the time it will take to transmit a message. Obviously this is not a property held by all networks but some do support this, time-sliced [23] networks used in airplanes for example. With these networks, a node is scheduled to read from and write to the network at a fixed time. No other nodes will be reading from or writing to the network while another has access.

Each node is running the same state machine which is waiting for events to arrive. When an event arrives, it is processed. Events are processed every clock tick. For a clock with a resolution of one nanosecond, this produces a billion events every second. Lamport [14] makes the assumption that most events are NULL events; that is nothing is sent and these events require no processing. Executing an event may change the state of the process. Once all the events received at time T have been processed, then the *timeaction* command is processed. This is a special command that manages the timeout actions.

Using the above formulas for event handling, the *timeaction* can easily compare the current time to calculated values of detecting when a node has failed. So it can compare if the current time, T against $oldtime+\delta+\Delta$ to see if a failure has occurred, and then take the appropriate recovery steps.

Figure 2.5 Distributed Executive main loop

```
REPEAT →
        FOR $j := 1$ to $N$ →
            execute Event
        END FOR
        execute timeaction
        generate my own command and broadcast to all other processors
        $clock := clock + 1$
FOREVER
```

The distributed executive sits in a loop executing the code in Figure 2.5.

This idea of all the nodes executing the same code and receiving the same messages is quite similar to that of Schneider [18]. However, Lamport [14] acknowledges that the amount of fault tolerance required by a particular application should be configurable. As all these features have their cost, some can be omitted. This makes the system less tolerant to faults but it will perform better. For example, if we rule out Byzantine failures, then only $f + 1$ connection paths are required for f failures as opposed to $2f + 1$.

2.9 Achieving Consensus

Turek *et al.* present a description of the type of distributed systems that can achieve consensus and the conditions required for consensus [24]. These describe the limits of what can and cannot be done to achieve fault tolerance.

Consensus is an important topic, as it is the part of the distributed application and system that decides which value(s) to use as a result of all the input received from the various replicated processes. If the wrong value is chosen, or a value cannot be decided, the system will lose its validity and correctness. This will cause it to degenerate, at best, and produce invalid and, incorrect output at worst.

Three cases are identified as allowing consensus to be achieved:

1. Processes are synchronous and communication time is bounded.
2. Messages are ordered and the transmission mechanism is broadcast (The processes can be asynchronous or synchronous).
3. Processors are synchronous and messages are ordered.

Also presented is the result that states that consensus cannot be achieved in a *synchronous* distributed system if one-third of the processes are maliciously faulty.

Interestingly, the *Byzantine Generals Problem*[15] can be solved if the network supports broadcast (*all* nodes can 'hear' what is being said) and authentication is supported [24, 15]. Authentication is required because each replica needs to be able to verify that the replica which says it sent a message *actually* did. Under Byzantine conditions, replicas can impersonate other replicas by using another replica's messaging identification and not its own.

Authentication need not be required to solve the *Byzantine Generals*, as a solution exists if there are $3t + 1$ nodes present and each node has $2t + 1$ connections to other nodes, where t is the number of supported failures. The condition on the number of connections is to avoid network partitions.

2.10 Conclusion

With the exception of a few implementers (e.g. Schneider [18]), many of the early authors researching into the incorporation of fault tolerance into a distributed system have referred to the narrow definition of a fault as a total failure, or only dealt with one failing component, the processor (computer). However many other types of failure can occur; dealing with fail stop seems to be the simplest.

One of the problems with the SPROC approach [18] is that of overhead. Each request must be broadcast to every replica, and acknowledgements must be received before the client can resume. In a busy network environment, this is unacceptable.

The SPROC approach does not deal with non-determinism. This, too, is a serious shortcoming. Many application areas rely on non-determinism and by not addressing this a large number of applications cannot support fault tolerance.

There are two elements to building systems which provide fault tolerance. It must possible to detect faults and, after a fault has occurred, recover from faults. Fault detection requires the use of time and voting to achieve consensus to check if an entity has failed. Fault recovery requires that mechanisms exist that allow the replacement of faulty entities with ones that are not faulty. A replacement must be in the provide the same functionality the entity being replaced.

These mechanisms need to be provided for software failure, hardware failure and network failure. Replication is the mechanism to use for handling software and hardware failure. Replicas of a software process are run at different nodes (i.e. different computers) than the original process. Strictly speaking, the process replica can be running on the same node as the original process, but in the face of hardware failure, this may not be wise. In the implementation done in this thesis, there is not main process, but a process group consists of a set of active process replicas. When a replica is detected as failed, it can be replaced ('hot-swapped') with another which has the same state as the failed replica. It is up to the distibuted executive if a failed replica is replaced at the time of failure, the main concern is to ensure that the **Combing Condition** is valid.

A network may provide alternative routes to transmit a message from one replica to another. So if one route becomes faulty, there are others which may be taken. These mechanisms allow a distributed system to recover from network failure.

The network layer must provide a lot of functionality to enable fully fault tolerant systems to be built. It must be possible to obtain different routes between replicas, broadcasting must be possible and it must be possible to predict message transmission times. The reason for this requirement is to allow bounding of message transmission times so the system can detect failure of the network or replica. Obviously not all networks provide these features. To support recovery from Byzantine failures, authentication must exist in addition to the above requirements from the network.

It is possible to build fault tolerant systems that do not posses all the properties listed in the chapter. However, this comes at a cost. For example, if authentication is not present the offered solution will not provide recovery in the face of Byzantine failures. Similarly, if message transmission times cannot be predicted, then alternative mechanisms must be built into the distributed executive to allow detection of a failed node. For this type of network, the distributed executive may use a timeout value to indicate that a replica has failed if it has not sent a message before the timeout expires. As Lamport [14] states, this means a replica can be marked as failed by not doing anything. It may be the case that the replica is running on a machine that is busy and it takes longer to produce a message.

The aim of this thesis is to provide an environment which support *transparent* fault tolerance. By incorporating the mechanisms discussed and applying them to a language a system can be produced which allows developers to focus on their application and not on fault-tolerance. The ftdlsystem will provide this.

The following chapter evaluates existing process models, sychronisation and replication alternatives. By evaluating this work, a new model will be proposed to enable the development transparent fault tolerance systems. This model and its fault-tolerance approach is based on a hybrid approach of currently existing systems.

Chapter 3

Supporting Fault Tolerance

3.1 Fault Tolerance Implementation Strategies

This chapter presents the current approaches to enabling fault tolerance in distributed systems. In order to provide this facility, we need to select a process model and an implementation language as the interface between the user and the system. The process model describes which distributed entities are replicated, how they interact to detect failure and how failure recovery can occur. The implementation language and the operating system are part of the user's tools for implementing a fault tolerant system. A number of options exist for providing fault tolerance: using libraries, extending a language, devising a separate language or using a combination of these. The ideas presented in the previous chapter are expanded for implementing such systems.

Selecting a process model is important as it determines the synchronisation primitives and the semantics on what entities are to be made fault tolerant. Two different approaches are examined – *Distributed Shared Memory* (DSM) and *Message Passing* (MP)[1]. The former requires the data of the system to be replicated, the latter requires the inter-node transactions to be replicated. Synchronisation is an important aspect in distributed systems as this the manner in which the entities coordinate their activities and exchange data.

[1]MP usually occurs between *communicating processes*

Following an examination of the different process models, a host language needs to be found to support the chosen model. One approach examined extends an existing language to provide fault tolerance through a combination of libraries and language syntax extensions. The system presented is Avalon/C++ [11] which has extended a non-distributed language. An extension to Linda, FT-Linda [5], which supports fault tolerance is also presented. These two base languages differ in their support for distributed computing. C++ [21] does not support distributed concepts whereas Linda supports distributed concepts from its inception.

Any language extension must respect the philosophy of the language and any additional extensions must be added in a clean manner. The advantages of extending an existing language are many fold. The major one is the user base. Many people already know how to use the language and by adding fault tolerance, it allows the familiar language to be used in building fault-tolerant applications. It is however difficult to add features to a language that are not built into it from the beginning. Both the Avalon/C++ and FT-Linda extensions have shown this.

When designing a distributed process model, the language interface needs to be constrained and parameters set. For example, when designing a language, is it better to allow the dynamic creation of processes at runtime, or to fix them at the compile time state, as in OCCAM? The main requirement is the ability to replace faulty replicas dynamically at runtime. As this feature will be present in the language runtime, ftdl(*Fault Tolerant Distributed Language*)will expose this to the user to allow a dynamic process creation model.

The first part of this chapter attempts to select a process model for use in our implementation. This is done by examining a couple of existing models. The next part presents the alternatives for the implementation language approach. Namely, do we extend a language or create a new one? Finally, the chosen approach is described.

3.2 Process Model

There exists a number of ways to express the distribution of a system. Two such approaches are examined here – DSM and communicating processes. The key difference between the two approaches is the type of structure to be distributed. With DSM, it is memory or memory objects which are the units of distribution; processes communicate and sychronise resources through this memory. Message Passing systems, also known as "communicating processes", communicate through shared communication channels. Communication primitives exist which allow the communicating processes to interact with one another. The DSM model distributes data. The message passing model distributes messages, which can be instructions, data or requests.

3.2.1 Distributed Shared Memory Approaches

DSM can offer ease of programming over the *message passing* model; programmers are able to work with familiar constructs. This model sits at a higher level than message passing and frees the programmer from the send and receive message paradigm. The distributed entity – shared memory – can be accessed and used almost transparently in existing code. Code can be ported quite easily from a sequential model. However, this may not be a good thing, in some cases, the distinction between remote and local objects must be made – for example, for synchronisation and performance issues.

MP requires the programmer to partition data and manage communicating *values*, which may not be the actual form of the data to be transmitted [3].

Nitzberg [17] claims that the overhead associated with DSM systems is small, or 'reasonably small'. This however depends on the design and implementation strategy. For example, some DSM systems flood the network with many update messages when a write must take place.

DSM systems can be implemented using hardware, operating system, library implementations, or compiler implementations. The hardware approach extends traditional caching to that of a distributed system. Operating system and library facilities have been implemented using virtual memory management mechanisms. Compiler implementations map the shared memory accesses to the corresponding synchronisation and coherence primitives.

The design of a DSM system requires a definition of the memory layout and selecting the granularity of the shared components. The data (memory) held in the DSM can be unstructured or structured. Unstructured memory is seen as an array of bytes. Different forms of structure can be placed on the memory. The stored data can correspond to an object, some other data type in the system. This may represent a tuple (database record) as in Linda [5].

The next issue which must be addressed when designing a DSM system is the granularity, i.e. the unit of sharing. The shared memory is partitioned into pages. Each page is of fixed size and contains the data. If the granularity is high, *false aliasing* occurs. This phenomenon occurs when two unrelated portions of shared data are contained in the same page. If page size is too low then a lot of thrashing/paging overhead is introduced [17, 3].

Excessive data transfers can cause certain implementations of DSM systems to exhibit the 'Ping-Pong' effect. Say we have two processors P_1 and P_2, both accessing the same page. Further, P_1 and P_2 are part of a process iterating through the data on this page, for example updating the values of a shared array. P_1 writes to the page, and in doing so, P_2's page is now invalid. So an invalidate message is sent to P_2; P_2 then gets the updated page. Now P_2 is going to write to the page, thus making P_1's copy invalid, and an exchange occurs. This can be reduced by better use of synchronisation and picking a different memory consistency model. For example, if P_1 got a mutex on the array, iterated through it and changed the values and then released the mutex, much less data will be exchanged and performance will be increased.

Caching and Memory Consistency

Many DSM systems implement some sort of caching strategy, similar to that of virtual memory systems found in operating systems. At each node, a copy of a few pages of the global memory are kept; when a page fault occurs, the required pages are brought into this cache. This (distributed) memory, when mapped into the local address space, must be protected. The local DSM manager must be made aware of any `reads` or `writes` that take place. This can be achieved by using the host operating system's memory management API to mark certain memory regions to be protected - whenever access is made, a notice is sent to the program. Many DSM implementations *replicate* the data.

Since there may be many copies of a shared object, a *consistency model* must be found which keeps the data in a correct and consistent state – i.e. a consistency model defines the expected memory behaviour of the system [3]. This model must minimise on network communications, as this will inhibit the performance and scalability of the system. It must avoid the Ping-Pong effect and false sharing failings.

Ideally, *strict consistency* can be defined as: whenever a `read` operation takes place on the shared data, the *most recently* written value is returned [17]. 'Most recently' is a very ambiguous term, but alternatives exist: *entry consistency, lazy release consistency* and *sequential consistency.*

Sequential consistency works by sending an *invalidate* message to all replicas whenever a `write` is made on a shared object. This requires them to update their pages. With *Release consistency,* messages are sent at every *synchronisation* point, *lazy release consistency* sends the update message after a lock release. This message is sent to the next lock that is acquired on the page. However, more management overhead is needed as all other subsequent locks must be brought up to date before receiving the most current data value. Other forms of the lazy consistency model exist; the *entry consistency* is also a lazy model. Here, all shared data *objects* have a synchronisation object associated with them (this differs from the previous models, where pages are shared and not the object). Whenever this is requested, the data for the object is made consistent. Lazy models have less message exchanges than their alternatives [3].

Consistency can be used with synchronisation to ensure local caches are kept up-to-date. Synchronisation is very much implementation dependent. *Threadmarks*[3] provides both a means of locking portions of shared data and of rendezvous. Its rendezvous mechanism is known as a *barrier* and is used to synchronise all existing processes at the barrier. Rendezvous is the term used to describe multiple process which must meet (synchronise) at a specific point, a barrier is a similar notion. Each barrier has a unique ID; the purpose is to allow all processes executing up to the barrier to synchronise their shared pages. This is a slightly different reason for allowing such a feature, but it help Threadmarks system to keep the cache of pages up to date. A more typical use to allow communicating processes to sychronise their execution points. Applying this to process replicas is a way to allow all the replicas of a process to synchronise at a specific point. When the barrier is reached by all the replicas they can exchange data, perhaps to synchronise the values of their local variables.

Portability of DSM Systems

An advantage of DSM, as championed by their implementers, is the ease of portability. Programs written on single machine/non-shared memory architectures can be easily ported to take advantage of distributed systems [3, 17]. However, there are a lot of differences between single-process programs and distributed ones. One must be able to control what each process does, where it is located, how it communicates with the outside world, how it synchronises data and state. Synchronisation is not transparent in DSM systems, it must be built on top of them. Programmers may be able to share data, but the manner in which it is controlled – e.g. locks – is implementation dependent. Programs must acquire the shared data, and in another step, acquire the locking or synchronisation mechanism and somehow tie them together. An exception to this is, of course, Linda [5].

Linda has the advantage that it is temporal and location independent. Its processes do not care where the processes they are communicating with are located, or if they are still in existence. All it is doing is acting on data, and may place the output back into its global shared memory for other processes to work on. When data is to be changed, it must be removed from the tuple space and then replaced. If another processes requires it, it is blocked until the first puts it back. It is a very elegant and powerful DSM system. However, like with other distributed shared memory based systems, we are dealing with raw data. Applications must be suited to work within the tuple based paradigm.

3.2.2 Communicating Processes

An alternative to DSM is using communicating remote processes. Here, we have many independent processes – each running in parallel with the others. Communication between them is achieved through message passing [8]. Some languages, like Ada [7] and SR [4], abstract this explicit message passing into the language. Rendezvous is established by using synchronous *`receive`* and *`send`* primitives [4]. This is based loosely on Hoare's CSP model where individual sequential processes would synchronise through the `send` and `receive` primitives. In CSP both the sender and receiver have to be named. This is not as flexible as the scheme offered by Ada [8] and SR [4] where only the sender specifies the receiver's name. Linda, because its a shared memory system, goes a step further: the user does not have to specify either. On the receiver side, an `accept` statement is executed to wait until data arrives from a given channel (in Linda this is the `in` operation). This provides an decoupling between clients and servers in Linda.

In SR, rendezvous is done by issuing a `call` from the sender; the receiver uses the `receive` statement (alternatively, the `in` statement can be used). This is analogous to the message passing structure of CSP, but note that semi-synchronous Interprocess communication (IPC) can be used. By issuing a `send` call, a remote process is started, while allowing the calling process to continue execution. This is like *forking* another processes [4] in UNIX.

SR is very expressive in terms of the various synchronisation and communication models that are accommodated: Remote Procedure Call, Message Passing Rendezvous and Dynamic Process Creation. Coupled with these various models, each have many options; the `in` command can prioritise the manner in which it deals with incoming requests. Also, these parameters can utilise the arguments of the request.

3.3 Fault Tolerance – Detection and Recovery

In order to enable fault tolerance, the distributed system must be able to detect failures and subsequently recover from them. Replication of data, or processes, enables a system to recover from failures. DSM systems and message passing systems are architecturally different, but combinations of their features can help produce a process model that provides efficient fault detection and recovery.

Like fault tolerance, DSM requires replication. As the DSM system is already tracking data changes, getting it to propagate the data changes to yet another copy (i.e. the replica) of the shared memory is not a major issue. DSM systems replicate the shared memory at different nodes. The problem arises in the detection of faults. This can be done by the co-operation of the different DSM managers at each node; a lot of the data can be stored in the global shared memory itself. Recovery, however, is more of a problem. What is to be done if not all the expected processes deposit their intended values in the shared tuple space due to failure? Deadlock can occur. Consider the Linda example shown in Figure 3.1. There are three processes executing concurrently. Process A writes into the tuple space the tuple ⟨*"Checkpoint"*, TRUE⟩. The processes B and C are blocked waiting for that tuple to be deposited into the tuple space. [2] Once that tuple is deposited, processes B and C can continue executing. If, however, Process A becomes faulty before it deposits the tuple, then B and C will be blocked forever.

Figure 3.1 Linda Example

Process A: **out**(*"Checkpoint"*, TRUE)
Process B: **rd**(*"Checkpoint"*, ?boolean_value)
Process C: **rd**(*"Checkpoint"*, ?boolean_value)

The synchronisation model is built onto the DSM system through the shared memory manipulation primitives. This does not lead to a very natural way of working. These systems, by definition, rely on raw data, not higher level entities of objects or processes.

Replication for communicating processes is slightly more complex than for DSM. DSM has replication built-in. For a system consisting of communicating processes, we need to propagate state change messages to the replicas. The best manner to determine state changes is at compile time from the source code itself. Consider the following fragment of code:

```
x := y + 3;
```

This sets the value of the local variable `x` to the sum of the value of the local variable `y` and the constant `3`. The result of this expression, when executed, is a new value for `x`. This is an example of a state change of the process in which `x` is the local variable. Clearly when compiling this fragment, the compiler sees that this expression causes a state change. This is one advantage of having access to the compilation phase of a program – most state changes can be noted.

[2]The Linda command **rd** is a blocking command which waits for the expected tuple to be deposited. Unlike the **out** command, it does not remove the tuple from the tuple space.

When a fault occurs, the runtime system of the language needs to redirect all communications to the faulty process to one of the replicas. This is the act of replacing a faulty replica with a non-faulty one. The problem is like that of replacing a faulty DSM node with a new clean copy.

Checking for faulty memory in a DSM system is a challenge. This may require application based code which integrates a consistency checker (for all network copies) and the user code that checks if the values are meaningful.

Another issue mentioned in the DSM research is that of replica updates. Many of the earlier systems used broadcast to update them, but it does not scale well [17]. As there may be many hundreds or possibly thousands of nodes, broadcasting to all and awaiting acknowledgement is not feasible. A multicast protocol or something less traffic heavy should be employed as an alternative.

3.4 Selecting a Process Model

For fault-tolerance in a communicating processes system, outgoing messages between processes should be propagated to all replicas. Determining if a message will change the state of a process can be best noted at compile time; however, the type of messages propagated may not be deterministic. Here, the data of the message can vary from each replica. This variance can result from valid sensor readings, or different random numbers being generated. Consider sensor readings, the replicated processes may be attached to different sensors, monitoring the same entity. Due to differences in calibration, the time when they are read or other external factors, the values from any two sensors may differ. The processes attached to the sensors may read from them at slightly different times, within nanoseconds of one another, and there may be a small difference in result. For example, if the temperature of a room is being measured, one sensor may read 20.1 while the other reads 20.09. This data does not differ by much and may be in an acceptable range for the application. Thus these values need to be encoded in any transfers as being a non-deterministic, it can be considered as *fragile* – its value is not permanent and may differ between readings from sensor to sensor. When the fault checker is invoked, these absolute values are not compared using their absolute values, but they are checked against the other values to ensure there is not great difference between any two values.

These stochastic elements may comprise some parts of the message. So it may not be a matter of transmitting all state change messages to the replicas, a two-tiered approach may need to be adopted to identify or isolate the non-deterministic elements: a copy of every message is sent to the replicas, when the data is modified, the change is noted as occurring from a deterministic source, or a non-deterministic source. The result is then propagated.

Figure 3.2 Identifying deterministic/non-deterministic data modifications

```
Process P →
        private Array a;
        op DoSomething( Param ) →
            for counter = 1 to a.size
                a[counter] := Param * RandomNumber();
            endfor
            synchronisation point
        endop
endproc
```

Figure 3.2 illustrates an example situation. In this case, the `DoSomething` operation is non-deterministic. The local variable `a` is updated using a random number generated on every iteration of the loop. This is the non-deterministic element of this process. It may be better to update the replicas after the `for` loop has taken place – i.e. the non-deterministic part is finished. The ThreadMarks system [3] calls such points *barriers*. Each barrier has a unique integer identifier. Every process that contains the barrier call will wait until all the other processes have read it – a type of multiple rendezvous. When all processes reach these barriers, the data pages are updated. Something similar should occur for the data replication of the fault-tolerance algorithm that will be used. The data sent and used by other processes in the system – the shared data – should be encoded into deterministic components and non-deterministic components.

There are a couple of reasons for synchronising after the `for` loop. In the example, the shared entity is the array `a`. If the replicas were to synchronise at every iteration of the loop, a lot of messages would be exchanged based on the number of elements in the array, `a.size`. Exchanging state after the loop has completed, reduces the number of messages from `a.size` to one. This message would contain `a.size` elements. Secondly, if state was updated at *each* iteration, there is a danger that a process could become faulty during the loop. This could leave the shared array in an inconsistent state as it would contain partially valid values from a defunct process. Traditional DSM systems would update at every data change, i.e. *every* iteration. A solution to this could be to have the user insert explicit synchronisation points. While this is a valid way of solving the problem, it does not address the issues fully, as it open to user error. These sort of state changes can easily be noted by a compiler, which can generate the synchronisation code.

A similar problem exists for other forms of non-determinism. How do the replicas of a process agree on the action to be taken when non-determinism occurs? Tough this is beyond the scope of this work, let us examine one possible solution. Consider a situation when a process is executing a guarded command and more than one guard becomes valid. One of these will be selected non-deterministically. Somehow all the replicas need to agree on which guard should be executed. One possible solution, when a non-deterministic event occurs, is to have each replica notify the others which action it has selected. When all the replicas have notified the others of their intention, a vote is taken, this will determine which action will be performed by all the replicas. Clearly this approach has significant performance implications and is not suitable for all situations.

3.5 Proposed Process Model

In this thesis a hybrid solution is proposed as the implementation strategy for the process model. Let the replicated units be the data contained in the processes; this is what will be changed by clients and internal operations. All state changing operations must be known at compile time so that they can be correctly propagated to the replicas. The synchronisation used should be flexible to allow multiple rendezvous as well as being non-intrusive at the programming model. The proposition is to use *communication channels*, as in Ada and SR, providing (synchronous) message passing, RPC and rendezvous capabilities, while a distributed shared memory approach is adopted for storing the state of a process. The DSM element is hidden from the users; they program to the communicating channels model. The DSM aspect is used by the executive for synchronisation of replicas.

One of the main differences between the suggested process model and existing implementations is the use of replicas and state. Both the DSM approach and Lamport's [14] approach have *all* the data of the system replicated at *every* node. The approach adopted by ftdl is to replicate the process (and their state) at nodes which comprise the replica group. All the replicas in a replica group coordinate their values and communication to ensure their data integrity. Thus communications only need to be transmitted to all the replicas in a replica group, not to all the nodes. Other systems broadcast this data and all nodes in the system process the event, updating their local replicas. Replication is used as a locality of reference and not to support fault tolerance.

The local variables of a process comprise the shared data of the replicas. The granularity of replication is the set of process local variables. This data can be changed through the use of incoming data on a process's communication channel or through normal program execution – i.e. data is changed not as a result of any communication. An interesting optimisation can be performed by the compiler. It can check if a remote invocation will cause a deterministic change; if it will, there is no need to synchronise after the invocation as all replicas will perform the same deterministic invocation. However, if the call is non-deterministic then the changed data needs to be synchronised. The DSM approach is used when transferring state; state transfers occur at checkpoints (such as after the `for` loop in Figure 3.2). Other communications use a modified SPROC /message passing mechanism.

Using this hybrid approach, when the processes synchronise, the language runtime system can update the replicas. This update scheme will need to be used in conjunction with updating the processes as well. The manner in which replicas are updated will vary with the type of computation. For example, non-deterministic or time intensive portions should only be updated after the main processing is complete; other updates may just require forwarding the change request to the replica. This model also works without processes exchanging communications. Consider a system comprised of one process, this is not distributed, as only one process exists and it performs all computations locally, without the need to interact with remote processes. The ftdl model still allows for replication of this process, with all state updates occurring at the correct points.

3.6 Fault Tolerance Through Language Extension

This section examines current approaches it extending existing languages to develop fault-tolerant systems. The design goals for any extensions to a language must provide enough functionality to allow fault-tolerance while trying to keep the cost associated with this to a minimum [5]. There should be little intrusion into the language's programming model and these extensions should respect the constraints and design of the language to ensure the added facilities are readily and easily adopted. Fundamental to the interprocess transactions which take place is the integration of the host language's process model (if it exists) with the required one for fault tolerance. If the language does not support a process model, then one will need to be designed in addition to the fault tolerance extensions.

Extensions to two existing languages are presented in the following sections. Section 3.6.1 describes an enhancement of C++ programming language [21] and Section 3.6.2 presents an extension of Linda [2]. Both of these host languages differ considerably; C++ is designed to be a single process environment which can be extended by the use of libraries to provide facilities for inter-process support. Linda is designed to be a language which lends itself to distributed computing through its use of shared memory. The model for Linda to provide communication and interaction between processes through a shared memory.

3.6.1 Avalon/C++

Avalon/C++ [11] is an extension to C++ that uses a transaction based model of concurrency. The system is insulated from faults by using two-phase commit protocols which each transaction. For non-database systems, where the execution of a set of operations is not arbitrary, the extra cost associated with these two-phase commit protocols is not required [5]. Avalon/C++ extends C++ through the use libraries and language extensions.

The following types of failure in a distributed system has been identified by Detlefs *et al.* [11]:

Local Storage Failure. Nodes can crash, maybe destroying local storage. Thus local storage is not a reliable medium.

Communication Failure. Communications can fail because of lost messages or network partitions.

According to Detlefs *et al.* [11], processes can reside in three types of storage: volatile, non-volatile and stable. Volatile storage can be thought of as the main memory of a machine; non-volatile is the local hard disk/persistent store. This store too can be subject to failure. Stable storage has a high probability of surviving crashes and other failures. An example of this could be replicated data throughout a distributed system. Logging each transaction that modifies a recoverable object to a stable store allows all the transactions to be 're-played' to a new process[11]. This allows the new process to have the same state as the failed one, prior to the failure.

Crashes themselves can be divided into two classes – *node* and *media failures*. A media failure will destroy both volatile and non-volatile storage. A node failure will only destroy volatile. This is the more common of the two.

The *write-ahead logging* protocol can optimise recovery from node failures. The virtual memory pages containing the object are `pinned` into volatile memory; they cannot be returned unless they are `unpinned`, which is after the object is modified. All the modifications are also logged to the stable store. This ensures against media failures. The `pin` operation ensures that no half modified objects are returned to non-volatile storage. This is akin to removing a book from a shelf where no one else can access it until it is returned.

The usage of `pinning`/`unpinning` is much like that of mutual exclusion's `wait` and `signal`: before data is modified the process `waits` for access to it. Once it has it, no other process can modify the locked object. When the modifying process is finished, `signal` is called. However, this synchronisation primitive has many failings, one of which is the matching of `waits` and `signals`. Another problem is that of recovery; if a process holding a mutex fails before it can `signal` that it is finished with the resources, deadlock can occur as subsequent processes will attempting to obtain the mutex that is locked by a failed process. This is a concern.

Components obtain the properties of *serializability* and *crash recovery* by inheritance. The authors state that these systems must satisfy application-defendant consistency: i.e. applications must provide their own serialise and recovery methods. This is one aspect where extending an existing language leads to compromises. It is much more difficult for the application programmer to keep track of how to serialise and how to recover from crashes than the actual runtime system of the language itself.

Transactions are sequential processes which are:

Atomic. This gives transactions the ability to be serializable (transactions appear to execute in a serial order).

Transaction consistent. A transaction either succeeds completely and commits, i.e. aborts and has no effect.

Persistent. Effect of transactions survive failures.

Transaction semantics are provided by atomic objects. Atomic objects ensure serializability, consistency and persistence. A transaction has a unique parent, a set of siblings (possibly empty) and sets of ancestors and descendants. A transaction can be considered its own ancestor or descendant. Transactions can be nested. Aborting in a parent rolls back a committed child's changes. The effects of a transaction only become permanent once a commit takes place at the top level.

An application in Avalon/C++ is made up of servers. Each server is made up of a set of objects which export the available operations of the server. Objects in a server can be stable or volatile. However, guaranteeing atomicity may be expensive, so those objects which require it can be so designated.

The communication between processes is done by invocations of operations of a process. Avalon/C++ receives its transaction management, IPC, commit protocols and atomic crash recovery from the operating system (OS).

Implementation

To allow the application fault tolerance, the developer must use inheritance to provide the functionality to those objects which are identified as critical and required to be fault tolerant.

1. **recoverable**. Provides primitives to ensure persistence. This implements the write-ahead logging protocol defined above. recoverable is a class defined in Figure 3.3.

Figure 3.3 recoverable class declaration

```
class recoverable
{
    public:
        virtual void pin( int size_to_pin );
        virtual void unpin( int size_to_pin );
};
```

   Whenever a recoverable object is modified, the pinning statement is used. This is an extension to C++ used by Avalon.

   Consider the example: there exists a class CustomerTable which is a recoverable list of customers. Figure 3.4 contains the code.

   This method is clearly very intrusive to the implementers. It is an added task which must be done in normal implementations. A lot of the fault-tolerance strategy is exposed to the programmer and this method of implementing pin() and unpin() can easily cause errors - what happens if all the data members of an object are not pinned? The advantage to this approach is that the programmer decides what is to be deemed recoverable, and what is not.

2. **atomic**. Derived from recoverable and provides primitives for atomicity in terms of read and write locking.

   Locking ensures serializability. Obtaining a read or write lock may suspend the calling process until holding processes are finished their transactions. While a **read_lock** is held, a requester for a write lock will be suspended until all the read and write locks are released for that object. This type of behaviour can lead to deadlock - where two or more processes are waiting for the other to release its lock. Locks are obtained by calling the respective methods of the atomic class in Figure 3.5.

   Using this approach, the user must partition the processes into readers and writers. It is thus not a general solution.

Figure 3.4 Avalon/C++ CustomerTable Example

```
class CustomerTable : public recoverable
{
public:
    // ctors, dtor omitted for clarity
    //
    AddNewCust( Customer & );
    virtual void pin( int size_to_pin );
    virtual void  unpin( int size_to_pin );
    ...
private:
    DataStoreImplementation CustStore;
    ..
};
CustomerTable::AddNewCust(Customer& NewCust)
{
    // The following is equivalent to:
    // pin( sizeof(*this) );
    // CustStore.Add( NewCust );
    // unpin( sizeof(*this) );
    //
    pinning(this);
    CustStore.Add(NewCust);
}
void CustomerTable::pin( int size_to_pin )
{
    recoverable::pin( size_to_pin );
    // forall data members -> pin()
    //
    CustStore.pin(sizeof(CustStore));
}
void CustomerTable::unpin( int size_to_pin )
{
    recoverable::unpin(size_to_pin);
    CustStore.unpin(sizeof(CustStore));
}
```

3. **subatomic**. Derived from `recoverable`. Like `atomic` but offers finer-grained control over synchronisation and recovery. This type has been provide to enable higher levels of concurrency and efficient recovery for more complex types. To this end both 'short-term' and 'long-term' synchronisation is provided. Short-term provides operation consistency and mutual exclusion between concurrent accesses (e.g. using semaphores or monitors); whereas long-term will ensure that the effects of transactions are serializable. The `subatomic` class is shown in figure 3.6.

   Every subatomic object contains a short-term lock which can only be held by one transaction at a time. Avalon/C++ has extended C++ by adding a **when** construct. This is like a conditional critical region; the calling

Figure 3.5 `atomic` class

```
class atomic : public recoverable
{
          ...
public:
    virtual void write_lock();
    virtual void read_lock();
          ...
};
```

Figure 3.6 `subatomic` class

```
class subatomic : public recoverable
{
          ...
protected:
    // These implement short term synchronisation
    // and object consistency
    void seize();
    void release();
    void pause();          // release lock, wait a bit,
                           // obtain lock
private:
    // The following are for transactional consistency
    // These functions are NOT called directly by the
    // user but by the runtime system when necessary
    //
    virtual void commit( trans_id& );
    virtual void abort( trans_id& );
};
```

process calls `seize` to obtain the lock and release when finished . The language guarantees that no partial effects will be observed if failure occurs while in the when construct.

3.6.2 FT-Linda

FT-Linda is an extension to the Linda programming environment which provides for the construction of fault tolerant systems. There are two potential problems in Linda that have been identified after a failure has occurred: lost tuples and duplicate tuples[5]. Both of these are symptoms of the lack of sufficient atomicity in Linda. Lost tuples come from incomplete update operations. The tuple is first removed from the TS and then immediately deposited with the new value:

$$in(\text{“}Var\text{”}, 10) \qquad \text{(i)}$$

$$out(\text{“}Var\text{”}, 20) \qquad \text{(ii)}$$

If the process fails between (i) and (ii), the tuple will have been removed from the TS; whenever subsequent reads from another process occur, they will be indefinitely blocked. Ideally the update operation should be a single atomic command.

The duplicate tuple problem lies in the **bag-of-tasks** model of programming. Under this scheme, a task tuple is taken from TS and a worker process, from a set of worker processes will take the first available task. If as a result of this task, subsequent tasks are generated, these too are placed into the TS. Clearly this may also suffer the lost tuple problem; but even if a solution existed, there is a danger that a worker will fail while it has not fully completed its task. Worse, it may have deposited some of its results. When another worker gets the uncompleted task(i.e. the one that failed), it will also generate some of the tuples already in the TS. This is the *duplicate tuple* problem.

Fault tolerance can be added to Linda by providing a stable TS and providing better support for the atomicity of TS operations. From [11] we see that truly stable storage must survive node and communication failures. To this end, stable TS's are implemented using the SPROC approach [18] by replicating the TS's across different nodes. Each replica is updated using atomic multicast. The TS operations are disseminated together in a single multicast message to all replicas. The message will then be executed in the serial order dictated by its contents. This method is relatively simple to implement as well as reducing the number of messages when compared to the transactional approach. Atomic multicast guarantees ([18, 5]) that if the commands sent to a SPROC are deterministic and executed atomically with respect to concurrent access, the state of the SPROCs will remain consistent.

The extension to Linda as described by Bakken and Schlichting [5] is a good example of integrating fault tolerance neatly and well into the host language. This is a good approach as the new features added can be easily adopted by users of the language. The limitations of this enhancement are that only fail-stop faults are recognised and arbitrary or Byzantine failures are not tolerated.

3.6.3 Possible Enhancements to FT-Linda

Many distributed languages use channels as a means of communication with remote processes; instead of using that method, shared distributed memory should also be investigated. This has the advantage that changes can be easily tracked and the data exposed would be core operational data. Using this with a variant FT-Linda's private tuple space may help to provide better recovery mechanisms.

As all the data a process is working on is in its private TS then, if it faults, another process can be brought in and given the private data.

Another reason for mentioning distributed shared memory support is that some applications may require it instead of, or with, the standard network (communication channel) method.

When data is changed in a process/shared memory region, a distributed callback facility would be a welcome addition. This would remove the requirement that polling must be done, or that each application must supply its own manner of doing so. The callback can be on data change notifications, data request notifications or any user defined event. For example, whenever the price of stock changes, all processes interested it will be notified.

3.7 Architecture of Selected System

This section describes the implementation approach adopted to enable fault-tolerance for this work. All the elements which comprise a distributed application must be made fault-tolerant but current systems only provide fault tolerance at the language and runtime levels.

The following constitutes a typical architecture for a distributed environment: user code, interface to distributed executive, distributed environment executive and network layer interface.

At the programming level, the user enters a statement to invoke an operation on a remote entity:

```
REMOTE_REF.OP(PARAM1,···,PARAMn)
```

Here, the operation *"op"* is being invoked on the remote object *"remote_ref"*, taking n parameters.

The compiler translates this simple statement into a series of interactions with the distributed executive, as well as producing the normal outputs from a compiler (e.g. generated machine code). For this operation to reach the remote node, it needs to be converted into a *message*. This contains at least the destination identity (i.e. the recipient) of the message and each parameter of the operation.

Each parameter needs to be inserted into the body of the message. This is one of the tasks of the interface to the executive.

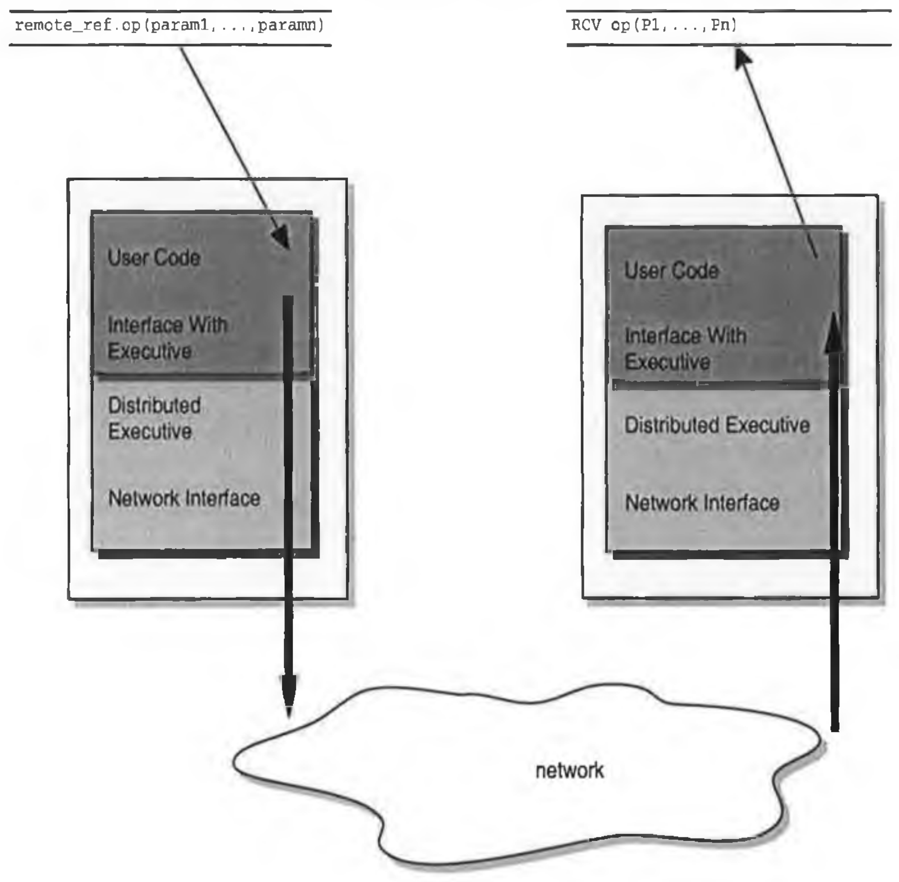
Once the message has been constructed at this high level, it must become a network message, that the network layer can use to deliver it. Close collaboration is required between the executive and the network layer interface.

Some communication protocols require the route from source to destination to be set-up in advance or alternatively the network layer may actually construct this route.

Once the message has been converted into something that the network layer can understand, it is transmitted to the remote node. Analogous actions take place at the destination node as on the sender when the message is received.

On receipt of the message, the network layer removes the network information and passes the new message to the distributed executive. Some protocols

Figure 3.7 Traditional distributed application architecture



will *authenticate* the sender of the message at this stage, using only the sender for the security references. Other security schemes will do this authentication at the executive level.

The executive must now decide *which* of its entities is the recipient of this message. It then passes it to that entity.

The executive interface code will take this message, extract each parameter and then passes control to the user code.

The receiver issues the following statement to indicate to the executive that it is waiting for a message on its communication port named `op`:

RCV `op(P1,···,Pn)`

In order to support fault-tolerance, each of these components need to be modified. As we will show, it is vital for *compile* time information to be gathered and passed on to the runtime environment to allow for fault recoverability. Figure 3.7 shows this interaction. This figure depicts a traditional but non-fault tolerant architecture.

3.7.1 Fortifying the Network

The *network layer* is primarily concerned with routing a message to its destination. Lamport [14] notes that for the communication layer to be fault tolerant, it must establish various alternate *paths* to destination nodes. As no direct link may exist between a sender and its recipient, the communication protocol may use intermediate nodes in order to ensure that the message arrives at its destination. This allows the following two conclusions to be made about the sending a message from node i to j:

1. Process j receives a message with the timestamp T at time $T + n\delta + \epsilon$ on its clock, where n is the number of nodes the message must be routed through to transmit the message, δ the processing time required at each node to process the message and ϵ, the greatest value that the clocks in the system can differ.

2. Define γ as the path from node i to j, containing the intermediary nodes $k_0, k_1, k_2, \ldots, k_{n-1}$. If all the process and communication links on the path γ are nonfaulty then the message sent is the one sent by node i.

This second point is very important. In order to be able to recover from Byzantine failures, we need to know which node has sent a message. If the network can guarantee supplying the originator of the message, it can help in guarding against Byzantine failures. Authentication is required because each replica needs to be able to verify that the replica which says it sent a message *actually* did. Under Byzantine conditions, replicas can impersonate other replicas by using another replica's messaging identification and not its own.

Another requirement of the network is that as long as there is some connection between any two nodes, the network can guarantee that the message will be delivered.

The network layer can ensure that transmissions and routing of messages is nonfaulty, while the executive and language interface can ensure that the messages themselves are not faulty.

3.7.2 Detecting Failure

The executive can check the originator of the message. For example, this information can be used to check if that node is in its suspected faulty list. It can store the message until it has a sufficient number in a cache in order to check the integrity of all the messages sent by replicas against each other. This is the main mechanism for fault inspection.

An important component in detecting failures is ensuring that the parameters of the message are correct. The newly received message will have its parameters extracted and checked against the others in the cache. At the user coding level, there should be no need to worry about fault handling.

The executive also has the responsibility for dealing with failures. It must evaluate the options it has when failure occurs. This can include reconfiguration and new replica activation [14, 18]. It must coordinate its activities with the other non-faulty nodes.

3.8 The Implementation Model

Many languages have tried to add-on support for fault tolerance after the language has been defined. Though this is possible, it does not yield an efficient solution, only compromises [21]. Both FT-Linda [5] and Avalon/C++ [11] attempt this.

Avalon/C++ does this through inheritance of different objects – `atomic` for example. But this approach does not provide transparent access to fault tolerance. It is dealing with a much lower level, the transaction level. As [18] showed, dealing with faults must be done at the system level, and left to the runtime portion of the language. This allows transparent detection and recovery.

Though many implementations deal with the syntax of new language features, a protocol must be established for the accurate detection of faults and how transparent system recovery can be managed.

By examining current approaches to providing fault tolerance, this chapter has produced a model for a language and environment to support fault tolerance. One of the things shown by the literature is that extending an existing language to provide fault tolerance does not result in a solution that is easily implementable or provides transparent failure detection and recovery. As a result, a separate language has been developed which provides the ability to develop fault tolerant distributed applications. This language is known as the *Fault Tolerant Distributed Language* (ftdl). It is a basic language which provides a limited set of features. This is used as an illustration of what is required in a language for fault tolerance: it must have a notion of distributed entities and communications with those entities. ftdlcomprises of a language grammar, compiler and a runtime support environment. The combination of language and runtime into one package allows for effective fault tolerance, as will be shown in subsequent chapters.

The process model of ftdl supports a message passing architecture between communicating processes. Transmission of messages requires the sender to indicate the process in the network which will receive the message. All transmissions carry the identifier of the sender, to allow authentication. This model has been selected as it supports the addition of replicas and the formation of replica sets quite easily. Currently, only synchronous message passing is supported.

The runtime system support is an extension of the capabilities of Schneider's [18] SPROC approach and FT-Linda[5]. The network model is based on Lamport [14] and Schneider [18]. ftdladopts a different replication model. Both Schneider [18] and Lamport [14] expect all the nodes in the system to hold replicas for each process. Lamport [14] does allow his model to run replicas on a subset of the nodes, but this is does not change the general approach. These tend to provide fault tolerance easily for applications that work on a shared-memory principal, like the FT-Linda [5]. All the nodes in the system contain their own state, this gets altered based on messages received from the network. An implied uniformity exists between clients and servers.

ftdl supports an independent runtime model. Each replica is an ***independent*** entity. Clients must explicitly obtain references to (remote) servers for invocations. The invocation parameters are propagated to the correct replicas, but each must process it differently and then dispatch the results. The replication and group communication is an extension to the distributed model and we use those entities to provide the fault tolerance. This allows replication of potentially computation intensive processes to be done across different machines. Our model, however, introduces a significant overhead in communication times.

The following chapters describe the design and implementation of our fault-tolerant simulation. We begin by describing the simple language which was created to describe the components required by a compiler, the generated (object) code and the language runtime interfaces required to produce a system that supports fault tolerance and can recover from failure.

The sections which follow describe the impact on the distributed executive (using the inputs from the compiled code) and the network itself.

Chapter 4

A Language and Environment for Providing Fault-Tolerant Distributed Systems

The distributed architecture described in Chapter 3 has been implemented. This implementation is presented here as the ftdl environment. The ftdl implementation consists of a simple programming language, a compiler, a simulation machine and a runtime system. The language is small, but sufficient to provide the necessary requirements for a language which supports fault tolerance. A runtime system has been written which implements the process model described in the previous chapter.

In order to measure the effectiveness of the proposed model, a simulation machine was created. This machine has a limited instruction set, that is interpreted by the code generated by the ftdl compiler. This machine is integrated into the simulator. This simulator provides an environment for multiple virtual distributed nodes to exist. These nodes represent the nodes in the network on which the distributed application is run. They are, however, run within the same simulation envrionment which is run as a single process. The ftdl runtime system interfaces with this simulator.

4.1 Language Overview

The language implemented illustrates the key requirements for fault tolerance and uses features of CSP [13] and SR [4]. The user declares a *protocol*. This protocol contains *communication channels* which is its interface. These channels are used by clients to communicate with the protocol.

In ftdl model, the implementation of a protocol lives in a process, that is, it can be said that a process *implements* a protocol. Clients send messages to other processes via the communication channels defined in the protocol. A protocol is thus an interface to a process's functionality. The current implementation maps every protocol definition to a process.Messages refer to network messages sent amongst processes. A message is a formated sequence of bytes. The term *operation* refers to the communication channel declared in the protocol definition.

Figure 4.1 A Protocol Declaration in ftdl

```
TIMER PROTOCOL →
        START      : Integer
        STOP(ID    :Integer)
        DELTA(ID   :Integer) :    Real
end  TIMERPROTOCOL
```

Users invoke *operations* on a protocol implementation, the language runtime converts these operation into messages which are sent communication channels. The terms *message* and *communication channel* are artifacts of the network layer, whereas *operation* and *protocol* are terms belonging to the language programming model.

Figure 4.1 declares a protocol for a simple timer. A timer is created by invoking the START operation, this returns a timer identification which must be used in subsequent operations. The STOP operation stops the *id*th timer; DELTA returns the lapsed time between START and STOP operations.

The operations declared in a protocol can be implemented as asynchronous procedures, running concurrently with other processes, or they can be used as static, synchronous 'pipes' from which data is read, and results sent down. The model presented here concentrates on the synchronous features.

4.1.1 Language Description

Variable Types and Operations

ftdl supports the following primitive types shown in Table 4.1. In addition, the user can declare a reference to a *protocol*. This reference is used to communicate with the remote protocol implementation.

Data Type	Description	Sample Value(s)
Integer	Signed integral type. Range: from −2147483647 to 2147483648	`-1,20001`
Real	Signed Floating point type. Range: from $\pm 3.4028234 \times 10^{38}$ to $\pm 1.40239846 \times 10^{-45}$	`1.5,-245.56778`
Char	Simple ASCII character type	`'A'`
String	Sequence of characters forming a string of fixed length	`''A STRING''`

Table 4.1: Primitive types supported by ftdl

Local variables and parameters are declared by giving the name of the instance followed by a colon and its type.

The following is declarator of an integer local variable called `LOCAL` :

```
LOCAL : Integer
```

For protocols, the declaration is the same but the type name is the name of the protocol. For the TimerProtocol example given in Figure 4.1, a local variable `LOCAL_REF` is declared like this:

```
LOCAL_REF : TimerProtocol
```

The return type of an operation declared in a protocol is given after the channel name:

```
AN_OPERATION:INTEGER                              (i)
AN_OPERATION(A_PARAMETER :  STRING) :INTEGER      (ii)
```

Declaration(i) declares the operation `AN_OPERATION` which does not take any parameters, but does return an integer. (ii) declares an operation that requires a **String** parameter and also returns an integer.

The language supports the arithmetic and Boolean expressions shown in Table 4.2.

Expression Type	**Operation**
Boolean	`and`, `or` and `not`
Arithmetic	addition(+), subtraction(-), multiplication(*) and division(/)

Table 4.2: Supported operations

Protocol Declaration and Usage

A protocol declaration consists of the protocol name followed by an optional list of its supported operations. This declaration defines the type of messages an implementation can receive and the expected reply from the operation. Figure 4.1 gives an example of a protocol declaration.

Once a protocol has been declared, an implementation must be provided. An implementation for a protocol must be of the form:

```
define protocol_name
        ⋮
        implementation statements
        ⋮
end protocol_name
```

Every protocol implementation needs a unique name. This name allows clients to connect to this particular implementation. The name is assigned by the **ID** statement in the implementation process:

```
ID("UNIQUE IDENTIFIER")
```

A client specifies the implementation it wishes to use by passing an identifier string – the same value as the one declared in the implementation – to the **Init** operation for remote references:

```
aRemoteReference.Init("UNIQUE IDENTIFIER")
```

The runtime system will establish a connection from the client process to the implementation specified to the **Init** operation.

4.1.2 Control Statements

Control statements have been kept to a minimum. The ones provided are sufficient to implement most classes of problems. An **IF** statement and **REPEAT...UNTIL** are supported.

The **IF** statement evaluates its predicate as a Boolean. If the predicate evaluates to `true`, then the body is executed. Otherwise, control passes to the first statement following the last statement of the **IF** block:

```
IF predicate THEN
        ...if body ...
end
```

The **REPEAT**... **UNTIL** statement executes, sequentially, the body of the **REPEAT** until the predicate of the **UNTIL** evaluates to `true`, at which point control is transferred to the first statement following the **UNTIL** statement:

```
REPEAT
    ...repeat body ...
UNTIL predicate
```

4.1.3 Distributed Operations

Remote invocations are performed by invoking on an operation of a protocol, using an initialised remote reference and passing the required parameters.

For n parameters a call has the form:

```
InitialisedReference.OperationN(param1, ..., Paramn);
```

An operation invocation not requiring parameters takes the form:

```
InitialisedReference.Operation0;
```

Operations which return values can be assigned to local variables, just like normal arithmetic expressions.

An implementation indicates that it wishes to synchronously receive data on channel *chan* by using the receive statement, **RCV**:

```
RCV chan( Param1, ..., Paramn);
```

where `Param1, Param2, ..., Paramn` are locally declared variables. These receive the values sent by the initiator.

Execution resumes after a remote invocation when the receiver issues a:

RETURN *chan* RETURNVALUE;

where RETURNVALUE is a locally declared variable. This value is passed back to the sender.

For operations that do not have return values, execution resumes immediately after the **RCV** on the receiver side. Effectively, operations that do not return values can be mapped to the statements shown in Figure 4.2 the receiver side:

Figure 4.2 Mapping of an operation which does not have a return a value

RCV *no-return-chan* maps to $\rightarrow$ **RCV** *no-return-chan*;
RTN *no-return-chan*;

4.1.4 Compilation

Protocols are declared and implemented in a text file. This file is used as input into the compiler `ftdl`. For example, to compile `test.dl`, the following is command is issued:

```
% ftdl test.dl
```

This produces a GEN file (with extension `.gen`). This is a formatted file which contains the simulation machine instructions to be executed. The format of the GEN file is described in section 4.2. The GEN file is used as input to the simulation machine, and is executed within that environment.

4.2 The Simulation Environment

The ftdl compiler produces a GEN file. This is used as input to the simulator to set up a process. This process represents a node on the network.

The format of the file resembles the format of an operating system's executable. The first few bytes are used by the loader to set up the execution environment for the process. This is similar to an activation record for functions as described by Aho, *et al.* [1] and Sorenson, *et al* [19].

Typically there is one protocol implementation per GEN file. The main body of the implementation is preceded by setting up the stack for local variables. This requires the number of variables, their size and names (for ease of reference in the simulation).

Following the local variables on the stack, space is reserved for the communication channels. The names of the channels are in the file to aid usability in the simulator environment.

As only synchronous operations are currently supported, the channels can be static entities; in an asynchronous environment a facility for having these as concurrent executing entities would be useful as in SR [4].

Following the channel area, the execution instructions are placed. The instruction set for the machine resembles simple assembly language. This is described in section 4.3.

4.3 Simulation Machine Instruction Set

This section presents the instruction set for the machine used by ftdl. This machine exists within the simulator. It has the responsibilites of acting as a *loader* of an image into the simulator and sending events (timer and network) to the correct protocol. The image (code) is read from the GEN file produced by the ftdl compiler. Following this section, the runtime system and the simulation environment are presented.

The examples presented are shown as the ftdl source mapped to the machine instructions to they correspond.

4.3.1 Standard Instructions

The standard instructions consist of arithmetic operations, Boolean operations, stack manipulation, comparison and flow control instructions. These are the type of instructions found in most languages, in one form or another. Table 4.3 contains the list of standard instructions, their runtime requirements and semantics.

4.3.2 Executive Interface Instructions

There are three dedicated instructions in our machine which provide an interface to the distributed executive: send, receive and reply. When these are executed, the executive expects the stack and any marshaled buffers to be set up for use. The distributed executive is the portion of the overall runtime envrionement which simulated.

Sending a Message

Recall that a **SEND** operation is performed when the user invokes an operation on a remote, initialised reference:

```
InitialisedReference.SomeChannel(param1,...,Paramn);
```

The generated code will push the values for *param*1 to *param n* onto the stack, from left to right, in marshaled form. Following this, the name of the remote channel is put onto the stack – in this case, SomeChannel – and a reference to the remote protocol. This reference contains the information for the binding between the local node and the remote implementation, for example, the target address and route to the target.

Finally the **SEND** instruction is placed on the instruction stack. This requires a parameter indicating if the operation returns a value or not.

If a value is returned, it is pushed onto the data stack. Otherwise no value is pushed.

Instruction Class	Instruction	*Behavioural Notes*
Boolean	**and**	Performs operation on top two stack values. Result placed on stack.
	or	*ibid.*
	not	*ibid.*
Arithmetic	**addition**	ibid
	subtraction	*ibid.*
	multiplication	*ibid.*
	division	*ibid.*
Comparison	**greater than**	*ibid.*
	less than	*ibid.*
	equality	*ibid.*
	compare	Remove the topmost two values from the stack; See if they are equal; Push the boolean result onto the stack.
Control Flow	**branch if true**	Branch to the address given as an argument if the top of the stack's value is true; otherwise continue to the next instruction.
	branch if not true	inverse of above.
Stack Manipulation	**push**	Place the argument of the operation onto the top of the stack.
	pop	Remove the topmost value from the stack.
	move	Remove the topmost value from the stack and place into the variable indicated by the argument to this operation.

Table 4.3: Simulation Machine Standard Instruction Set

When the runtime executes this instruction, it generates a message to be sent to the relevant set of replicas. It takes the marshaled data from the stack, appends it. This message contains the identifier for the implementation group (the string passed to the **Init** operation), the name of the operation and the marshaled parameters.

This message then filters through the remainder of the runtime system to be sent via the network. Execution does not resume until a reply message is returned from the destination.

Receiving a Message

An implementation issues a **RECEIVE** directive by supplying the channel from which it wishes to read data. The process is blocked until data is received, or the system fails. At system failure, the process' execution is halted. The ftdlstatement is of the form:

RCV *chan* (PARAM1, ..., PARAM*n*);

The identifiers *Param*1 to *Paramn* are local variables of the protocol's implementation. References (addresses) for these are placed onto the data stack. When the newly received message is propagated to the implementation from the runtime system, these variables are filled with the incoming data.

Like the **SEND** instruction, **RCV** requires a parameter indicating if a return value is expected (This is shown as an argument to the **RCV** machine instruction.) If one is not expected, once a successful receive is performed, a reply message is generated and sent back to the sender to allow it to resume execution. Otherwise, the reply is not sent back until a **RTN** instruction is encountered. The following section describes this instruction.

Replying to a Message

There are two forms of the reply instruction, **RTN**. One is the explicit form, which the user uses to send the return value from an operation. The implicit from is generated by the compiler for operations that do not return values. Figure 4.2 on Page 67 illustrates this latter form.

RETURN *chan* *return-expression;*

The runtime will construct the reply message for the operation when the **RTN** instruction is encountered. The *chan* argument tells the runtime which channel is issuing the **RTN**. A return value is marshaled into the message if one is expected – this an argument to the **RCV** instruction, described above.

4.4 The Runtime System

The implementation of an ftdl protocol exists in one process. At execution time, the number of replicas to be is used is specified. Each replica is a process running on it own machine, which is a node on the network. Each process which is a replica of a particular implementation is grouped together. This grouping is referred to as a *replica group* or *replica set.*

Fault tolerance is provided by the use of process replicas [6, 18]. Each replica is assigned to a group. Each group consists of replicas all implementing the same protocol. Each replica grouping consists of a group name, derived from the protocol name, and a set of member nodes. This set is managed by the runtime component, the CTXMGR , which is local to every process.

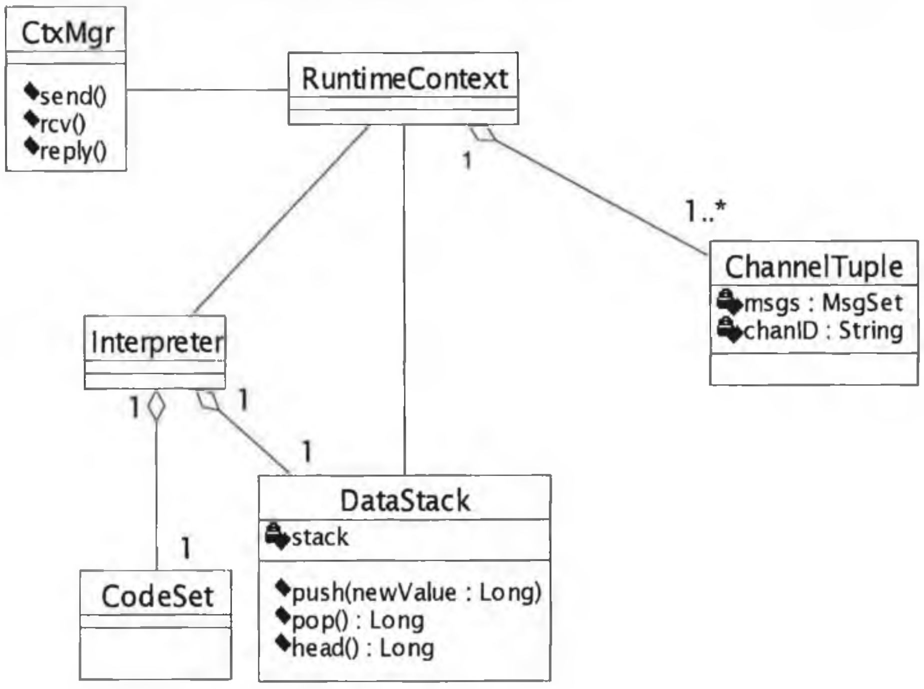
The following sections detail the runtime architecture and interactions of an ftdlmachine. We will examine how the system identifies faulty processors and handles failures. The mechanics of distributed directives will also be explained. Also described in this is the workings of the runtime system. The next section describes a brief architectural overview, subsequent sections describe the major components, fault detection scheme and the fault handling procedure.

4.4.1 Major Components

Local Node Component Interactions

The distributed system consists of a set of interacting processes, running at different nodes (hosts), connected by a network. Each local process contains the program to execute, extracted from the GEN file. This is coupled with an interpreter to execute the code.

Figure 4.3 The Runtime context component



From the GEN file, tuple sets can be created. These sets are arranged into tuple spaces. We have two tuple spaces: one for the communication channels and the other for local variable storage. The communication channel tuple set contains the operation names declared for the protocol implementation in the source ftdl file, along with a special channel – _repl . This channel is used by recipients of **SEND** messages to place their replies. The code, interpreter, data stack, channel set and variable tuple space form part of the runtime interface artifact, referred to in section 3.7. This interface is known as the RUNTIME CONTEXT . Figure 4.3 shows the major components of the runtime context.

The RUNTIME CONTEXT is the heart of an ftdl process. It drives the interpreter, that is, gives it an execution context; dispatches remote invocations to the CTXMGR ; routes incoming messages to the correct replica and feeds the data to the other components when requested.

The CTXMGR , in collaboration with the this component, has the responsibility to identify potentially faulty replicas by semantically checking each invocation's parameters from every participating replica involved in the transmission context. These checks compare the values of each parameter. The comparison must consider the *fragility* of the value, as described in Section 3.4 and format of the value. The runtime system knows how to do the comparison as it uses information gathered from the GEN file. All value comparisons are not always a simple case of examining a sequence of bytes but the format of the data must be considered. For example, if the values being compared is a reference to an protocol, what needs to be compared is the name of protocol implementation being used and ensuring that all the replicas are referring to the same one. RUNTIME CONTEXT and the CTXMGR provide the functionality of the runtime layer, as described by section 3.7.

The CTXMGR is concerned with dealing with other replicas. It manages the replica groups, identifies faulty replicas (in conjunction with the RUNTIME CONTEXT component), manages incoming connections, provides reply management and remote invocation delivery. One of its primary responsibilities is to provide a *context* in which remote operations take place. It creates and manages the contexts for sending data during an operation invocation, waiting for data on a channel and issuing reply messages to all the replicas in the group.

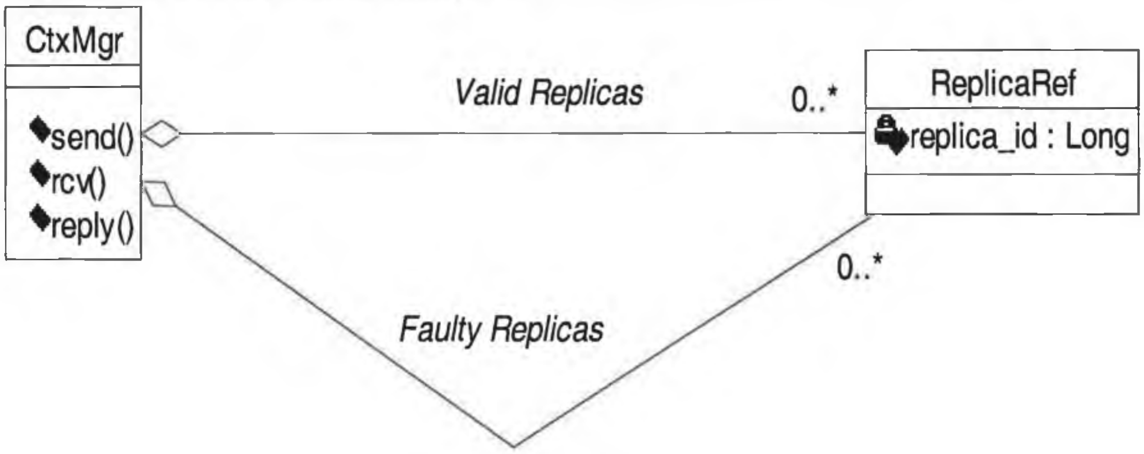
Inter-Node Interactions

Request Identifiers and Managing Messages

The CTXMGR divides each replica set into two sub-sets:

1. Valid Replicas. These are the replicas that are non-faulty.

Figure 4.4 The Context Manager



2. Faulty Replicas. These are the replicas identified as being faulty; they are not used in subsequent transactions.

Let S represent the replica set, F represent the faulty replicas, and K the valid replicas. Define $K = k_0, k_2, \cdots, k_{n-1}$ at time T. Define $F = f_0, f_2, \cdots, f_{m-1}$. Then $(S = K \cup F) \wedge (K \cap F = \emptyset)$.

When a message from node i is sent to the $n-1$ nodes in the valid replica set, its done as a specialised form of an *atomic broadcast* [18] to all the replicas in that replica group; this is known as *atomic multicast.* Recall that Lamport[14], defines one of the properties of the broadcast algorithm to ensure that all non faulty processes in the replica set receive the message. This is also known as the **agreement** condition, imposed by Schneider[18].

Figure 4.5 Request identifier composition

Sending Protocol ID
Destination Protocol ID
Operation Name
Occurance Point
Iteration Value (optional)

As the system only supports synchronous requests, transmitted by atomic multicast, order is not as much of an issue as in asynchronous systems [24]. The code generated by the ftdl compiler can provide a unique identification key for requests. This is composed of the sending protocol identifier, destination protocol identifier, the operation name and the occurance point of the request in the code. Each time a **SEND** instruction is encountered a counter is started, each subsequent occurance increments this number. The receiving process knows from the request identifier exactly which request this is for. Figure 4.5 shows the layout of the request identifier.

The message cache of the receiving CTXMGR keeps a queue for each expected occurance number for the given operation. The receiver protocol's CTXMGR manages a special cache for messages. Each operation has its own queue, each operation's queue contains sub-queues classified by the sending protocol. Each sending protocol queue also has a queue based on the occurance number described above.

If a **SEND** statement occurs within a loop, the occurance number is also subdivided. It contains two portions – the occurance point and an instance count, based on the number of times the loop has been executed. While the other identifier components are compile time entities, this last component is obtained at runtime.

Deriving Failure from Time

Once the message has been multicast to all the non-faulty replicas in the process group, the sending process allows them time to process and issue the reply message. As Lamport[14] states, the required time to wait is determined by network latency, the number of replicas executing the code, generating the reply and delivering the reply.

If a process i issues sends a request at time T, and assuming that all process are executing at roughly the same rate, the non-faulty process j will receive the message at $T+\delta$, and can execute at the time T_{proc}, where $T_{proc} = T+\Delta+\epsilon$, for a network where delivery of multicast message takes the same time as delivering one message. $T_{proc} = T + (n-1)\Delta + \epsilon$ for a network where a multicast message takes the sum of delivering a message to one node. In this equation, n is the number of non-faulty replicas in the group at time prior to T; Δ is network transmission constant and ϵ is a constant. If process j takes δ seconds (at most) to process the event and to generate a reply, it will issue a reply at time $T_{proc} + \delta + \epsilon$, which will arrive at i at time $T_{proc} + \delta + \Delta + \epsilon$. Call this time T_{arrive}. (These variables and equations have previously been defined in further detail in section 2.8.)

If the processing is not carried out at a similar rate on all the nodes, an upper bound can be established to factor into the equations. Note that the generated code can calculate the value of δ at compile time by measuring the amount of time it takes to execute the set of instructions when a protocol is dealing with the message on the operation channel. It thus computes a maximum time from all the channel processing times and issues that value, which can be used at runtime when determining the timing conditions and variables. This would be quite difficult to do if we did not have direct access to the compilation and machine language generation phases, so as to establish upper bounds on the execution times. These bounds become properties of an operation for the protocol.

With these equations, the process which issues a message can determine if a particular process has failed. This mechanism can *only determine fail stop* failures. Byzantine failures can be determined by checking each parameter of the message and by using the approach outlined by Tuerk *et al.* [24] and shown in the *Literature Review* in section 2.9.

Communicating Processes

Communication between processes occur through their process groups. Messages sent from one process go to all the processes in the target process group. Consequently a **SEND** statement like:

```
INITIALISEDREFERENCE.SOMEOP(PARAM1, ...,PARAMn);
```

is issued from the node P_i, where P is the a protocol implementation replica set, i is the ith replica. Initially the replica set contains t replicas, where t is failure tolerance level. The system will be able to withstand $t-1$ faults.

The **SEND** message is delivered from P_i to all the replicas in the target protocol set D. This is done through the atomic multicast feature discussed previously. The target protocol set initially contains t replicas. At time T, P contains n nodes and D contains m.

When a destination node, D_j encounters its **RCV** statement – for channel SOMEOP – it must wait for n **SEND** messages, one from each node in P. Then it can allow the execution to proceed to the statement following the **RCV** request, after the received messages have been validated.

4.4.2 Transmission and Reception Context Creation

A context is created when the user specifies a **SEND** or a **RCV** statement. These create *transmission* contexts and *reception* contexts, respectively.

Reception Context. When the interpreter encounters a **RCV** instruction, this message is passed to the runtime module to establish a *reception context*. Recall that a **RCV** statement is of the form:

```
RCV SOMEOP(P1,···,Pn)
```

This instruction informs the runtime to wait for at most t replicas from the same protocol implementation group to issue a **SEND** request for the operation *SomeOp*.

Initially, the number of replicas per replica group is t, as the system will tolerant to t faults. Over time, as different replicas fail at different rates, so the number of non-faulty replicas per group varies to being less than t as more failures occur per group.

The runtime environment transitions from normal execution to the *receive message* state. When the messages arrive, or the specified timeout occurs (based on T_{arrive}), a fault check is performed and if there is sufficient data left in the non-faulty set, execution is returned to the interpreter.

The context takes control from the normal execution to wait for the messages. The context does not know which replica group will be sending the messages, so it must monitor the incoming requests to determine which group is issuing the request. The same fault detection algorithm is applied to the sent data at time T_{arrive} to determine if the correct number of replicas have sent the invocation requests.

The local runtime environment keeps a map of replica groups to the replica information sets. These information sets contain the number of faulty replicas, the number of non-faulty replicas, the network identifiers for each replica and other system information.

Upon successful arrival of the desired messages, the data is unmarshalled, the context is destroyed and execution is resumed at the user level.

Transmission Context. A transmission context is created when a **SEND** instruction is encountered. This context is responsible for creating the transmission message, transmitting it and waiting for the required replies, on the _repl channel.

Creating the Transmission Message. This requires the parameters for the remote invocation to be marshaled and the header of the message filled in with the destination group identifier. These are obtained from the RUNTIME CONTEXT local variable area.

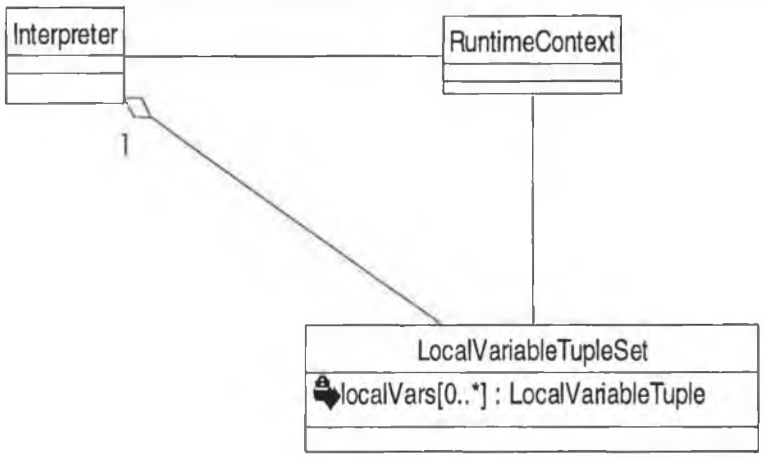
Message Transmission. Once the message is created, it is multicast to the process group identified in the header.

At the time of message transmission, all the non-faulty processes in the sending group's replica set will transmit the same message with the same contents to destination group, D. Thus n messages will be transmitted to m nodes.

The Waiting Game. When transmission is completed, the replica must wait for a reply from the destination group. The amount of time to wait is bounded by network latency, the processing rate of the replicas in the same group as the transmitter and the processing rate of the replicas in the destination group. All these factors must be taken into account when determining the upper bound of the time to wait before the failure detector is invoked.

Network latency is quite an important factor, as it affects when the message is first transmitted and when the reply is transmitted. There will be at most $2nm$ messages sent.

Figure 4.6 Runtime Context Local Variable Interaction



When replica i in the transmitting group P sends its message, it cannot assume that the other members in the group will send the message at the same instant. It must allow time for each group to execute the code up to the **SEND** statement and marshaling time. . Fortunately, as we have access the code at compile time, we can calculate this time, ρ. This result can be multiplied by some constant c for each process, allowing time for each to run at their own rate. ρ is determined at compile time, c is a runtime calculation that can be set as the system is running.

A similar time must be allowed for the processing of messages for the destination group's replicas. Let these times be represented by T_{wait}.

Once these values are computed, the sending process must wait for that amount of time, unless m replies come in before it. Then it can resume execution.

The transmission context has a significant amount of steps to perform in order to ensure a fault tolerant environment. Some of the steps are similar to the receive context.

When marshaling/unmarshalling parameters, they may contain information specific to the current process, e.g. the number of ticks on the clock. It is important that this information is marked in the marshaled message so it is not used as an absolute value in the fault detector.

4.4.3 Identifying Faulty Processes

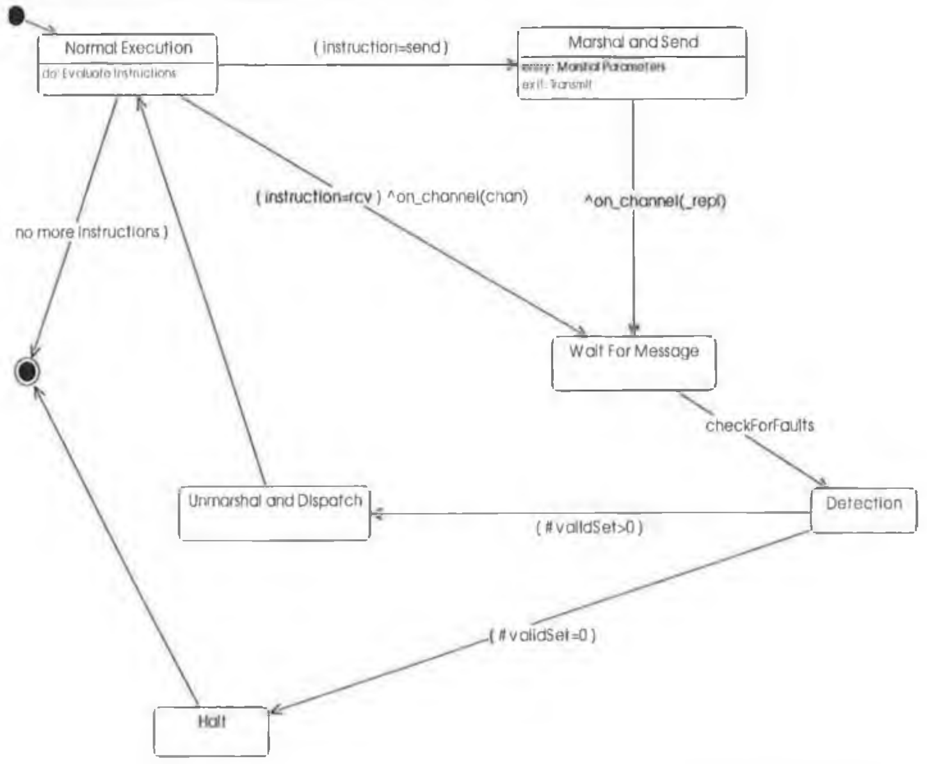
The fault detector is a component embedded in the runtime. The implementation is driven from the RUNTIME CONTEXT and require collaboration between it and the CTXMGR .A process is determined as faulty if it does not send (or reply) a message by a given time or if the data in the message does not agree with the other replicas' equivalent messages.

The detector is invoked when the process is in the `wait` state. This state occurs after the transmission of a message, when the sender is awaiting a reply message, or when a process issues a **RCV** instruction. In the `wait` state, a process waits for up to h messages to arrive, where h is the number of non-faulty replicas in the replica group which is issuing the messages. If h messages do not arrive by the time T_{wait}, the fault detector is activated.

A process indicates that it wishes to wait for a message on a given channel. The process then waits on this channel. Detection occurs by *each* replica in the implementation group. The detector is executed against the received messages from one replica group.

When the detector is active, the process is in the `detection` state. Figure 4.7 contains the state transitions that take place for **SEND** and **RCV** instructions. The detector attempts to recognize faulty processes in the sending replica set. It will recognize two types of failures, *fail stop* and a limited form of *Byzantine*.

Figure 4.7 State transitions of the runtime



Fail stop faults can only be detected after T_{arrive} has been reached. Those nodes which have reached this form of failure will not have sent any messages. These replicas are then removed from the replica set, and any subsequent communication from them will be ignored.

This recognition scheme has the disadvantage of potentially confusing process that have failed with those that are running slowly, or a network that is operating very slowly. Both of these alternatives are possible, as machines can become quite overloaded for short periods of time due to unplanned user activity (e.g. starting a window manager). Ethernet is known to perform badly due to the exponential back off characteristic as the network becomes busier [22]. This why it is vital for T_{arrive} to be accurately calculated and updated throughout the lifetime of the system.

The problem of a slow process trying to communicate with the system after it has been falsely identified as faulty can be addressed as a special case of replica activation, described in the *Future Extensions*, section 6.1.

Byzantine failures, unlike fail stop, must be detected while consensus is being established by the detector. Fail stop failures are detected *before* the main consensus [24] algorithm is run. The Byzantine detector is an integral component of the failure detection/consensus phase.

When the detector is run on the messages of a given channel's message cache, the first thing that is checked is the number of parameters sent in each message. These should be the same value. Those messages from replicas that do not conform to the determined value are marked as faulty. The fault could be malicious (as in Byzantine failures) or a genuine fault due to network transmission and reliability problems or a genuine fault in the processor. The correct value is determined through a consensus algorithm, described in section 4.4.4. It is worth remembering that for the special reply channel, **_repl**, all the messages should not have any parameters.

Once the cardinality of the parameters has been resolved, each parameter's value must be checked. Every parameter of the operation is used to check for faults.

There is a problem with checking parameters: we assume that the values will not differ. This may not necessarily be the case. The processes which send the message may receive their inputs from independent sources, or it may be stochastic. For example, each process may be connected to its own sensor. The values between the sensors probably have a tolerance range which is acceptable. When checking for faults using these values this should be born in mind. The current scheme does not take this into account, but an extension is proposed below whereby this can be added.

The parameters are checked one at a time. The same consensus algorithm is applied as the one used for determining the cardinality of the parameter set.

4.4.4 Achieving Consensus

Turek *et al* [24] describes the correctness conditions for a consensus protocol. A consensus protocol is correct if it maintains:

- **Consistency Condition**. When determining the value, all the participants [receiving replicas] agree on the same value and the decisions are final.
- **Validity Condition**. The agreed value must have been some replica's input (sent value).
- **Termination**. Each agent decides on the value within a finite number of steps.

The consensus algorithm, shown in Figure 4.8, utilised by ftdl's runtime, takes as an input the data set to check. This data set is assembled from all the replicas which have sent the messages. For example, if parameter i is being validated, then the data set is populated with the values of all the ith parameters in the queue for the reception channel from the same replica group.

The value that occurs most frequently is the agreed upon value which will be used. However, the frequency occurance of the value of the valid set, returned from the **determineValue** routine, must be greater than 1. If it is not, then we do not have concensus and the system, from the point of view of the current replica, will be degenerate as we have no means of discovering which replicas are faulty or not. This is the boundary condition and is highly undesirable. Generally the minimum number of replicas entering the consensus algorithm should be three. As voting is used to determine which replicas are faulty, three is the minimum number required to form a majority. If only two replicas are used, and they differ, then the faulty one cannot be identified. A third replica can be used to break this tie.

Does the presented algorithm meet with the criteria imposed by Turek *et al*[24]? It meets the **consistency condition** as the same data will be used

Figure 4.8 Consenus Alogrithm

```
procedure determineValue
    in dataSet : ValueSet
    out faultySet, validSet : ValueSet →
        local valFreqSet : ValueWithFrequencySet
        ∀ elements in dataSet →
                find cur_elem in valFreqSet
                if found increment frequency
                else insert cur_elem into valFreqSet
        end ∀
        local valid_value := extractMaxFreqValue from valFreqSet
        validSet := extract from dataSet elementsOfValue valid_value
        faultySet := $validSet \cup dataSet \setminus validSet \cap dataSet$
end proc
```

as input to all the replicas involved. The (deterministic) algorithm is used to select the valid value and this is not changed later. Similarly the **validity condition** is met, as the algorithm uses only the data sent by the transmitters. As the consensus algorithm does not loop infinitely, but only for a finite size, it does terminate. Thus the **termination condition** is satisfied.

As the presented system allows multicasting messages to all the replicas in the group, and messages are ordered, the work of Schneider [18] and Dolev *et al* [12] show that these conditions mean deterministic consensus can be achieved and the system can tolerate up to $k-1$ failures, where k represents the maximum number of nodes that we can multicast to. At system startup $k=t$.

4.5 Dealing With Faults

When the discriminating node identifies a particular replica as having failed, it must take some action. The simplest action would be to do nothing. This produces the result that the number of non-faulty replicas in the set from which the faulty replica has been removed has decreased. The total number of failures that the system can withstand has subsequently decreased by 1 as well. So for a system to be tolerant of t faults, all the replica groups must start with at least t process replicas.

Another option available to the system is to replace the faulty replica with a newly started replica. This replica would need to know the state of the system and the state of the replica it is replacing. This approach is discussed in section section 6.1.

Whatever approach is chosen, *dynamic reconfiguration* of the system is required when a node is identified as having failed. ftdl currently does not support dynamic replica replacement strategies, but is does remove faulty replicas from a replica set.

Dynamic reconfiguration requires that the network of replicas adjust to the update. Clients and other users of the replica group must also adjust their state.

Fortunately, as ftdl uses a multicast protocol, not very much needs to be done when a fault occurs. The identified faulty replica is removed from the *Valid Replica* set of the CTXMGR to its *Faulty Replica* set, as shown in Figure 4.4. This ensures that any subsequent communications from replicas posing as replica in the *Faulty Replica* set are ignored.

One of the advantages of removing faulty replicas from the communication group is a potential increase in the overall throughput of the system. As the number of replicas involved in the communication aspect decreases, the consensus algorithm (and the number of messages sent on the network) may lead to better performance as some agreement algorithms are proportional to the total number of replicas [18]. ftdl's agreement protocol clearly is dependent on the number of non-faulty replicas in the group.

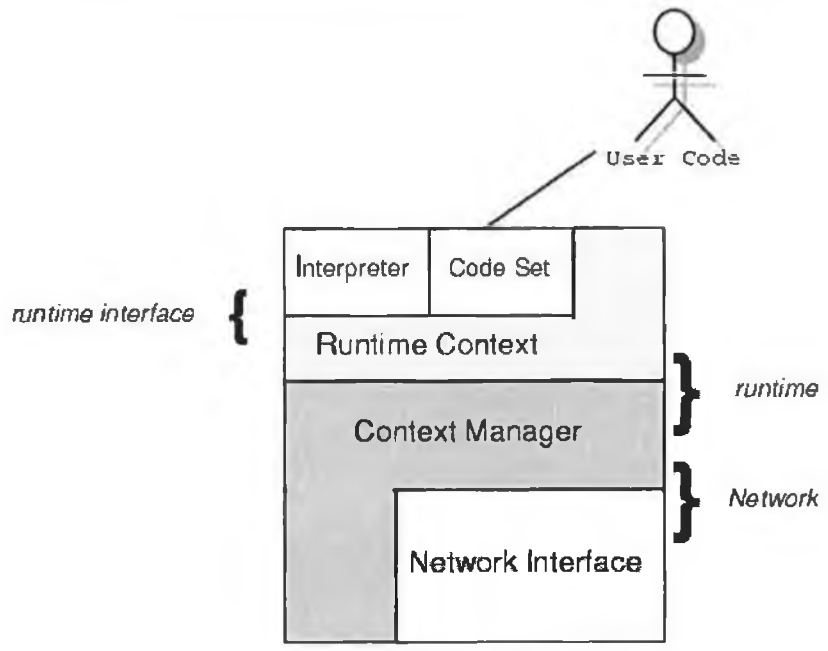
This, however, is a limited advantage. As the number of faults increase, the overall reliability of the system decreases. As it stops being a t-tolerant system, it becomes a $< t$ tolerant system. Having replica replacement could help in keeping the system tolerant to t faults.

The runtime realizes a fault has occurred by examining the *faultySet* returned from the fault detector, *determineValue*, algorithm. If the returned *validSet* is empty, then the system has failed and execution is halted. This typically occurs when the Combining Condition [18] is violated.

4.5.1 Summary of Supported Features

In summary, the language and distributed executive support synchronous messaging, fail stop processes, network faults and limited Byzantine failures.

Figure 4.9 ftdl Distributed Architecture



A modified version of the diagram presented in Section 3.7 is presented here, showing how the ftdl environment can be mapped to that architecture. This is shown in Figure 4.9.

The ftdl environment extends the capabilities of the SPROC approaches as used by Schneider [18] and FT-Linda[5]. Much of the node management and communication is based on Laport [14] and Schneider [18]. However, the major differentiating factor is the replication model. Both Schneider [18] and Lamport [14] expect all the nodes in the system to hold replicas for each process. Lamport [14] does allow his model to run replicas on a subset of the nodes, but this is does not change the general approach. These tend to provide fault tolerance easily for applications that work on a shared-memory principal, like the Linda extension [5]. All the nodes in the system contain their own state, this gets altered based on messages received from the network. An implied uniformity exists between clients and servers. Thus it is alright for a client and server to be processing on the same node. This immediately negates the usefulness of a distributed system. If the client and server are on the same host, then the system is not distributed!

ftdl supports an independent model. Each replica is an *independent* entity. Clients must explicitly obtain references to (remote) servers for invocations. The invocation parameters are propagated to the correct replicas, but each must process it differently and then dispatch the results. The replication and group communication is an extension to the distributed model and we use those entities to provide the fault tolerance. This allows replication of potentially computation intensive processes to be done across different machines. Our model, however, introduces a significant overhead in communication times.

4.6 The Simulation Machine

A runtime interpreter has been written which can execute the instructions generated for the simulation machine. This runtime interpreter has been integrated into a full simulator which emulates a number of process nodes, the runtime environment and network connecting these nodes.

The implemented network that links these nodes is very primitive. It does not suffer delay, delivery times are instantaneous and does not fail. However, we can represent network failure, or delay by causing particular replicas to fail (or delay). A full model for a network is beyond the scope of the current study.

This simulator models the interactions between interacting processes on different nodes. This allows us to study the effects of fault tolerance on the processes and ultimately, the effect or reliability of the system.

Each protocol implementation is represented as a process on its own host (node). For the sake of this model, a process represents a unique node.

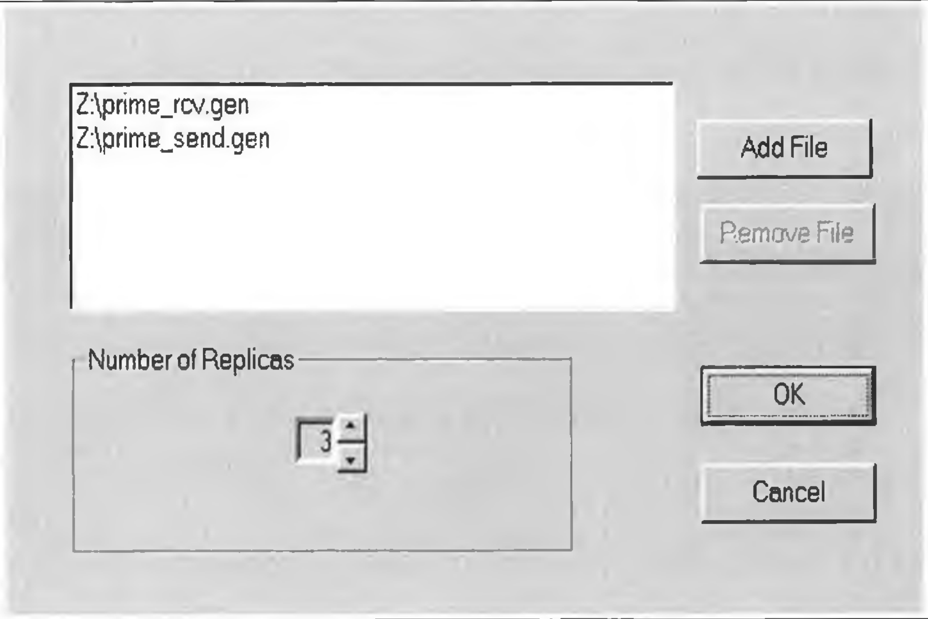
Tolerance to failures are established by using replicas. These replicas execute the same code as their originators, they receive the same communication messages, and collaborate to determine which replicas are faulty. Each replica represents a process executing at a separate node.

4.6.1 User Interface

Initialising The Environment

The user specifies the GEN files containing the protocol implementations it wishes to execute. It must also specify the number of replicas per process to be used. The greater the number of replicas, the system becomes more tolerant to failures.

Figure 4.10 Environment Initialisation

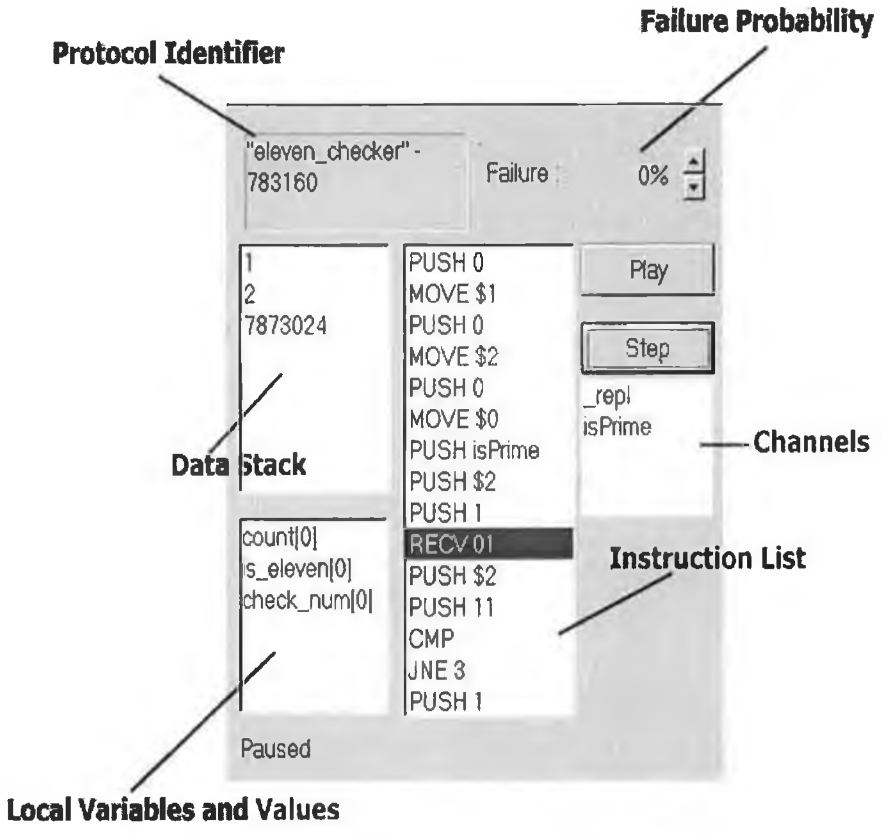

Z:\prime_rcv.gen
Z:\prime_send.gen

Number of Replicas: 3

Once these parameters have been specified, the system is initialised and execution can begin. Figure 4.10 shows the initialisation screen.

Each replica cell, shown in Figure 4.11 , contains the runtime information for that process. This area is divided into eight subsections and two control points.

Figure 4.11 Replica Cell



"eleven_checker" - 783160

```
1
2
7873024
```

```
count|0|
is_eleven|0|
check_num|0|
```

```
PUSH 0
MOVE $1
PUSH 0
MOVE $2
PUSH 0
MOVE $0
PUSH isPrime
PUSH $2
PUSH 1
RECV 01
PUSH $2
PUSH 11
CMP
JNE 3
PUSH 1
```

```
_repl
isPrime
```

The information subsections convey:

- the replica group identifier of the process, as defined by the Init statement;
- the probability of failure. This percentage can be dynamically set while the system is running;
- the channel list. These are all the channels declared for the protocols;
- the wait channel list. This indicates which channels are expecting data. It is situated beneath the channel list;

- a status area. This indicates the execution state of the replica.
- data stack;
- local variable status area – displaying the names values of the local variables
- the instruction list

  Figure 4.11 shows a replica about to execute a RCV instruction.

  Simulator Screen

  The main simulation screen consists of a window divided into cells. Each cell represents a replica process running on its own node. Each row of cells represents one replica group. Figure 4.12 shows the screen for two replica groups, with three replicas in each group. The bottom row of replicas – the “ElevensesWaiter” process group – are indicating that they are about to send data. The message displayes is informational, and not a status.

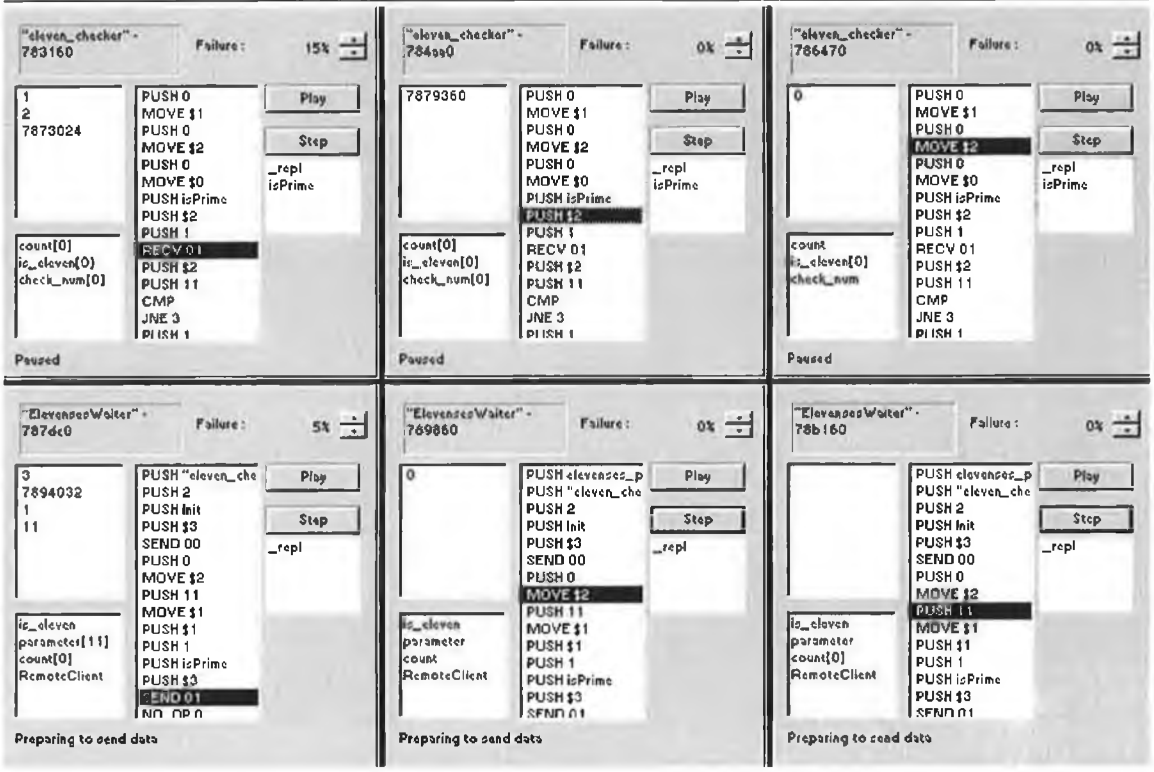
  A replica can be “paused”, “executing”, “waiting” or “faulty”. The first state means that the instruction pointer is advanced through user intervention – by use of the *step* button – as opposed to being internally driven.

  The *executing* state means that a statement is currently being executed. The *waiting* state means that the replica is awaiting a reply from a remote invocation. The channel it is waiting for is listed in the wait channel list.

  The last state, *faulty*, is for information purposes only. When a replica is identified as faulty, the status field displays a message that it is; all execution stops for the replica.

- the instruction list shows all the instructions, in the form as generated by the compiler, for the replica.

Figure 4.12 Main Simulator Screen



```
"eleven_checker" - 783160        Failure : 15%
1
2
7873024
PUSH 0
MOVE $1
PUSH 0
MOVE $2
PUSH 0
MOVE $0
PUSH isPrime
PUSH $2
PUSH 1
RECV 01
PUSH $2
PUSH 11
CMP
JNE 3
PUSH 1
_repl
isPrime
count[0]
is_eleven(0)
check_num[0]
Paused
```

```
"eleven_checker" - 784aa0        Failure : 0%
7879360
PUSH 0
MOVE $1
PUSH 0
MOVE $2
PUSH 0
MOVE $0
PUSH isPrime
PUSH $2
PUSH 1
RECV 01
PUSH $2
PUSH 11
CMP
JNE 3
PUSH 1
_repl
isPrime
count[0]
is_eleven[0]
check_num[0]
Paused
```

```
"eleven_checker" - 786470        Failure : 0%
0
PUSH 0
MOVE $1
PUSH 0
MOVE $2
PUSH 0
MOVE $0
PUSH isPrime
PUSH $2
PUSH 1
RECV 01
PUSH $2
PUSH 11
CMP
JNE 3
PUSH 1
_repl
isPrime
count
is_eleven[0]
check_num
Paused
```

```
"ElevensesWaiter" - 787dc0        Failure : 5%
3
7894032
1
11
PUSH "eleven_che
PUSH 2
PUSH Init
PUSH $3
SEND 00
PUSH 0
MOVE $2
PUSH 11
MOVE $1
PUSH $1
PUSH 1
PUSH isPrime
PUSH $3
SEND 01
NO_OP 0
_repl
is_eleven
parameter[11]
count[0]
RemoteClient
Preparing to send data
```

```
"ElevensesWaiter" - 769860        Failure : 0%
0
PUSH elevenses_p
PUSH "eleven_che
PUSH 2
PUSH Init
PUSH $3
SEND 00
PUSH 0
MOVE $2
PUSH 11
MOVE $1
PUSH $1
PUSH 1
PUSH isPrime
PUSH $3
SEND 01
_repl
is_eleven
parameter
count
RemoteClient
Preparing to send data
```

```
"ElevensesWaiter" - 78b160        Failure : 0%
PUSH elevenses_p
PUSH "eleven_che
PUSH 2
PUSH Init
PUSH $3
SEND 00
PUSH 0
MOVE $2
PUSH 11
MOVE $1
PUSH $1
PUSH 1
PUSH isPrime
PUSH $3
SEND 01
_repl
is_eleven
parameter
count[0]
RemoteClient
Preparing to send data
```

- the data stack. The user can see as values are added and removed from the stack. In figure 4.11 the values on the stack represent the number of arguments this operation requires, in this case 1; the local variable to receive the value, in this case the variable CHECK_NUM (variable counting begins at 0). The last number on the stack is the reference to the channel that is to receive the data.

Execution can be controlled on a cell-by-cell bases or applied to all the replica cells. Each cell has two control button – a **play/pause** and **step** button. The **play** button means that the replica is self driven. It executes instructions on its own, no user intervention is required. However, if the replica is in the "paused" state, then user intervention is required to advance the instruction pointer. This is done through the **step** button. The replicas start in the "paused" state.

Every cell has these buttons, but there are global buttons, contained on the frame of the window, which operate on all the replicas when pressed.These buttons provide the play, pause and step functionality. Pressing any of these applies the action to all the replica cells. So pressing the pause global button will move all the replicas to the paused state..

By setting the failure probabilities, we can not only model individual node failures, also network behaviour and reliability. By setting all the replica's failure probability to the same value we can show how the system copes with a busy network, for example.

4.6.2 Measuring Effectiveness

In order to measure the effectiveness of the model, a log file is generated by the simulator. This file contains timestamps of when each operation takes place. Every action taken by the system and by each virtual processor is written to the log file as a timestamped event. This data is used when presenting the results in the next section.

Chapter 5

Results

5.1 Introduction

The data presented here aims to show the cost of failure in a simulated fault-tolerant system. The following questions will be answered in the course of this section:

- How does the system deal with errors? This is presented in section 5.3.1.
- What is the degradation as errors occur? This is presented on section 5.3.2.
- What is the overhead of the system in the presence of no errors? section 5.3.3.

For the system to provide fault tolerance, each process in the system must be *replicated*. Whenever inter-process communication occurs, through **SEND** and **RCV** directives, all replicas must take part. Messages are synchronous and all **SEND**'s receive replies.

The recipient of a message waits for all replicas to respond. However, it does not wait indefinitely. When a timeout occurs, all replicas that have sent messages are used in the processing; all those replicas which did not send in the timeout are marked as faulty.

The system is tolerant for $n - 2$ faults, where n is the number of replicas per process.

5.2 The Test Case

The test consists of two processes – a client and a server. The client issues 25 requests to the server. Each request requires a response from the server. When the server receives the request, is processes it, computes the result and issues a rely to the client with that vale. Once this response is received, the client can continue processing.

The test case was run for various numbers of replicas and at different replica failure probabilities. Three, five and ten replicas were used in the test; 0%, 5%, 15%, 20%, 45% and 65% were the failure probabilities against which all the replicas were run. Each combination was run 5 times.

The data presented does not measure network communication time. As this is its own varied and deep research area, attempting to simulate a network is beyond the scope of this study. Each network has its own characteristics of latency and message arrival, this would have added another variable to digest. However it is *vital* for such costs to be factored in as this is an area of utmost importance as the number of inter-communicating replicas increases. The network presented here is, unfortunately, artificial. The simulation lived in a perfect environment, where data is received the exact moment it is sent!

The current implementation does not support replica replacement, whereby a faulty replica is replaced by a newly started one. This ability would have significant impact on the duration of the system as well as its performance. As faulty replicas would be replaced by new replicas, a steady state could be introduced where a certain level of replicas would always be present in the system preventing the system from failing completely.

Performance, however, would be compromised as the negotiation required among the non-faulty replicas to start up a new replica would be quite significant. In fact some of the non-faulty replicas may become faulty during this procedure. Computation effort would be expended not on the required problem (the user code), but on system maintenance. This is part of the balance that must be achieved between the conflicting aims of availability, responsiveness and performance.

Presented below, in Figure 5.1 is the test case for the first run of the system. The client is the process which implements the protocol NUM_SENDER the server process is the implementations the protocol ELEVENSES_PROTOCOL

5.3 The Results

5.3.1 Dealing With Errors

When a node identifies another as having failed, it must handle this fault. The system marks the failed node as faulty, removes it from its acceptable replica set and then continues processing, if possible.

When the number of nodes in the replica set becomes less than two and the other replicas have sent their messages for this invocation, or have timed out and the data from both of the remaining nodes differ, the current node cannot continue processing and thus will terminate.

Thus a node will terminate if it completes normal processing or $n-2$ failures have occurred.

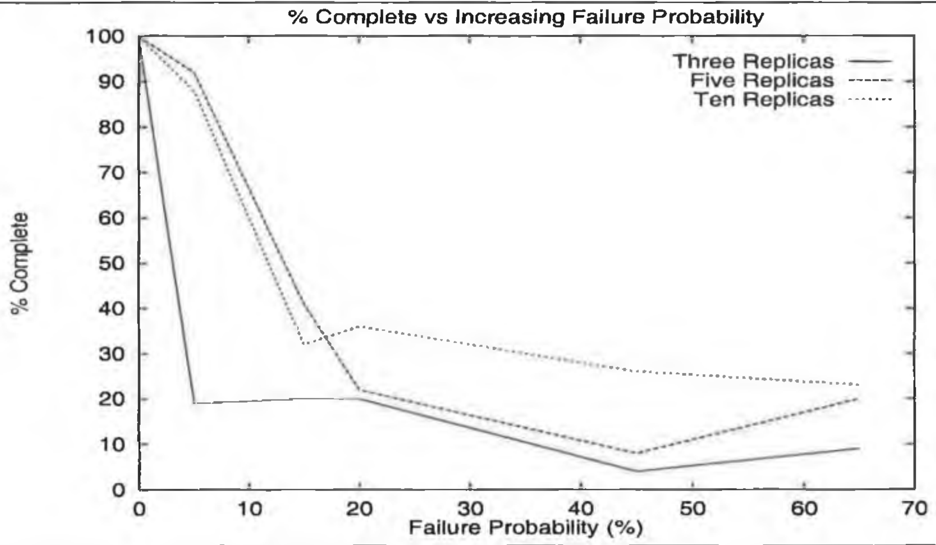
Referring to Figure 5.2, we can see a relationship exists between the number of replicas, the failure probability level and the amount of system execution successfully completed. Table 5.1 shows that system did not run error free at any failure level. However at certain points the percentage completion was very high. It is interesting to note, that even without active replica activation, an optimum relationship could be attained with the number of replicas at a given failure level.

For a given probability of failure, there exists an optimum number of replicas for which the system completion reaches a maximum. Table 5.1 shows that for a 5% probability of failure, the optimum number of replicas is 5. Having 3 replicas only produces 19% system completion, whereas 10 replicas produces

Figure 5.1 Eleven Test

```
– Server communication protocol
–
Elevenses_protocol →
        isPrime( given_num : Integer ) : Integer
end elevenses_protocol


– Client Code
–
define num_sender →
      RemoteClient : elevenses_protocol;
      count : Integer;
      parameter : Integer;
      is_eleven : Integer

      RemoteClient.Init( “eleven_checker” );
      ID( “ElevensesWaiter” );

      count :=0;
      parameter := 11;

      REPEAT →
              is_eleven := RemoteClient.isPrime( parameter );
              count := count +1
      UNTIL count > 25
end num_sender

– Server Implementation
–
define elevenses_protocol →
      check_num : Integer;
      is_eleven : Integer;
      count : Integer
      –
      – initialisation
      –
      ID( “eleven_checker” );
      is_eleven := 0;
      check_num := 0;
      count := 0;
      REPEAT →
              – wait for a request
              –
              RCV isPrime( check_num );
              IF check_num = 11 THEN
                 is_eleven := 1
              end;
              RETURN isPrime is_eleven;
              count := count +1
      UNTIL count > 25
end elevenses_protocol
```

Figure 5.2 Completion vs. Probability of Failure



88%, both of which are less than the 92% completion exhibited by the 5 replica case. Thus having too few replicas as well as too many can have a detrimental effect.

This situation, where the 5 replica scenario dominates all others is observed for the 5% and 15% failure cases. Referring again to Figure 5.2, the 10 replica line crosses the 5 replica line at about 18% failure probability. This is the point where the effectiveness of 5 replicas has diminished. At the 20% failure level, 10 replicas complete 36% of the system, whereas 5 replicas complete 22%. Though this gap is not too big, it becomes very evident at the 45% failure level where 10 replicas complete 26% of execution, but the 5 replicas only 8%. There seems to be a minimal advantage of increasing the number of replicas.

Failure can occur as a result of a replica becoming faulty or a message being faulty. With more replicas, there are more messages transmitted per invocation. Each of these is subject to failure. In order to determine the optimum number of replicas, the number of replicas plays an important role. Not only does it provide more resilience as the failures increase, but it has a negative impact on the total number of messages transmitted. At low probabilities of failure, the probability of the replica failing is low but the probability of a message failure depends on the total number of replicas. If a process group consists of 10 replicas and there are 2 process groups, the total number of messages involved in a send-reply scenario is 20 messages per invocation. For the test presented in Figure 5.1 one invocation occurs per iteration of the main loop, there are 25 iterations in total. This gives 500 message transfers, whereas 3 replicas per process group yields 150 messages. The system is being swamped with the number of messages compared to the number of replicas when there are 10 replicas per process group at low probabilities of failure.

FAILURE PROBABILITY	NUMBER OF REPLICAS	AVERAGE RUN TIME	% COMPLETE
5%	3	32723	19%
	5	28679	92%
	10	35547	88%
15%	3	34724	20%
	5	34251	41%
	10	44795	32%
20%	3	41201	20%
	5	34787	22%
	10	37741	36%
45%	3	43658	4%
	5	59203	8%
	10	46141	26%
65%	3	40717	9%
	5	53903	20%
	10	39574	23%

Table 5.1: Replica Count, Failure Probability and Average Invocation Time

It is interesting to note that the system also exhibits a convergence of system completion for a given replica count and failure probability. This can be used to measure the effectiveness of the number of replicas. At the 65% level, for example, the difference in completed system execution between 10 replicas and 5 is only 3%.

At the 20% failure level, 3 replicas complete 20% of the system, where as 5 complete 22%. This is only a difference of 2%.

Using this observation, we can see that there exists a threshold level for the number of replicas at a given failure level. If the number of replicas is within this threshold, the percentage completion does not vary greatly between the actual replica counts. We can see this window in the Figure 5.2, where the distance between the replica lines is very small. Thus from the 20% to 48% failure level, this threshold would be 5 replicas. Having between 3 and 5 replicas would not greatly effect the amount of the system actually executed.

This window then changes at the 55% to 65% failure range, where the threshold is 5-10 replicas.

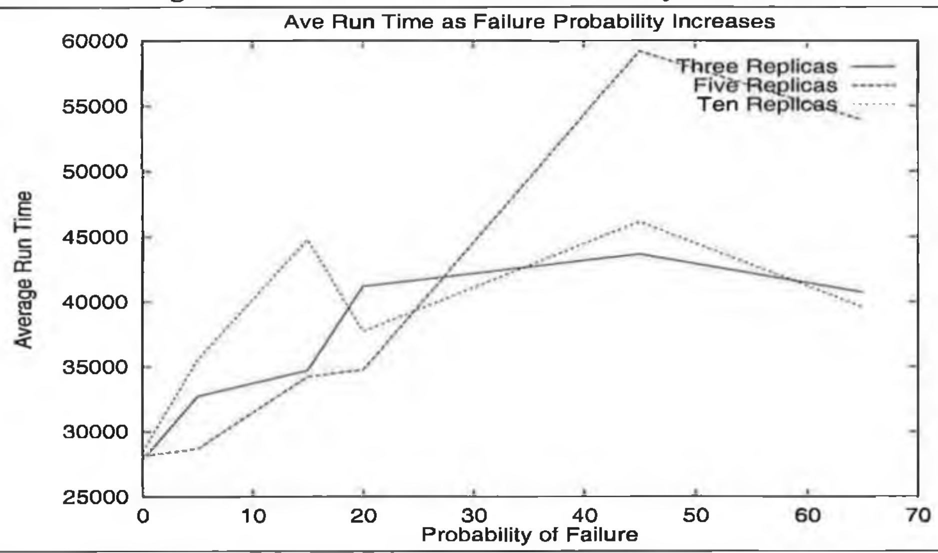
5.3.2 The Effect of Errors and Replicas - Performance Degradation of the System

Two main aspects of the system degrade as the number of failures increase. The general availability of the system sharply declines as the failure probability increases, regardless of the number of replicas. As the failure probability rises from 5% to 65%, percentage of the number of runs where the system actually completed falls sharply. At higher failure levels (>20%), the system halts at about the same execution point regardless of the number of replicas. This is because at these high level, the number of replicas failing per invocation is very high. For example, at 45% failure with 10 replicas, in the 5 test runs of the system, the number of replicas failing between the first and second invocations was about 4. It is doubtful that having active replica activation available would increase the survivability of the system, as these highly fault prone replicas would have to communicate to reach a consensus that new replicas must be started. Figure 5.2 clearly illustrates this.

The other aspect that leads to system degradation is that, as the probability of failure increases, the average invocation time per operation also increases. Referring to Figure 5.3, as the number of failures rise (indicated by the increase in failure probability), the average invocation time increases sharply.

The replicas exhibit a similar invocation time profile. A slight rise occurs, then a sharp dip, followed by gradually increasing phase and then a sharp dip. Table 5.1 shows the average invocation time figures. These are very much influenced by the number of replicas, failure distribution and system completion.

Figure 5.3 Average Invocation Time vs. Probability of Failure



The graph shows that as the number of failures increases, the average - invocation time also increases. This is part of the cost of failure in the system. As each non-faulty node tries to discover which nodes are faulty the overall performance degrades.

This can lead to the conclusion that the effect of failure on the system has the same impact on performance as having lots of replicas. That is to say, the performance impact of replicas, in a zero failure environment, is equivalent to the system recovering from failures. So the cost of a failure can be expressed in terms of having extra replicas – or vice versa. This topic will be further explored in the next section, section 5.3.3.

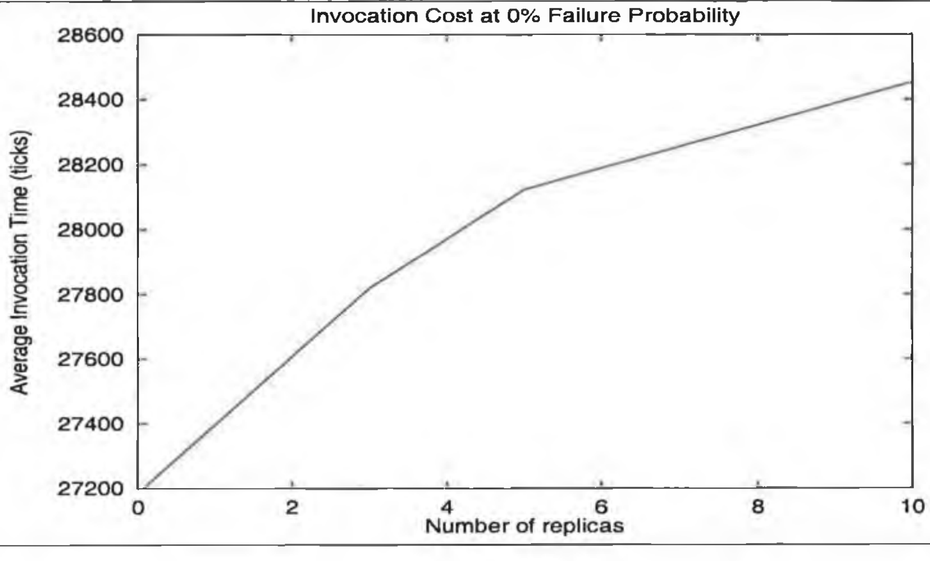
The dips can be explained as massive failure points. When a large number of replicas fail around the same iteration, the system has less "baggage" for subsequent communications. As there is no facility to start new replicas, the number of replicas that survive after the first few initial iterations is significant. Consider the 20% failure probability case with 10 replicas. Here, the number of replicas failing before the fifth iteration is 5, on average. That is, over half of the original replicas are left on the sixth iteration. Thus the system would perform like a 5 replica case.

Generally, the time taken to execute the system has an inverse relation to the percentage of the system completed. This relationship is influenced by the number of faults; if the system halts due to too many failures, the time taken to execute this incomplete run is greater than the time taken to do a full run. This is because of the failure identification algorithm and removal of faulty replicas from a non-faulty replica's communication set. For example, when 3 replicas have only completed 4% of execution, it takes 43658 ticks; where as 20% completion takes 34724 ticks. These numbers are similar for other probabilities and failure levels. Table 5.1 shows this.

Another factor in this phenomenon, is that as more faults occur, there are fewer surviving replicas on a system wide basis. Thus the communication set at each replica is smaller with each failure and, when subsequent interactions occur, the data that must be validated forms a smaller set.

5.3.3 Replica Overhead

Figure 5.4 Replica Cost



To measure the cost of system overhead, the test was run at a zero level of probability failure level for all the replica scenarios. Table 5.2 below presents these results. The system was also run with one replica. This helps give a baseline figure.

The replica overhead is measured as the total execution time of the system divided by system execution time for the one replica case. Also shown in the table is the average invocation time. This measures the amount of time taken when a replica receives all of its messages, inspects them for faults and then finally dispatches them to the application code. The graph, shown in Figure 5.4, helps to illustrate this.

Number of Replicas	System Execution Time (ticks)	Replica Overhead	Average Time Per Invocation
1	679640	n/a	27185
2	702770	3%	28110
3	718320	5%	28732
10	716390	5%	28655

Table 5.2: Replica Cost at 0% Failure Probability

It is interesting to note that a maximum appears to be reached for the overhead and the average invocation time, relative to the no-cost option. As these figures do not include network communication times, this would factor in greatly as the number of replicas increases. However as these figures are not present, we can see the system overhead.

5.4 Conclusions

These results have shown that while the cost of having replicas is between 3% and 5% the increase in survivability of the system can be worth it. But these results also indicate that research needs to be done in measuring the effectiveness of having replica activation incorporated into the model. Perhaps with this feature, the upper limit of the number of replicas for a given failure probability can be removed. Section 5.3.1 shows that there is an ideal number of replicas for a particular failure level. If this number is not met then the overall usefulness of the extra replicas is reduced.

Chapter 6

Conclusions

The central aim of developing the ftdl environment was to provide an integrated approach to enable the construction of fault tolerant systems. By providing language and runtime support, ftdl is a system that allows *transparent* fault-tolerance. Through the use of collaborating replicas acting as a group, interacting with other replica groups, fault-tolerance is provided. The distributed model used is based on communicating processes; the fault-tolerance mechanisms are based on a hybrid approach of the mechanism used by distributed shared memory and communicating process systems.

A language and compiler were developed to provide the user to develop fault tolerant applications. This language is lightweight but was developed to allow the system to obtain compile time information from the user code. This is a very important requirement as the system needs to know the semantic nature of remote invocations and the manner in which parameters are used and their values obtained. Using a custom language has provided some considerable advantages. It allows us to identify the features required in a target language and compiler that are required for building fault tolerant systems. Though ftdl is lightweight it provides a minimum set of features required for building distributed programs. Branching, looping, remote operation invocation, receiption of operations from clients and the use of local variables are all included. By scoping the language we were left to focus on the support required from the compiler. For example, for modules (in the case of ftdl these would be protocol declarations), we need to be able to identify if an operation is deterministic or if the output from it is stochastic. A major feature missing from the language is the lack of support for non-determinsism. This was deemed beyond the scope of the current work but is an area which requires further investigation as it is very important. If we had chosen to use an existing language, the implementation would have been forced to put with limitations of that host language – for example trying to extend C++, a non-distributed language, would have required a work around to add in distributed concepts. A lot of information is required as a result of the compilation phase.

The ftdl language can be replaced by another, but not without some loss in the amount of fault tolerance that could be required. If the ftdl system was to be integrated into an existing language, the important requirement would be to have a compiler for the replacement language. If ftdl was replaced, the compiler for the replacement language would need to gather the semantic information required by the runtime environment.

The ftdl system can be integrated into existing languages and environments. Doing this, however, would require compromises to be made. One of the major compromises would the loss of language support. Ideally an integration with another language should require such an integration to be non-intrusive. The steps required for fault-tolerance at the language level should be minimal. For example, if every remote invocation would need to be preceded by some statement this would not be a good integration. The compiler and integration layer should notice that the remote invocation is taking place.

This thesis has shown what features are required to implement a distributed system that tolerates failure. It is important when implementing such a system that the correct parameters are used to ensure that the overhead associated with fault tolerance do not detract from the actual application.

When determining the number of replicas required it is important to look at the results presented in Chapter 5. By increasing the number of replicas in a system, the number of failure points has increased. Not only can the original user processes fail, but so can the replica processes added by the ftdlsystem.

The number of messages sent per invocation also increases as the number of replicas increase. So the possibility of a message failing also increases.

However, having a small number of replicas does decrease the number of failure points, but the number of failures which can be tolerated has also decreased, resulting in a system that is less likely to withstand failures. When selecting the number of replicas, we want to avoid the system causing failures, by adding more failure points.

The survivability of a distributed system is not directly linked to the number of replicas in a process group. That is, increasing the number of replicas in a replica group *may* not increase the total survivability. The effectiveness of the number of replicas depends on the probability of failure; at low probabilities, a system can have too many replicas thereby swamping the number of replicas. Determining the optimum number of replicas to achieve maximum survivability is dependant on the failure probability of the system components coupled with the desired number of replicas.

6.1 Future Extensions

A major area that requires investigation is *dynamic replica activation*. This feature allows new replicas to be integrated into a running system. Typically, the new replicas will replace those identified as faulty. In order to implement this, a full checkpointing facility also needs to be developed.

A number of issues exist when a new replica is to be integrated into the system. A protocol needs to be defined which allows all replicas in a replica group to agree to start a new replica, agree on the host node, transfer the current state and integrate the new replica into the system. While this is happening, normal operation would need to be suspended. This means that user code will not be executed when updating is taking place.

This scheme, with the barrier checkpoint proposed below could help integrate slow replicas which were erroneously identified as faulty to be updated in state and re-join the system. When a previously identified faulty replica attempts to send a message to a receiving replica, it needs to check that:

- this *is* the replica it says it is. Authentication is required, perhaps through digital signatures [14].
- if the replica is identified as a previously faulty one, it was marked as faulty because it did not send a message by the appropriate time. That is, it was *slow* and not faulty. The CTXMGR will need to check this by referring to the request identifier and then using the update algorithm, in co-ordination with the replica group it was a part of, to get the current state.

In order to support state transfers, a form of *checkpointing* is required by executing replicas. All the messages sent to a replica group need to be stored in order to replay them to a new replica so it can have the same state as the others in its group.

It is prohibitive to store *all* the outgoing messages to a protocol group. This can be very time and space intensive and the application may not allow all the messages to be stored. A long-living server, for example, will have many messages over the course of its life.

A solution to this is to periodically store the state of a replica group and then store the delta messages after the storage. If any state updates need to occur between state storage points, the destination replica is given the last saved state and all the subsequent messages.

This can be supported by allowing a *checkpointing* facility as outlined by Detlefs *et al* [11]. Let us assume that the language, ftdl, would be extended to allow checkpointing to occur. This can be achieved through a new language statement, or a runtime initialisation method (e.g. checkpoint every 10 seconds) or a combination of the two.

The aim is to store the current state of a replica. This would be used at a later stage as input so another replica could update, or set, its state to the input value. When the runtime encounters the checkpoint event, all the replicas in the group agree on the state value before archiving locally. This archive value, plus message playback is used to update a replica, when requested.

In this sense the checkpoint acts like a barrier, which is akin to the rendezvous approach supported by Ada [7] and explained in *Principles if Concurrent and Distributed Programming* [8]. The replicas execute independently until the co-ordination point (checkpoint) is reached. This can be used to ensure continual system integrity, as the replicas in a group synchronise their state and remove faulty processes outside the normal client-server communication chain.

The desire for a new replica to be inserted into a group is used to maintain the Combining Condition [18]. For example, a new replica could be added to a group which has incurred failure. This should increase the failure tolerance of the system. Another point when replica state transfer could be used is the updating of a slow replica. The mechanism for re-integrating slow replicas and introducing new replicas to a replica group is quite similar.

When a new replica is to be added to a replica group, the replica is started on a node and enters the initialisation state. This will *broadcast* that this is a new replica and it wishes to be part of the given replica group. Broadcast ensures that *all* the processes in the system see the message, and thus they know who the message is coming from. When using broadcast, authentication schemes are not required [24, 14]. All the existing elements of the desired replica group will then broadcast their state to the new replica. The new replica will use this data to set its state to that of all its peers (this is done by using the same agreement protocol for normal communications). The new replica will now broadcast that it *is* the new replica in the process group. All the processes (replicas) in the system can now update their replica tables. Using broadcast ensures that all replicas get the notification and Byzantine failures are avoided as the other nodes can know who is sending the message. Re-integration of old (slow) replicas is similar to the steps described above, however the main difference is how it is initiated.

Re-integration is initiated when a slow replica, P_i, replies to a message *after* the expected time of arrival, T_{arrive}. This reply is sent to the replica group D. All non-faulty replicas will receive this late-reply and notice that it is slow. At this point, all the replicas of D will multicast to the non-faulty replicas of P and to P_i, that P_i is a slow replica needing to be updated. The rest of the protocol is similar to that described previously for integrating a new replica.

The channel cache of all the replicas is used as input to the new/updating replicas for messages sent after the last checkpoint. As a proposed alternative, checkpointing can be made to occur when a slow, or new, replica needs to be (re-)integrated into the system. This can reduce the storage requirements for all replicas.

One of the primary disadvantages of implementing a dynamic replica integration scheme, as described above, is an increased failure rate. As there is significant processing and inter-communication required to integrate the replica, failure could occur during this phase. The algorithms involved are tolerant of failures, but the system would be less stable and processing resources effectively wasted doing non-application required work.

Other areas which could extended would be the addition of support for asynchronous messaging and notifying user code when failures have occurred.

Though these extensions are required for a fully functional fault tolerant system, the model implemented by ftdl provides the basic framework while supporting fault tolerance. This framework is the minimum required for a usable system and the results in Chapter 5 reflect this.

6.2 Conclusion

The ftdl model is an example of a hybrid approach derived from work done with distributed shared memory systems and state machine approaches to providing fault tolerance. One of the main differences in its implementation is the replica management; previous systems identified all processes (and data) at all nodes, or a common subset. ftdl groups replicas of the same protocol implementation and treats these as separate communicating entities. This provides a platform for developing systems that are fault tolerant, without excessive runtime communication. The ftdl system can be integrated into existing languages through library support, but at the cost of ease of use and transparency.

This thesis has shown that it is possible to construct a fault tolerant system that provides transparent fault tolerance and that fault tolerance semantics can be built into a language. This is achieved by adopting a hybrid approach for the distributed process model. The overall cost of replicas has been shown to small and non-linear with respect to the number of replicas and system execution rate. Most importantly, the model can deal with failures and has shown that even without faulty replica replacement an effective fault tolerant system can be constructed.

Appendix A

ftdl Language Description and Compilation Environment

This appendix presents the ftdllanguage's lexical conventions, EBNF grammar and a description of the compilation environment. By understanding these steps, it should help in the reading of Chapters 4 and 5.

The EBNF notation used can be read as follows. Terms in quotation marks are the exact sequence of symbols expected, emphasised terms, as in *id_statement* are non-terminals these terms are defined separately. A non-terminal is defined by the name of the non-terminal followed by ::=. Terms enclosed in square brackets indicate that the term is optional. Curly braces indicate zero or more repetitions of that term. A vertical bar ("|") separating terms indicates one of the terms, on either side of the bar, can be used.

For example:

`impl_statement` ::= *`assignment`* | *`id_statement`*

is read as the non-terminal `impl_statement` is made up of an assignment or an id_statement. The id_statement rule is defined as:

`id_statement` ::= "ID" "(" *string literal* ")"

An example of this is: `ID(''SomeName'')`

bool	char	define	else	elsif	end
ID	if	int	integer	proc	rcv
real	repeat	return	string	then	until

Table A.1: Reserved Words in ftdl

A.1 The ftdl Language

ftdl is a simple language which provides the minimum constructs necessary in constructing a viable distributed application. This chapter describes the syntax and should be read as a supplement to the presentation of the ftdl programming model in Chapter 4. The implementation of the compiler was undertaken using the compiler tools lex and yacc [16] which helped to implement the lexical analyzer and parser respectively.

The compiler takes as input a text file which consists of one or more protocol declarations followed by a protocol implementation. The compilation phase is divided into, lexical analysis, parsing and code generation. The lexical analyzer goes through the source file and partitions sequences of characters into tokens which are given to the parser. The parser assembles the tokens, using the language grammar into an abstract syntax tree to be used by the code generator. The generator traverses the tree to output the GEN to be used by the simulator machine.

A.1.1 Lexical Analysis

ftdl is not a case sensitive language, so tokens must differ by more than case. The reserved words of the language are shown in Table A.1. These cannot be used as variable names or protocol names by the user. Comments begin with "−−" (two consecutive dashes) and last until the end of the line. These are ignored by the compiler during the parsing phase.

In addition to the reserved words, the lexical phase identifies the entities shown in Table A.2 for the parser. The *number* entity is used in arithmetic expressions, boolean expressions, assignment statements and remote invocations. The *string literal* is also used in the same contexts as a number in addition to being used in Init and ID operations. An *identifier* is used to name user definitions: variable names, protocol names and the operations of a protocol.

Lexical Entity	**Definition**	**Example**
Number	[0-9]*.[0-9]+	101
String Literal	"[a-zA-Z][a-zA-Z_]*digit*"	"Hello"
Identifier	[a-zA-Z][a-zA-Z_0-9]*[0-9]*	aVariable3

Table A.2: Lexical Definitions

A.1.2 ftdl Grammar

An ftdl source file consists of one or more protocol declarations. Followed by a protocol definition. This definition must be of a protocol previously declared.

Protocol Declaration

A protocol declaration describes which operations can be invoked on a protocol. The EBNF grammar for a protocol declaration is shown in Figure A.1. It consists of a unique name for the protocol followed by an optional list of operations and the declaration is completed by the keyword "end" followed by the protocol name. Once a protocol has been declared it is considered a type by the language. This means it can be used as any of the built in types, for example it can be used in a variable declaration.

The minimum declaration of a protocol is:

```
MinimumProtocol ->
end MinimumProtocol
```

This protocol is quite useless in that no operations can be invoked on it. But it is useful when you want to write a pure client program.

A more substantial example, taken from Chapter 4 Figure 4.1, is:

```
TimerProtocol ->
        start                       : Integer
```

Figure A.1 EBNF Description For Protocol Declaration

```
protocol_declaration ::= decl_header decl_body decl_end
decl_header ::=  identifier  "→"
decl_end ::= "end"  identifier
decl_body ::= {operation_list_decl}

operation_list_decl ::= {operation_decl}
operation_decl ::= op_name [argument_list] [op_return_type]
op_name ::=  identifier
argument_list ::= "(" [arg_decl","] arg_decl ")"
arg_decl ::= parameter_decl
parameter_decl ::= param_name ":" type_name
op_return_type ::= ":" type_name
type_name ::= "integer" | "real" | "char" | "string" |  identifier
param_name ::=  identifier
```

```
                stop(id  : Integer)
                delta(id : Integer)      : Real
        end TimerProtocol
```

This declares three operations which can be invoked on the protocol: `start`, `stop` and `delta`. `start` does not take any parameters but returns an integer value. `stop` and `delta` both take integer parameters but `delta` returns a real (floating point) value whereas `stop` does not return any value.

Protocol Definition

The source file given to the ftdl compiler must provide a *protocol definition.* This definition must of a previously declared protocol. A protocol declaration consists of statements which are executed at runtime. A protocol declaration maps to a process at run time in the ftdl runtime environment. The declaration of protocol begins with the keyword "define" followed by the name of the protocol to be defined. The EBNF grammar of the protocol declaration is shown in Figures A.2, A.3, A.5, A.6 and A.4.

Referring to Figure A.2, the definition body is broken up into two sections: the variable declaration section and main execution body. Variables must be declared before use. To define a variable, a unique name must be given and its type must be specified. The type may one of the built-in types or a previously encountered protocol declaration. An example of a variable declaration is:

```
        anInteger : Integer
```

This declared an integer variable known as *anInteger*.

Figure A.2 EBNF Description For Protocol Implementation

```
protocol_definition ::= def_begin {def_body} def_end

def_begin ::= "define"  identifier  "→"
def_end ::= "end"  identifier
def_body ::= {local_var_decls} {impl_statements}

local_var_decls ::= {var_decl ";"} var_decl
var_decl ::= variable_name ":" type_name
variable_name ::=  identifier
type_name ::= "integer" | "real" | "char" | "string" |  identifier

impl_statements ::= {impl_statement ";"} impl_statement

impl_statement ::= assignment |
                   id_statement |
                   repeat_statement |
                   if_statement |
                   remote_call |
                   rcv_statement
```

Following the variable declarations, the main execution body begins. This consists of zero or more implementation statements, which can be one of: assignmentstatement, id_statement, repeat_statement, if_statement, remote_call or rcv_statement. These are shown in Figure A.3.

Figure A.3 Statements in ftdl

```
assignment ::= variable_name ":=" arithmetic_expr
id_statement ::= "ID" "("  string literal  ")"
repeat_statement ::= begin_repeat repeat_body end_repeat
if_statement ::= if_header if_body endif
remote_call ::= prot_reference "." op_name {call_args}
rcv_statement ::= "rcv" which_rcv_op rcv_args
```

assignment takes two arguments: the target variable and an arithmetic expression. The target variable is a previously declared variable whose value will be set to the value of the right-hand side of the assignment operator (which is ":="). The type of the variable and arithmetic expression must be the same. So only an integer can be assigned to an arithmetic expression whose type evaluates to that of an integer.

Figure A.4 Expressions in ftdl

```
expression ::= arithmetic_expr | boolean_expr

arithmetic_expr ::= arithmetic_expr "*" arithmetic_expr |
                    arithmetic_expr "/" arithmetic_expr |
                    arithmetic_expr "+" arithmetic_expr |
                    arithmetic_expr "-" arithmetic_expr |
                    remote_call
                      identifier |
                      number |
                      string literal


boolean_expr ::= arithmetic_expr "=" arithmetic_expr |
                 arithmetic_expr "<" arithmetic_expr |
                 arithmetic_expr ">" arithmetic_expr
```

The grammar for expressions is shown in Figure A.4. The type of an expression is determined by the operator and its arguments. For the binary operators ("*","/","+","-","=","<",">"), both arguments must be of the same type. If the binary operators are boolean operators ("=","<",">") then the type of the expression is boolean. The arithmetic operators ("*","/","+","-"), however, evaluate to the type of their operands. No type conversions are performed.

The `id_statement` is used to name a protocol implementation. It takes a string type as its argument.

Two types of execution control statements exist in ftdl: `repeat`, shown in Figure A.5(a), and `if`, shown in Figure A.5(b).

The `repeat` statement is used to loop over a sequence of statements. The loop terminates when the boolean condition given to the `until` branch becomes true.

The `if` statement is used for branching. The body of the `if` is executed provided that the boolean condition evaluates as true. The statements dealing with remote operation invocation and reception are shown in Figure A.6. The

Figure A.5 Control Statements in ftdl

```
repeat_statement ::= begin_repeat repeat_body end_repeat
begin_repeat ::= "repeat" "→"
repeat_body ::= impl_statements
end_repeat ::= "until" bool_expr
```

(a) Repeat Statement

```
if_statement ::= if_header if_body endif
if_header ::= "if" bool_expr
if_body ::= impl_statements
endif ::= "end"
```

(b) If Statement

invocation statement, shown in Figure A.6(a), consists of the reference to a protocol (a local variable whose type is that of a previously encountered protocol declaration), followed by a "." and then the operation name and its arguments. Before any operation can be invoked, the Init operation must be called to initialise the reference. This takes as a parameter a string which is the name of the implementation to which this reference refers.

For the other operations, the compiler ensures that the operation being invoked is actually declared in the protocol declaration and all the parameters passed match the type in the declaration.

Figure A.6 Operation Invocation and Reception Statements in ftdl

```
remote_call ::= prot_reference "." op_name {call_args}
prot_reference ::= identifier
op_name ::= identifier
call_args ::= "(" parameter_list ")"
parameter_list ::= {expression ","} expression
```

(a) Remote Invocation Statement

```
rcv_statement ::= "rcv" which_rcv_op rcv_args
which_rcv_op ::= identifier
rcv_args ::= "(" {rcv_arg_list} ")"
rcv_arg_list ::= {target_var ","} target_var
target_var ::= identifier
```

(b) Reception Statement

Similarly with the **RCV**, shown in Figure A.6(b), the compiler ensures that the operation being received has been declared in the current implementation's protocol declaration and that all arguments' types match the declarations. When receiving an operation, the values are stored into variables, so it is important the `rcv_args` are all variables, previously declared. The exact number and type for the parameters must be given for the invocation and **RCV** statements.

A.2 Compilation

The compilation unit for the ftdl compiler is a single file. For code to be generated, this file contains at least a protocol declaration and only *one* protocol implementation – there can only be one protocol implemented per file.

The compiler itself is single pass and does not perform any preprocessing. In order to share a protocol declaration between its implementation and clients, the user can either copy the declaration in the implementation file and client file, as shown in Figure A.7.

Another approach is to use a preprocessor and invoke that prior to running the ftdlcompiler.

The compiler is invoked by supplying the name of the file containing the code as an argument. So to compile the two files in Figure A.7 the following commands would need to be executed:

```
1% ftdl prot_impl.ftdl
2% ftdl prot_use.ftdl
```

The compiler produces a file with extension `.gen`. This is loaded by the simulator.

A.3 End Remarks

Though ftdl is a small language it has most of the features required for building distributed applications. The basic program building blocks are built into

Figure A.7 Copy Approach To Sharing Protocol Declarations

```
-- File: prot_impl.ftdl
-- Implementation of Protocol

-- Protocol Declaration
--
Elevenses_protocol ->
        isPrime( given_num : Integer ) : Integer
end Elevenses_protocol


-- Server Implementation
define elevenses_protocol  ->
        ... implementation of
        ... protocol ''Elevenses_protocol''
end elevenses protocol
--
-- end of file prot_impl.ftdl

-- File prot_use.ftdl
-- A client who uses the Elevenses Protocol
--

-- The protocol(s) this client wishes to
-- use are copied here:
--
-- Protocol Declaration
--
Elevenses_protocol ->
        isPrime( given_num : Integer ) : Integer
end Elevenses_protocol


-- the client
--
Num_Sender ->
end Num_Sender

define Num_Sender ->
        ... the client code ...
end Num_Sender
--
-- end of file  prot_use.ftdl
```

the language: control structures in the form of branching and looping; variable declaration and storage. Also included are primiatives to allow the building of distributed programs – invoking and servicing operations, remote process naming and identification. By using these features useful distributed programs can be built which are ideal for simulations. One major area missing is that of non-determinism.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 1985.

[2] S. Ahuja, N. Carriero, and D. Gelernier. Linda and friends. *Computer*, 19:22–34, August 1986.

[3] C. Amza, A Cox, S Dwarkadas, P Keleher, H Lu, R Rajmony, W Yu, and Zwaenepoel W. Threadmarks:Shared Computing on Networks of Workstations. *Computer*, pages 18–28, Feb 1996.

[4] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[5] David E. Bakken and Richard D. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Computing*, 1995.

[6] Henri E Bal, Jennifer G Steiner, and Andrew S Tanenbaum. Programming Languages for Distributed Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[7] J.G.P. Barns. *Programming in Ada : Plus an Overview of Ada 9X*. Addison Wesley, fourth edition, 1993.

[8] Ben-Ari. *Principles if Concurrent and Distributed Programming*. Prentice Hall, Englewood Cliffs,N.J., 1990.

[9] A. Borg, J. Baumack, and Glaser S. A Message System Supporting Fault Tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 90–99, Bretton Woods NH, October 10-13 1983.

[10] Rational Software Corportation. Unified Modelling Language; Version 1.1. Internet. http://www.rational.com/uml.

[11] D.L. Detlefs, M.P. Herlihy, and J.M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *Computer*, pages 57–69, December 1988.

[12] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronization Needed for Distributed Systems. *Journal of the ACM*, 34(1):77–97, January 1987.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs,N.J., 1985.

[14] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[15] Leslie Lamport, R. Shostak, and M. Pease. The Byzantine Generals probelm. *ACM Transactions on Programming Languages and Systems*, 4(3):382–481, July 1982.

[16] John. R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly and Associates, 1992.

[17] Bill. Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, pages 52–60, August 1991.

[18] Fred B. Schneider. Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[19] Paul.G Sorenson and Jean-Paul Tremblay. *The Theory And Practice of Compiler Writing*. McGraw-Hill, 1985.

[20] W. Richard Stevens. *UNIX Network Programming. Networking APIs, Sockets and XTI.*, volume 1. Prentice-Hall, Upper Saddle River, N.J. 07458, Second edition, 1998.

[21] Bjarne Stroustrup. *The C++ Programming Language. Second Edition.* Addison Wesley, 1993.

[22] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs,N.J., 1996.

[23] Andrew. S. Tanenbaum. *Computer Networks. Third Edition.* Prentice-Hall, Upper Saddle River, N.J. 07458, 1996.

[24] John Turek and Dennis Shasha. The Many Faces of Consensus in Distributed Systems. *Computer*, pages 8–17, June 1992.

[25] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2), February 1997. also avaiable from:http://www.iona.com/hyplan/vinoski/#articles.