Reliable Processing on the Seljuk-Amoeba Operating Environment

Érica de Lima Gallindo, Francisco Vilar Brasilheiro, and Vladimir Soares Catão

Universidade Federal da Paraíba - UFPb/Campus II
Centro de Ciências e Tecnologia - CCT
Departamento de Sistemas e Computação - DSC
Laboratório de Sistemas Distribuídos - LSD
Av. Aprígio Veloso, 882, Bodocongó
58109-970, Campina Grande, Paraíba, Brazil
http://www.dsc.ufpb.br/~lsd
{erica, fubica, vlad}@dsc.ufpb.br

Abstract

Processing on the Amoeba distributed operating system is not fault-tolerant. The only concern of its processing service is to perform load balancing on the existing processors, trying to find the processor that best suits a particular process in terms of memory availability and CPU speed. In this paper we introduce a fault-tolerant processing service for the Amoeba system. This service is used in the implementation of the Seljuk-Amoeba operating environment, which offers processing service with different controlled failure semantics on a per-process basis.

1. Introduction

Due to its inherent redundancy, a distributed system may provide a service that is more dependable than that provided by a centralised one. However, for a particular distributed application to attain high dependability levels, it is necessary to carefully design both the application and its execution environment. Particularly, the failure of an isolated processing site, or a network connection, should have small impact, if any, on the operation of the system as a whole. This leads to the necessity of introducing fault tolerance mechanisms into the system.

The complexity of the fault tolerance mechanisms introduced is directly affected by the failure semantics of the underlying components over which the mechanism is implemented; the less restrictive the failure semantics the more complex the mechanism implementation [6].

When choosing a failure semantics for a processor, besides hardware properties, other important aspects must also be considered. These include the dependability requirements of the application as well as the duration of its execution (time of mission). For instance, although it is known that conventional processors may fail in an arbitrary way (Byzantine failures) [7], the probability of this kind of failure occurring is so small, that it is possible to assume a controlled failure semantics for these processors, and still guarantee the reliability requirements of a large number of applications. On the other hand, there is an increasing number of applications whose dependability requirements are strong enough so to rule out the assumption that conventional processors have controlled failure semantics. For these applications it is necessary to build nodes which ensure the assumed controlled behaviour.

Fail-controlled nodes may be built by means of coupling conventional processors, which fail independently, and in an arbitrary way. Computation is replicated and executed simultaneously on each processor forming the node. The outputs generated by the replicated processors are then evaluated by a suitable mechanism (e.g. comparator, majority voter, etc.), which ensures the node failure semantics. [3] introduces protocols which can be used in the implementation of software based replicated nodes with different fail-controlled semantics.

[2] proposes a model, named Seljuk, which can be used to implement development platforms to facilitate the construction of reliable distributed applications. A Seljuk platform must offer a reliable processing service which allow developers to assume that applications execute over nodes which fail in a controlled and predictable way. At activation time each application can choose the failure semantics of the processing service to be used. Whenever the processors, over which the service is implemented, have a failure semantics less restrictive than that assumed
presented in Section 6.

despite the occurrence of a bounded number of semantics, that is, the node delivers its standard service semantics which is equivalent to their operational fail-silent nodes. Failure-masking nodes possess failure discussed. This family encompasses failure-masking and fail-controlled nodes, named the Voltan family, is the required node failure semantics. Values from appearing at the application level, ensuring such as a comparator or a voter, which avoids incorrect processors are then evaluated by a filtering mechanism, the same output stream. Outputs generated by replicated processes in the same processors, each non-faulty implementation.)

2. Fail-controlled nodes

Many researchers have proposed ways of building both hardware and software implemented fail-controlled nodes [7, 1, 19, 14, 15, 16, 3, 4]. The basic approach is to replicate the computation on a sufficiently large number of independent processors, which can fail in an arbitrary way. Processors are driven by some synchronisation mechanism which guarantees that non-faulty processors will produce the same output stream. Outputs generated by replicated processors are then evaluated by a filtering mechanism, such as a comparator or a voter, which avoids incorrect values from appearing at the application level, ensuring the required node failure semantics.

In [15] the design of a family of software based fail-controlled replicated nodes, named the Voltan family, is discussed. This family encompasses failure-masking and fail-silent nodes. Failure-masking nodes possess failure semantics which is equivalent to their operational semantics, that is, the node delivers its standard service despite the occurrence of a bounded number of components failures, which are masked. On the other hand, fail-silent nodes possess a safe failure semantics, that is, after the detection of the failure of any node component, the node neither delivers its standard service, nor unspecified ones, and simply halts.

[16] and [4] describe implementations of Voltan nodes at the application level, which use efficient replica management protocols presented in [3]. The Seljuk-Amoeba fault-tolerant processing service will provide both failure-masking and fail-silent processing services, by introducing the protocols studied in [3] at the microkernel level of the Amoeba distributed operating system.

It is assumed that (non-replicated) distributed applications are composed of a number of processes that do not share memory, and interact only via messages. In this paper we also assume that application processes present a deterministic behaviour. (Elsewhere we have discussed ways of dealing with applications which incorporate non-deterministic behaviour [18].) Thus, synchronisation among replicas is attained by simply ensuring that each application process replica receives the same stream of input messages in the same order, thus yielding identical streams of output messages to be produced by non-faulty replicas.

We consider failure-masking nodes and fail-silent nodes comprised of \( N \) processors, where \( N = 2\pi + 1 \) in the case of failure-masking nodes, \( N = \pi + 1 \) in the case of fail-silent nodes; and \( \pi (\pi > 0) \) is the node’s resilience degree, i.e. the upper bound on the number of processors of a node that may fail. The restrictions \( N = 2\pi + 1 \) and \( N = \pi + 1 \) are only necessary to assure that the validating techniques work well. The validating techniques operates on the output messages produced by the replicas of the application. All valid messages produced by a node possess \( \pi + 1 \) signatures, thus ensuring that at least one non-faulty processor has participated in the validation process. We assume that mechanisms exist for generating and validating digital signatures, which provides an authentication facility with arbitrary high probability [13].

We also assume that intra-node communication between any two non-faulty processors is synchronous, i.e. there is a maximum bound \( \delta \) for message processing and transmission between any two non-faulty processors. (In [5] we show how this assumption can be achieved on an asynchronous network, such the one we are using on our implementation.)

Figure 1 shows the Voltan node architecture. In addition to the application processes, each non-faulty processor of a node runs five other processes, namely: Receiver, Transmitter, Order, Validator and Sender processes. In order to communicate, processes in the same processor use message passing over shared queues and lists, whilst processes in different processors use message passing over internal communication links.
We now briefly describe the function of each system process.

**Sender process**: this process takes messages deposited into the Processed Message Queue (PMQ), that have been produced by the application processes of that processor, signs and sends them to the other processors of the node for validation, i.e. voting in failure-masking nodes, and comparison in fail-silent nodes. Further, it deposits a copy of the messages into the Internal Candidate List (ICL).

**Validator process**: the function of this process depends on the type of the node. In failure-masking nodes, the Validator process is a Voter process. It compares authentic messages deposited into the External Candidate List (ECL) which have been signed and sent by other processors of the node, with their counterparts produced locally and that have been deposited into the ICL by the Sender process. If the comparison is not successful, the message is discarded. Otherwise, the message is countersigned, and if there are now $\pi+1$ signatures, the message is handed over to the local Transmitter process for network delivery to destination nodes. If there are less than $\pi+1$ signatures, then the message is sent to the other processors of the node that have not signed the message yet. In fail-silent nodes the Validator process is a Comparator process. Its functioning is similar to the Voter process, with only one difference: once a failure is detected, instead of simply discarding the received message, the Validator process terminates its activities, and so does the Sender process. This guarantees that the node will remain silent after a failure.

**Transmitter process**: this process is responsible for retrieving the $\pi+1$-signed messages deposited into the Validated Message Queue (VMQ) and sending them over the network to the destination nodes.

**Receiver process**: this process authenticates messages received from the network or from the internal links and discards any message which fails authentication or any duplicated message received. Authenticated and valid messages received from the network are deposited into the Received Message Queue (RMQ) for ordering, whilst authenticated messages received from other processors of the node, which carry less than $\pi+1$ signatures, are deposit in the ECL for validation.

**Order process**: this process executes an order protocol with its counterparts in the other processors of the node. The function of the order protocol is to construct identical queues of valid messages received from the network for processing by the local application processes of all non-faulty processors of the node. Ordered messages are deposited into the appropriate Delivered Message Queue (DMQ).

### 3. Amoeba’s basic concepts and structure

Amoeba is a distributed operating system based on microkernel technology. The basic principle of this technology is to minimise the part of the operating system which executes in kernel mode (the microkernel), with the goal of increasing system flexibility. Every Amoeba processor runs a small microkernel which is responsible only for the minimal functionality of the system. All the operating system’s functionality which is not provided by the microkernel is supplied by server processes which execute in user mode.

#### 3.1. Objects and capabilities

All Amoeba servers work based on the object concept. An object is an encapsulated piece of data upon which certain well defined operations may be performed. Each object is managed by an object server process. Operations on an object are performed sending a message to the object’s server.

When an object is created, the server returns a capability to the process which creates it. Capabilities have coded into them the set of operations that the holder may carry out on the object and they also contain enough redundancy and cryptographic protection information to make it unfeasible to guess an object’s capability. A typical capability is shown in Figure 2.

![Figure 2: Example of a capability](image-url)

<table>
<thead>
<tr>
<th>Server Port</th>
<th>Object</th>
<th>Rights</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 bits</td>
<td>24 bits</td>
<td>8 bits</td>
<td>48 bits</td>
</tr>
</tbody>
</table>
some mechanism is needed to prevent users from tampering with them. The check field holds encrypted information to achieve this protection feature.

3.2. Processes and threads

The key element of the Amoeba’s process management mechanisms is the process descriptor. It is a portable structure that describes the state of a running process. A process descriptor consists of four parts. The first part is the host descriptor. It describes the properties of the host on which the process may run (e.g., CPU architecture, memory requirements, etc.). A process can only run on a host that has the properties matching those in its process descriptor.

The next part, the memory descriptor, describes the layout - but not the contents - of a process’ virtual address space. Virtual memory is segmented, and for each segment, there is a segment descriptor with the following fields: virtual address, length, how and source. The virtual address field indicates where the segment is mapped into the address space; the length field indicates how long the segment is; the how field describes what sort of access the process has to the segment (e.g., read-only, execute-only), and whether the segment can grow or not and in which direction it can grow; the source field is a capability for a file-like object that provides the contents of the memory segment.

The third part of the process descriptor is the thread descriptor. Amoeba has kernel-space threads, that is, the kernel manages the individual threads of a process. For each thread, the thread descriptor contains: program counter, stack pointer, processor status word, registers and the state of blocking system calls.

The last part of the process descriptor contains two capabilities needed for process management. The first one is the capability for the process; a process is viewed as an object which is managed by the kernel of the host on which it runs. Requests for operations on the object (e.g., suspend, migrate, kill) are sent to its managing server, i.e. the kernel. Thus, when a process is started at a host, the kernel needs to have a capability with which requests for operations are addressed. The second one is a capability for the exception handler of a process. Usually, it is its parent process, but sometimes it may be a debugger. When an exception occurs, the kernel builds a processor descriptor for the process and sends it to the exception handler.

3.3. Communication in Amoeba

Amoeba supports both Remote Procedure Call (RPC) and Group Communication (GC). In order to support them, Amoeba provides a network protocol called FLIP - Fast Local Internet Protocol [8].

Figure 3 shows the organisation of the communication layers into the Amoeba’s microkernel. It contains two layers: an upper layer implementing RPC and GC and a lower layer that implements FLIP. The RPC and GC protocols are session layer protocols, and since there is no transport layer, the reassembling of fragments in order to restore messages and the re-transmission of lost messages is performed at RPC and GC levels.

3.4. Amoeba’s servers

Server processes are important components of the Amoeba system. Apart from the basic functions provided by the microkernel that executes on each processor of the system, all the system’s functionality is provided by user level servers; these include servers for providing file, directory, boot and execution services. In this section, we will pay special attention to Amoeba’s processing service which is implemented by a user level server - the Run Server, responsible for higher level scheduling and load balancing, and a kernel level server - the Process Server, responsible for basic process management.

Under Amoeba, tasks are submitted to the Run Server through its command’s interpreter (the shell). When a user types in a command on the terminal, the shell extracts the first word of the command line, assumes that it is the name of an executable program, looks for this program into Amoeba’s File Server, and in finding it, it takes steps to execute the program.

First of all, the shell tries to find the architectures for which the program is available. To do this, it looks into the /bin directory. If that program is available for multiple architectures, then it will be present not as a regular file, but as a directory. This directory contains executable programs for each available architecture. The shell then does an RPC with the Run Server sending it all possible process descriptors and asking it to choose an architecture and a specific CPU.

Hosts in Amoeba are grouped into pools. Pools can be heterogeneous (i.e. consist of hosts of different architectures). A host may also be present in more than one pool at a time. The Run Server is responsible for
maintaining status information (e.g. CPU architecture, CPU speed, amount of free physical memory currently available, average number of executable threads in the last few seconds, etc.) about all hosts in the pools registered with it. Based on this information, the Run Server is able to select the most suitable host to run a given process. After choosing such host, it then returns to the shell an exec capability for the Process Server on the selected host.

The shell then does an RPC with that server, asking it to effectively create the process. Figure 4 shows all the steps involved on creating a process under Amoeba.

![Figure 4: Amoeba’s process creation](image)

4. A fault-tolerant processing service

As seen in the previous section, Amoeba’s processing service does not provide any support for fault tolerance. In this section, we show how the processing service of the Amoeba system can be upgraded to provide fault-tolerant processing service. This is achieved by developing a new Run Server, named Fault Tolerance Run Server (FT Run Server), which is responsible for higher level replicated scheduling, and by introducing a new kernel level server - named Node Server, which is responsible for managing the replicated nodes formed by the FT Run Server. This new environment with provision for fault-tolerant processing we have named Seljuk-Amoeba.

4.1. The FT Run Server

Applications executed by the FT Run Server, may choose, at activation time, the failure semantics of the node where they will be executed, as well as the effective failure semantics of the underlying processors. The FT Run Server provides the required failure semantics, in a transparent way, by replicating processing on enough processors and ensuring that a suitable validation function will be applied to the outputs generated by them. The choice of the number and types of processors to use in a node depends on the application’s reliability requirements.

Besides tolerating physical faults, the nodes formed by the FT Run Server are also able to tolerate hardware and software design faults. Since Amoeba supports heterogeneous processors, it is possible to achieve fault tolerance at hardware design level by executing a process on a node formed by processors with different architectures. Further, if different versions of a program are available, it is also possible to tolerate software design faults by executing different versions of the process on each of the processors forming a node.

In order to allow software design fault tolerance, the executable program organisation under Amoeba was adapted. In its original version, Amoeba only deals with different versions of a program for different architectures. We have made changes to the Amoeba File Server organisation so that a particular program may have several versions for each architecture. The different versions are independently designed to satisfy the specification of the program. Figure 5 shows the /bin organisation, after the introduction of these changes.

![Figure 5: Seljuk-Amoeba file system](image)

In this example, the /bin directory contains two commands: `dir` and `sort`. The `dir` command is available only for VAX architecture, whilst the `sort` command has three implementations available for the 80386 architecture. This arrangement allows that application designers, ask the FT Run Server to execute their applications in replicated nodes, with each replica executing a different implementation of that program. Note that with the diversity of processor architectures and the availability of different versions for each replica executing on them, the degree of fault tolerance attained can be very high.

Now let us see how applications can take advantage of the service provided by the FT Run Server. Suppose a shell desires to execute a process in a fail-controlled node; its first step is to do an RPC with the FT Run Server. In addition to all available process descriptors, a number of extra parameters must be provided. First, it is necessary to inform the failure semantics of the system’s processors, over which fail-controlled nodes will be built. Further, since application programmers are able to choose between fail-silent and failure-masking semantics, one of the parameters provided to the FT Run Server is the failure semantics of the node where the application will run. The node resilience level is also provided by the shell as a parameter. The FT Run Server uses it for deciding how many independent processors will form a node. Another parameter passed by the shell, is a list which contains the identification of processors that the application does not want to use (this parameter may be used, for example, if an application designer thinks that a given processor is not
reliable enough, and does not want to submit any task to it, or to prevent processors to take part on two or more nodes where processes involved in the same distributed application execute). Finally, a last parameter added to the RPC is a flag that indicates whether the application will be execute with design diversity or not. Figure 6 shows a process creation scenario using the FT Run Server.

Figure 6: Seljuk-Amoeba’s replicated process creation

After doing an RPC with the FT Run Server, the shell stays blocked waiting for a reply. The FT Run Server returns to the shell a descriptor for the created node. This node descriptor contains the following information: a node identifier, a list containing all processors in this node (the first one being the node co-ordinator) and the corresponding process descriptors that have been chosen for each processor, and a capability for the node. Then, the shell does an RPC with the Node Server of the co-ordinator processor, asking it for process creation (see Figure 6).

The Node Server of the co-ordinator processor asks the local Process Server to create a process and sends the same request to all other processors of the node. The Node Servers of these processors, in turn, ask to their respective Process Servers to create the replicas. The Node Server as well as the Process Server are implemented into the Seljuk-Amoeba’s microkernel.

4.2. Microkernel level processing management

Nodes formed by the FT Run Server use the mechanisms described in Section 2, for replica control. Thus, it is necessary to adjust Seljuk-Amoeba’s microkernel to implement these protocols. Figure 7, shows how the threads, queues and lists shown in Figure 1, as well as the information stream between them, are logically merged within the Seljuk-Amoeba’s microkernel, in order to implement a reliable processing service. In Section 4.2.1 we give details of how the replica management functionality is implemented.

When a message arrives at a network adapter of a processor its destination must be considered, in order to decide whether the message is going to be ordered or not. If the message has been sent to a replicated process, it must be ordered before being relayed to the part of the kernel that implements message receiving in the higher level protocols (represented in Figure 7 by RPC/GC). In this way all application replicas (Application) are going to receive the same messages and in the same order. Otherwise the message must be immediately relayed to RPC/GC. Another two types of messages may be received by a processor: messages sent by the other Order threads (of the other processors of a replicated node), which are relayed to the local Order thread; and messages sent by the other Validator threads (of the other processors of a replicated node), which are delivered to the local Validator thread for validation. Messages produced by the application processes (Application), are deposited into the PMQ. After this, the message is delivered to the Sender thread. If the message has been generated by a replicated process, it is deposited into the ICL and stays there waiting for a message into the ECL that matches it; also a copy of the message is sign and sent to the other processors of the node. Otherwise, the message is immediately relayed to the part of the kernel that implements message delivery into the higher level communication protocols (represented in Figure 7 by RPC/GC). After being processed by RPC/GC, the message is delivered to the Transmitter thread which sends it to its destination.
4.2.1. Implementation details

As seen in Figure 7, the replica management protocols required to implement a reliable processing service are merged into the communication protocols of the microkernel. Also, for each process executing on a particular processor, the microkernel must know whether the process is replicated or not, and in the case it is replicated, which validation function to use, and on which processors the counterpart replicas are being executed. All these information are kept at microkernel tables, and initialised by the respective Node Server.

Replica management consists basically of ordering input messages and validating output messages. We have decided to implement these functionalities at the higher level communication protocols, i.e. at the RPC and GC layers. Thus, the Receiver and Transmitter threads shown in Figure 7 are implemented by the core reception/transmission functions of the FLIP protocol. We have however added new functionality to the FLIP layer, in order to implement a synchronous communication channel for the exchange of ordering and validation control messages with bounded transmission delays. (Details of this service have been reported in [5].)

We now describe how the higher level communication protocols have been modified. Due to space limitations, we will only discuss the RPC layer. We start with a brief introduction to RPC in Amoeba, and then we show how the protocol can be modified to accommodate ordering and validation.

All point-to-point communication in Amoeba is structured along the client/server model; that is, a client thread sends a request message to a server, one of the server’s threads gets the request, carries it out and returns a reply message to the client thread. The RPC layer identifies processes in the network through FLIP addresses. Each client or server application uses only one FLIP address (randomly generated) for sending/receiving messages. Additionally, server applications are addressed by RPC ports. Thus, the pair FLIP address plus RPC port uniquely identifies a service under RPC.

The RPC interface offers three primitives for interprocess communication, namely getreq, putrep and trans. The client thread invokes a transaction (a request followed by a reply) by calling the primitive trans. A server thread receives a request via a call to the primitive getreq and returns a reply by calling the primitive putrep. Trans, getreq and putrep are blocking primitives; that is, a call to trans suspends a thread until the request is sent, carried out and a reply is received; getreq suspends a thread until a request has been received and putrep suspends a thread until the reply has been received by the kernel executing on the host where the client executes.

When a client calls the trans primitive, before the request message can be sent, it is necessary to find out in which FLIP address there is a server listening to the desired RPC port. Thus, the trans primitive broadcasts a LOCATE message to the network. This message also contains the RPC port that the client desires to reach. It then blocks waiting for the response of a server.

After receiving a LOCATE message, every kernel in the network verifies if there is any server listening to the desired port. If there is such server, the kernel then sends a HEREIS message back to the client.
Once a server is located, the *trans* primitive now sends a REQUEST message to the server (identified by the RPC port plus the FLIP address). The server’s FLIP address is cached, obviating the need of a LOCATE broadcast on future interactions. When the REQUEST message arrives on the host where the server is executing, the thread which issued a *getreq* is unblocked and receives the request message. The server then effectively executes the RPC, and sends back the reply with the result of the RPC through a call to the *putrep* primitive. In the client’s side, the *trans* call is unblocked and the response is delivered to the client.

The first problem to be solved is that of addressing both replicated clients and replicated servers. The model in Figure 7 assumes that a client message sent to a server is broadcast to all server replicas in the node. Also, if the client is replicated, replies must be sent to all client replicas. Unfortunately, this is not the case with the RPC protocol under Amoeba; if more than one server responds to the LOCATE message, the *trans* primitive will send the request via a unicast to only one of them. Further, reply messages are sent via unicast to the client address that comes in the request message.

Replicated server addressing is solved as follows: the HEREIS message sent by every server will contain information that allows the client microkernel to distinguish HEREIS messages sent by replicated servers from those sent by non-replicated servers; when a request has to be sent to a replicated server, instead of sending it via unicast to one of the servers, the *trans* primitive will send a multicast to all replicated servers that have answered to the LOCATE message. On the replicated client side, addressing is solved as follows: a request message from a replicated client carries the addresses of the \( \pi + 1 \) clients that have validated the request message; thus, the server can send its response via a multicast to these client replicas; since at least one non-faulty client will receive the message, the ordering protocol executed by the processes forming the replicated node where the client executes, guarantees that all non-faulty clients will receive the server’s response.

The second problem to solve is that of ordering. As mentioned before, messages received at the RPC layer may have been fragmented by the underlying FLIP layer, and therefore must be reassembled by the RPC protocol. We have decided to apply our ordering protocol to the individual fragments of messages instead of applying it to reassembled messages. Since the reassembling of fragments is a deterministic operation, provided that all non-faulty processes will receive the same fragments in the same order, they will produce the same input messages in the same order. Spurious (i.e. corrupted messages or messages with incorrect signatures) and duplicated messages are dealt with after the reassembling operation is finished. Fragments from messages addressed to non-replicated processes need not be ordered and follow the standard protocol. Fragments addressed to replicated processes are sent to the other processors that hold replicas of the destination process through the synchronous communication primitives provided by the enhanced FLIP layer. These fragments are identified by a new message type (ORDER) that we have introduced into the RPC protocol. Stable fragments are ordered and handed over to the reassembling function. (See [3] for details on ordering protocols for both fail-silent and failure-masking nodes.)

Finally, we need to validate output messages (i.e. replicated client’s requests and replicated server’s replies). This is achieved in the following way. When a replicated client calls *trans* or a replicated server calls *putrep*, first it is calculated a message digest for the output message [12], this is then signed and sent (via the synchronous channel) to the processors executing the other replicas. These messages are tagged with the VALIDATOR type, which is also a new type of messages that we have introduced into the RPC protocol. VALIDATOR messages received are sent for validation (either comparison, in the case of fail-silent nodes, or majority voting, in the case of failure-masking nodes). When \( \pi + 1 \) replicas have signed the message digest of a valid message, the \( \pi + 1 \)-signed message digest is appended to the original message and handed over for delivery.

5. Related work

Most fault-tolerant architectures reported in the literature are ad-hoc solutions to specific problems (see [7], for example). Others, like [1] and [19], are suitable for a broader range of applications (on-line transactions processing, in the aforementioned case). However, in both cases, some problems arise: firstly, their reliable processing service is heavily dependent on a proprietary hardware architecture; secondly, applications with no dependability requirements will have to pay, normally with a reduction on their performance, for unwanted reliable services; finally, technological advances (e.g. faster processors) can only be incorporated into these system, after substantial redesign has been carried out.

The Delta-4 architecture [11] is one of the first general purpose fault-tolerant architectures proposed in the literature. Its approach is to offer an open dependable distributed computing system that attempts to use as much as possible off-the-shelf components, accommodating heterogeneity of the underlying hardware and software, and providing application portability across many platforms. The architecture offers specifiable dependability levels and provides support for replication of software components, fault diagnosis and system reconfiguration.
The Seljuk model follows many ideas put forward by the Delta-4 project, but applicable to a different setting. Its approach is to enhance the services provided by a distributed operating system (Amoeba, in the case of Seljuk-Amoeba) with the provision of reliable processing and fault tolerance services. Further, unlike Delta-4 which requires special hardware to provide the fail-silent semantics required for the execution of communication software and replica co-ordination entities, Seljuk-Amoeba incorporates the necessary replica management protocols to provide reliable processing using fail-arbitrary off-the-shelf processors.

The distributed operating system ROSE [10] is a system that follows Seljuk-Amoeba philosophy of offering fault tolerance services at the operating system level. It provides a Replicated Address Space object abstraction that makes portions of an address space highly available through replication. Based on such objects a Resilient Process abstraction is provided to application processes transparently. However, ROSE presents two restrictions when compared to Seljuk-Amoeba: i) it assumes that the system underlying components have fail-silent semantics; and ii) it tolerates only physical faults.

6. Conclusions

The reliable processing service of the Seljuk-Amoeba operating environment is an important tool for the construction of robust distributed applications. It provides a reduction in the applications’ development complexity by allowing programmers to assume that the node failure semantics over which their applications execute is more restrictive than that provided by the underlying processors. The Seljuk-Amoeba is in charge of the management of redundant processors needed to ensure the node failure semantics assumed.

This service also offers flexibility to applications, by allowing them to choose their failure semantics at activation time. In this way, if the dependability requirements of the application change, it is possible to provide the required service without the need for recompiling the application, since the reliable processing service is implemented at the level of the Seljuk-Amoeba microkernel.

Another advantage of the service is that its cost (in terms of performance reduction) is paid only by those applications with dependability requirements. Note that when the failure semantics of the available processors is at least as restrictive as the failure semantics required by applications, the processing service of the Seljuk-Amoeba system behaves exactly as the original processing service of the Amoeba system, and does not impose any performance reduction to applications.

Acknowledgements

The authors would like to thank the Brazilian Research Council (CNPq) for its financial support (grants 180.301/94-2 and 300.646/96-8).

References


