Abstract

In this paper we present the design of fault-tolerant processing services for asynchronous distributed systems upon which dependable applications execute. These services provide mechanisms for hardware (processor) and software (application server) fault tolerance which can increase the reliability and availability degree of applications that use them. The implementation strategy proposed is based on the object oriented paradigm. We analyse factors that affect the dependability as well as the cost of the services proposed. Since these services are likely to be used by applications with some degree of criticality, it is important to have their design and implementation validated. Thus, we discuss a strategy that facilitates the validation of practical implementation of distributed protocols for asynchronous distributed systems.

We follow a hybrid approach that combines two classical asynchronous system models exploring their advantages at the same time that circumvents their drawbacks.

1 Introduction

In the last few years, with the increasing popularity of distributed applications, the number of applications with dependability requirements has greatly increased. For these applications, failures in hardware or software components of the system can cause undesirable consequences, such as loss of money, clients, information and confidentiality. To attain their dependability requirements, these applications must execute extra mechanisms that are able to tolerate faults.

Usually, mechanisms for fault tolerance are introduced at the application level, making the task of developing dependable applications more complex. In order to reduce this complexity, the application programmer should ideally use mechanisms provided as services by the execution platform, without needing to know how they were implemented. Based on this observation, developers of dependable distributed applications have adopted two complementary approaches: i) the use of appropriate developing tools to support the implementation of the application (e.g. libraries of functions for fault tolerance programming toolkits and specialised services of operating systems and ii) the choice of the most restrictive failure semantics possible for the components that form the system’s underling execution layer thus reducing the complexity of the mechanisms for fault tolerance developed at higher levels.

In this paper we discuss the deployment of fault-tolerant processing services for asynchronous distributed systems. We first describe the design of fault-tolerant processing services. These services are designed to make good use of the natural redundancy of a distributed system (processors, communication channels, etc) and include the mechanisms required to tolerate both hardware and software faults. The basic idea is to restrict the failure semantics of the hardware (processors) and software (application) components
that form the distributed system’s underlying execution layer, so that, processing failures can be either masked or propagated to the application level with a more restrictive semantics [1].

Dependable distributed applications are, in a lesser or greater extent, critical. This fact yields the necessity of validating all services being used by these applications. To facilitate this task, a number of system models have been proposed in the literature (e.g. the synchronous model [2], the (time-free) asynchronous model [3], the (time-free) asynchronous model augmented with unreliable failure detectors [4], the timed asynchronous model [5], etc), each one having its own advantages and disadvantages. To validate the services presented in this paper we propose a hybrid system model that gathers the facilities of two well known system models - the time-free asynchronous system model augmented with unreliable failure detectors and the timed asynchronous system model - eliminating their respective inconveniences. We use a generic strategy that allows the specification of simple and practical solutions for asynchronous distributed systems.

The processing services described in this paper provide tolerance to hardware and software faults using well known mechanisms, such as active replication [6], recovery blocks [7] and $N$-version programming [8]. Thus, the main contribution of this paper is the strategy that has been used for combining these mechanisms in order to provide reliability and availability in a flexible way. Further, the approach used to validate the services is new and can be applied in a generic way to any distributed protocol for asynchronous systems.

The rest of this paper is organized as follows. In Section 2 we describe the fault-tolerant processing services. Then, in Section 3, we study the reliability and availability of the services presented; the main parameters that can affect these measures are discussed. Section 4 discusses a strategy to validate the services. Finally, Section 5 concludes the paper.

2 Fault-Tolerant Processing Services

2.1 Assumptions

We assume that applications execute over an off-the-shelf distributed system, where both communication and processing delays are unbound (i.e. an asynchronous system).

Applications are structured in terms of clients and servers. For the sake of simplicity, we will consider that only the server entities of an application use the fault-tolerant processing services proposed. Further, servers are complex entities, this means that they are prone to contain design faults and, therefore, their failure semantics is assumed to be arbitrary.

Although off-the-shelf processors can fail in an arbitrary manner, they usually fail by crashing. The failure semantics assumed for the hardware depends not only on the physical properties of the hardware, but also on the dependability requirements of the applications that execute on it. We consider applications with low degree of criticality, thus it is acceptable to assume that off-the-shelf processors fail just by crashing - a failure semantics less restrictive than the real one.

Under these assumptions, and in order to obtain processing services that provide reliability and availability, we propose specialised mechanisms to tolerate crash processor faults and arbitrary application faults. The mechanisms are the following: recovery blocks [7] and $N$-version programming [8] to tolerate software faults; and active replication [6] to tolerate hardware faults. Next, we present the fault-tolerant processing services proposed. We first describe how software faults are tolerated and later we show how hardware faults are dealt with.

2.2 Tolerating Application Design Faults

We propose two distinct classes of services; one is based on the recovery block approach and the other on the $N$-version programming approach. In both classes, each application server is formed by $N$ redundant modules which follow the same specification but are developed using design diversity. All redundant modules are executed on top of the same processor.

---

1 A typical example of an off-the-shelf distributed system is one represented by an Ethernet network of workstations, where each workstation execute a Unix-like operating system and the communication among processors is achieved via Internet protocols, such as, TCP/IP.
Following the recovery block approach, each redundant module of a server is executed sequentially and until the output generated by one of them is deemed correct by an acceptance test. It is assumed that the acceptance test is reliable and every recovery block is composed by \( N = f \) redundant modules, where \( f \) represents the number of faults that are tolerated. If, for an specific input, all redundant modules of a recovery block fail, then an exception is raised, meaning that the block has failed. Notice that even when the block fails, the failure semantics is safe, i.e. it crashes.

For the \( N \)-version approach, all redundant modules that implement a server are executed in parallel. Fault free versions produce identical outputs for the same inputs which are processed at the same order. Output validation is performed using majority voting or comparison schemes. In the first scheme, failures are masked, so, for each replicated server there are \( N = 2 * f + 1 \) modules. In the second scheme, the outputs produced are compared; if they do not match, then the processing service stops functioning, i.e. it fails by crashing. In this case, the number of modules required for each replicated server is \( N = f + 1 \) (the detection of incompatible outputs through comparison requires at least one non-faulty module).

Note that, using the processing services discussed above arbitrary failures can be either masked or propagated with a more restrictive failure semantics (crash). It is important to notice that, as the redundant modules of a server execute on top of a single processor, if it fails the processing services will also fail, and the failure semantics is crash. In the next section we discuss how processor failures can be masked.

2.3 Tolerating Processor Faults

We have chosen the active replication strategy to tolerate hardware faults. Every server that is implemented by redundant modules is replicated in different processors that fail independently, and these replicas are executed in parallel. All replicas process every input request in the same order, producing, in absence of faults, identical outputs. As application failures are masked or propagated with a safe semantics and processors can fail just by crashing, the final result of the replicated processing will be given by any output produced by any replica of the server. So, the active replication strategy can be used without voting.

To avoid replica divergence it is required that every correct processor receive the same input values and process these values in the same order. Considering the failure semantics assumed for the processors and replicas executing on top of them, the referred requirements can be assured through an ordering protocol.

The execution of the fault-tolerant processing service just described is illustrated in Figure 1. In this case, there are \( N \) processors \((P_1, P_2, \ldots, P_n)\) executing replicated computation and an Order module that implements the ordering protocol. Further, each replica \( R_i \) represents a copy of a server implemented according with the recovery block or \( N \)-version programming approach previously discussed.

![Figure 1: Fault-Tolerant Processing](image-url)
order. Output messages produced by any replica $R_i$ are transmitted to the client that requested the service. Duplicated messages are filtered on the client side.

### 2.4 Implementation Issues

In practice, to provide the processing services being proposed it is necessary to implement the mechanisms added to tolerate processor faults (active replication) and application design faults (recovery blocks and $N$-version programming approaches). In this sense, we suggest an implementation strategy at the programming language level. The objective is to use an object oriented programming language (e.g. Java) in order to implement the referred mechanisms through specialised classes and interfaces. Application specific functionality can then be added by extending the existing classes and implementing the interfaces defined.

#### 2.4.1 Providing the Mechanisms for Application Fault Tolerance

In the case of using the recovery block approach, it is necessary to define: 1) the redundant modules which implement the application server; 2) the acceptance test; 3) the procedures which are responsible for saving and restoring checkpoints (checkpoint operations); 4) a procedure that is responsible for executing all redundant modules according with the recovery block approach; and 5) a mechanism to raise an exception when the block operation does not succeed.

The execution of a recovery block follows the steps described below:

1. Save checkpoint
2. Execute module $i$
3. Submit output produced by module $i$ to the acceptance test
4. If acceptance test is satisfied then:
   - finish the recovery block operation returning the result accepted by the test
   else:
   - restore checkpoint
   - if there is another module to execute, then go to 2; else raise an exception

Using the Java language it is possible to provide the service through the following classes and interfaces.

*Exception classes.* Two classes are required. One to deal with failures of the recovery block as a whole (when the block operation does not succeed) - let this class be named `RecoveryBlockFailureException`, and another to cope with individual failures of redundant modules - let this class be named `AlternateExecutionFailureException`. These classes can be extensions of the `RuntimeException` class available in the Java API.

*RecoverableBlock interface.* It contains the declaration of methods that are responsible for implementing each of the requirements expressed by items 2, 3, 4 and 5, above (see Figure 2).

*RecoverableBlockImpl class.* It implements the interface `RecoverableBlock`, i.e. the methods defined in this interface. Also, this class must implement the redundant modules of an application server. Thus, for each recovery block of an application server there is a particular `RecoverableBlockImpl` class.

*RecoveryBlock class.* This class monitors the execution of a recovery block. It contains the method `execute()` which is responsible for executing the processing steps of a recovery block, previously mentioned. Figure 3 shows this class. An application server first instantiates an object of this class giving the appropriate `RecoverableBlockImpl` object as a parameter, and then invokes the `execute()` method to activate the corresponding recovery block.
public interface RecoverableBlock {
    public int getNbAlternates();
    public Object invokeAlternate(int i, Object[] params) throws AlternateExecutionFailureException;
    public boolean isAcceptable(Object[] params, Object result);
    public void saveCheckpoint();
    public void restoreCheckpoint();
}

Figure 2: The RecoverableBlock Interface

public class RecoveryBlock {
    private RecoverableBlock recoverableBlock;
    
    public RecoveryBlock(RecoverableBlock block) { // constructor
        this.recoverableBlock = block;
    }
    
    public Object execute(Object[] params) throws RecoveryBlockFailureException {
        Object result;
        int i = 0;
        this.recoverableBlock.saveCheckpoint();
        while (i < this.recoverableBlock.getNbAlternates()) {
            try {
                result = this.recoverableBlock.invokeAlternate(i, params);
                if (this.recoverableBlock.isAcceptable(params, result))
                    return result;
            }
            catch (AlternateExecutionFailureException e) {
                System.out.println("RecoveryBlock: " + e);
                this.recoverableBlock.restoreCheckpoint();
            } finally {
                i++;
            }
        }
        throw new RecoveryBlockFailureException();
    }
}

Figure 3: The RecoveryBlock Class

In the case of using the $N$-version approach, it is necessary to define: 1) the redundant modules (versions) that implement the application server; 2) the validation mechanism to be used (majority voting, inexact voting or comparison); 3) a procedure that is responsible for executing all redundant modules according with the $N$-version approach; and 4) a mechanism responsible for raising an exception when the validation mechanism is comparison and there are faulty versions. The execution of an $N$-version replicated server proceeds as follows (considering comparison as the validation mechanism):

1. Invoke each of the modules which are executed in parallel
2. Wait for all modules to complete their execution
3. Submit outputs produced to the validation function

4. If the validation succeeds then:

   finish the \( N \)-version block operation returning the output produced by any one of the modules

else:

   stop the \( N \)-version block operation (failure semantics is by crashing)
   raise an exception

The following classes and interfaces can be used to implement this service.

**Exception classes.** Two classes are required. One to deal with failures of the \( N \)-version block as a whole (when the validation strategy is comparison and there are faulty modules) - let this class be named `NVersionModuleFailureException`, and another to cope with individual failures of redundant modules - let this class be named `ModuleExecutionFailureException`. These classes can also be implemented as extensions of the `RuntimeException`.

**NVersionedModule interface.** It contains the declaration of methods that are responsible for implementing the requirements listed in items 2, 3 and 4, presented before (see Figure 4).

```java
public interface NVersionedModule {
    public int getNbModules();
    public Object invokeModule(int i, Object[] params) throws ModuleExecutionFailureException;
    public Object validate(Object[] results) throws NVersionModuleFailureException;
}
```

**Figure 4:** The `NVersionedModule` Interface

**NVersionedModuleImpl class.** It implements the interface `NVersionedModule`, including the methods that are responsible for implementing all versions of an application server.

**NVersionModule class.** It contains the method `execute()` which is responsible for monitoring the execution of an application server implemented as a \( N \)-version block. This class is presented in Figure 5.

It is important to emphasize that the application programmer must implement the methods defined in the interfaces `RecoverableBlock` and `NVersionedModule`, as well as, the methods related to the redundant modules of a recovery block or a \( N \)-version block. All the other methods have already been implemented and require no intervention from the programmer.

### 2.4.2 Providing the Mechanism for Processor Fault Tolerance

Following an OO strategy and making good use of the facilities provided by the Java language, it is possible to implement an active replication scheme based on Java Remote Method Invocation (RMI). Java-RMI [9] is one of the most popular object technologies used to develop distributed applications.

An RMI application consists of two separate programs: a client and a server. A typical server application creates a number of remote objects\(^2\), makes references to these remote objects accessible (registering such

---

\(^2\)A remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host. An object of this type is described by one or more remote interfaces which are Java interfaces that declare the methods of the remote object.
public class NVersionModule {
    protected NVersionedModule nVersionedModule;
    public NVersionModule(NVersionedModule module) {
        this.nVersionedModule = module;
    }
    public Object execute(Object[] params) throws NversionModuleFailureException {
        int nVersions = this.nVersionedModule.getNbModules();
        Object[] results = new Object[nVersions];
        ThreadGroup group = new ThreadGroup("theGroupOfReplicas");
        for (int i = 0; i < nVersions; i++) {
            try {
                (new Thread(group, new RunnableModule(this.nVersionedModule, i, params, results))).start();
            } catch (Exception e) {
                results[i] = null;
            }
        }
        while (group.activeCount() != 0); // Espera até que todos as threads disparadas terminem suas execuções
        return this.nVersionedModule.validate(results);
    }
}

private class RunnableModule implements Runnable {
    private NVersionedModule myModule;
    private int myNb;
    private Object[] params;
    private Object[] results;
    // Constructor
    public RunnableModule(NVersionedModule module, int moduleNb, Object[] params, Object[] results) {
        this.myModule = module;
        this.myNb = moduleNb;
        this.results = results;
        this.params = params;
    }
    public void run() { // It executes all threads created
        try {
            this.results[this.myNb] = this.myModule.invokeModule(this.myNb, this.params);
        } catch (ModuleExecutionFailureException e) {
            System.out.println("ParallelNVersionModule: "+ e);
            this.results[this.myNb] = null;
        }
    }
}

Figure 5: The NVersionModule Class

references with a name server, named RMI registry) and waits for clients to invoke methods on those remote objects. RMI provides mechanisms by which the server and client communicate and pass information back and forth, they are the following: stubs and skeletons. All these facilities are provided by the Java RMI package (java.rmi) that is a part of the Java API.
In this context, it is possible to provide fault tolerance by replicating the server objects (remote objects) and distributing the replicas over different machines on the network (hosts) where they will execute in parallel following the active replication strategy. In such case, it is necessary to satisfy some requirements: 1) the method invocation on remote replicated objects must be received by the skeleton associated with each replica of the remote object; 2) invoked at the same order in every replica of the remote object; 3) duplicated results must be filtered by the stub that has invoked the remote method, thus, the replicated processing can be transparent on the client side; and 4) all the replicas of a replicated object must be registered under the same name with the name-server. Items 1 and 2 are necessary in order to guarantee the two requirements to implement active replication schemes: agreement and order, respectively.

In [10] it is proposed a Java package, named FilterFresh, for building replicated fault-tolerant servers. Managing of replicated servers is supported by a Group Manager object instantiated with each replica to form a logical group. The Group Managers use a Group Membership algorithm to maintain a consistent group view and a Reliable Multicast mechanism to communicate with others Group Managers. It is important to notice that most of the classes and interfaces available in the FilterFresh package are extensions of classes available in the java.rmi package.

The FilterFresh package was used to construct a fault-tolerant RMI registry, named FTRegistry, that allows, among others facilities, the registration of multiple servers under the same name. Further, the classes available in this package can also be applied in order to implement a client-side mechanism that masks server failure in a transparent manner to the client. Such a mechanism, named FTUnicast, follows the passive replication strategy. This strategy can be easily extended to allow the implementation of replicated remote objects through active replication.

3 A Comparative Study of the Services

In this section we make a comparative study on the probabilities of the fault-tolerant processing services proposed being correct and operational. The objective is to identify the main aspects that influence the reliability and availability level of the services. This simplified analysis provides information that can be used to estimate the costs of using each service in respect to a specific dependability requirement. Knowing the costs associated with each processing service the application programmer will have additional elements to choose the most appropriate service.

We denote the probability of a service being correct, i.e. behaving in accordance with its specification (failure and operational semantics), by $PC$. Likewise, we denote by $PO$, the probability of the service being operational, i.e. given that the service is correct ($PC = 1$), $PO$ gives the probability of the service presenting its operational semantics.

$PC$ and $PO$ values are influenced by the probabilities associated with the assumptions made on the behavior of the processors and software fault tolerance mechanisms considered, as well as, the redundancy level being used, i.e. the number of processors ($N_{hp}$) and the number of redundant modules or versions ($N_{sw}$) being used to implement the respective hardware and software fault tolerance mechanisms. Due to space limitations, in this paper we only present a study of $PC$ and $PO$ for the fault-tolerant processing service based on the recovery block approach. The probabilities $PC$ and $PO$ corresponding to the fault-tolerant processing services based on $N$-version programming are presented in [11].

We have assumed that processors fail by crashing, however it is possible (although very unlikely) that a processor presents a different failure semantics. If a processor can fail with a failure semantics different from that assumed, so can the processing service provided by this processor. Thus, the value of $PC$ will never be greater than the probability of a processor fail by crashing ($P_{fc}$). Considering the software fault tolerance mechanism, its correctness depends on the reliability of the acceptance test associated with the recovery block. Thus, the probability of such mechanism being correct is represented by the probability of the acceptance test being reliable ($P_{at}$).

Therefore, $PC$ can be expressed as follows:

$$PC = P_{fc} \times P_{at}.$$
Since the probability of off-the-shelf processors fail only by crashing is very high (\(P_c \approx 1\)), the value of \(PC\) will increase as the value of \(PC_{at}\) increases, i.e. as the acceptance test becomes more reliable. One way of doing that is by simplifying the acceptance test, such that, it is responsible for checking just the minimal required conditions for validating the outputs produced during the recovery block operation. In fact, this strategy does not come for free. Simplifying the acceptance test implies in decreasing its level of rigor; this in turn, can be unacceptable for certain applications.

Considering that the fault-tolerant processing service based on recovery blocks is correct, for this service to be operational it is necessary to have, at least, one non-faulty processor among the replicated group of processors on top of which the service executes. Further, the mechanism used to tolerate software faults must be operational. The probabilities of such situations to occur are represented by \(PO_{hw}\) and \(PO_{sw}\), respectively.

Assuming that the probability of a processor to fail is given by \(F_p\), then, the probability of having at least one processor behaving in accordance with its operational semantics is given by:

\[
PO_{hw} = 1 - (F_p)^{N_{hw}}.
\]

On the other hand, the probability \(PO_{sw}\) associated with the mechanism that implements the recovery block approach is given by:

\[
PO_{sw} = 1 - \prod_{i=1}^{N_{sw}} F_{r(i)}^{br(i)},
\]

where \(F_{r(i)}\) is the probability of the \(i\)-th module of a recovery block to contain a design fault. Therefore, the probability of the fault tolerant processing service based on recovery blocks be operational can be expressed as follows:

\[
PO = (1 - (F_p)^{N_{hw}}) \times (1 - \prod_{i=1}^{N_{sw}} F_{r(i)}^{br(i)}).
\]

Considering the value of \(PO\), we observe that the higher is the value of \(N_{sw}\), the higher is the probability of having, at least, one non-faulty module within the recovery block. Further, \(PO_{hw}\) increases with the increase on \(N_{hw}\). Thus, the value of \(PO\) can be increased by increasing not only the number of redundant modules of each block, but also the number of processors over which the replicas of a block execute.

Note that, the values of \(PC\) and \(PO\) represent, respectively, the assurance level on the failure semantics and the reliability/availability level of each processing service. By increasing such values it is possible to provide processing services with any required level of dependability.

4 Validating the Services
The fault-tolerant processing services being discussed in this paper are due to be used by applications with some level of criticality, thus, it is necessary to guarantee their correctness.

The mechanisms for software fault tolerance can be validated by assuring the correct execution of the recovery blocks and \(N\)-version blocks which implement application servers, i.e. by assuring the correctness of the \(execute()\) methods defined in both the \(RecoveryBlock\) and the \(NVersionModule\) classes. Note that, both \(execute()\) methods are simple, and do not require inter-process communication (they are not distributed). Therefore, they are much less prone to contain design faults. Thus, the validation of the \(execute()\) methods is straightforward.

The mechanism for hardware fault tolerance can be validated by assuring the correct execution of the replicated processing based on the active replication strategy, i.e. by assuring the correctness of the replica management protocols required to implement an active replication scheme. In the case of the services presented, since the failure semantics of a recovery block or a \(N\)-version block, as well as, the failure semantics of a processor is by crashing, the only replica management protocol required is a distributed ordering protocol. Note that the distributed characteristics of the ordering protocol renders it more complex, therefore, validating such a protocol is not as easy as validating the \(execute()\) methods.
To facilitate the task of validating a fault-tolerant distributed protocol, such as an ordering protocol, it is common to use well known system models to describe the implementation system over which the protocol will execute. Theoreticians have proposed a number of system models that can be used to describe distributed systems. These models are normally differentiated from each other by the restrictions that they impose on two key characteristics of the model, namely its synchronism guarantees and the failure semantics of its components. Based on these models, generic solutions of important problems in the context of fault-tolerant distributed systems, such as distributed consensus, atomic broadcast, group membership, leader election, atomic commitment, etc, have been proposed. Practitioners can then use these results to facilitate their job in building reliable distributed systems.

An off-the-shelf distributed system is characterised by the absence of upper bounds on both message transmission and scheduling delays, i.e. it is an asynchronous system. Unfortunately, an impossibility result presented in [3] makes the task of designing fault-tolerant distributed protocols difficult in asynchronous systems. For this reason, a number of models that strengthen the asynchronous distributed system model, allowing the solution of some particular problems (e.g. agreement problems), have been defined. Among them, the asynchronous distributed system model augmented by an unreliable failure detector (A+UFD, for short) [4] and the timed asynchronous distributed system model (TA, for short) [5] have been widely discussed.

The following observation is the core of the impossibility result presented in [3]: due to the uncertainty on communication and scheduling delays, it is impossible to differentiate a processor that has failed from one that is simply slow. From the above observation it is possible to establish a cause-effect relationship between the absence of bounds in the communication and scheduling delays and the impossibility of detecting crashed processors. The A+UFD and TA models have followed different approaches to circumvent this impossibility result.

The A+UFD model tackles the effect, i.e. it assumes the existence of an “oracle”, named a failure detector, that is able to give an idea of which processors have crashed. Although the information provided by this oracle may be unreliable, it is precise enough to allow deterministic solutions for a number of agreement problems. However, if for one side, the unreliable failure detector is a powerful abstraction that simplifies the design of distributed protocols, on the other side it is impossible to build such a device within an asynchronous system (otherwise the result in [3] would not hold). Thus, a practical solution will require additional assumptions.

On the other hand, the TA model tackles the cause, i.e. it establishes unreliable bounds on communication and scheduling delays. The model is based on the observation that although it is not possible to define bounds that are always respected, for many systems it is feasible to define bounds that will be respected most of the time; further, in practice, these systems alternate between long periods of stability - during which communication and scheduling delays are always below the assumed bounds - and short periods of instability - during which these delays may be larger than their respective bounds. Since during a period of stability the system behaves like a synchronous system, deterministic solutions for agreement problems can be implemented and will work accordingly provided that the system is stable. The unreliable bounds are used to detect performance failures during periods of instability and allows the provision of a “safe” execution mode for the distributed protocols. The model also introduces the notion of a progress assumption that provides a measure of the probability of the system being stable in a given instant of time. The TA model is well-suited to describe most off-the-shelf distributed systems currently available. Moreover, the notion of time introduced by the unreliable upper bounds allows the specification of timed services that cannot be specified in other asynchronous system models. The main drawback of the TA model is that validating the design of complex distributed protocols is normally much more difficult. Further, the specification of the problem needs to be changed (to incorporate a “safe” operation mode when the system is unstable) and suitable progress assumptions must be defined.

To validate the ordering protocol required, we propose a hybrid approach. We use the TA model to describe the asynchronous distributed system, but instead of designing and validating the ordering protocol

---

3 The synchronism guarantees of a model define the bounds that might exist on communication and task scheduling delays, while the failure semantics of the components describes the number and the type of faults that might affect processors and the communication infra-structure.
considering this model, we first identify abstractions that can be used to facilitate the design and the validation of such protocol (e.g. unreliable failure detectors, fault-tolerant communication services, etc). Then, we design simpler protocols to provide these services and validate them using the TA model. Finally, we use the TA model augmented with the extra services (already validated) to design a (higher level) ordering protocol that is simpler and easier to validate.

4.1 Applying a Hybrid Approach to Validate an Ordering Protocol

To achieve ordering of input messages in a group of replicated processors, it suffices that every message received by a non-faulty processor is atomically broadcasted to all other processors of the group [12]. Therefore, it is possible to solve the ordering problem from a solution to the atomic broadcast problem, for instance by using the atomic broadcast protocol presented in [4].

The protocol presented in [4] was designed considering the A+UFD model and requires a reliable broadcast service and a consensus service that uses a failure detector of class $\oplus S$. Applying the strategy discussed above we design and validate the ordering problem as follows: first we assume the TA model to design and validate a reliable broadcast service and a failure detection service (at least as strong as a $\oplus S$ failure detector); then, by assuming the extended TA model, we can use the same protocols and proofs presented in [4] to solve the ordering problem.

The extra services required are characterised as follows.

Properties of the Reliable Broadcast Service:

- **Integrity**: every message $m$ delivered by a correct process was previously broadcasted by some process and is delivered at most once;
- **Agreement**: every message $m$ delivered by a correct process is also delivered by all correct processes;
- **Validity**: every message $m$ broadcasted by a correct process is delivered by all correct processes; and
- **Termination**: a message $m$ broadcasted by a correct process at real time $t$, is delivered by all correct processes at latest by $t + \Sigma$.

Properties of the Failure Detection Service:

- **Completeness**: if a process crashes at real-time $t_c$, then its failure is detected by all correct processes by real-time $t_c + \Omega$; and
- **Accuracy**: for every instant of real-time $t$, there is a real-time $t', t' > t$, after which a majority of correct processes will not suspect a correct process $p$ for at least $\Theta$ units of time and $t' - t < \Gamma$.

4.1.1 A Reliable Broadcast Service

This service is implemented by a very simple protocol\(^5\). It has two functionalities, one responsible for message diffusion, i.e. the broadcast and the delivery of messages by the group of processes, and the other taking charge of recovering broadcasts initiated by processes that fail.

The service is implemented by Protocol 1 which is divided into five modules (executed in parallel) and has one access primitive, described below:

\(^4\)In fact, these properties are closer to those provided by a failure detector of class $B$ [13] which are stronger than those provided by a failure detector of class $\oplus S$.

\(^5\)Other solutions, possibly more efficient, may exist. Our purpose is not to focus on the efficiency of the extra services, but on the strategy used to validate higher level protocols.
Protocol 1 Reliable Broadcast Protocol Executed by Process \( p \)

/* Global Variables */
Delivered = {} 
NotDeliveredId = {} 

broadcast(m):
send \( m \) \( \forall p_j (j \neq i) \)
Delivered = Delivered \( \cup \{ m \} \)
execute the corresponding application call-back to deliver \( m \) 

|| Module: DeliverMsg
when receive message \( m \)
Delivered = Delivered \( \cup \{ m \} \)
execute the corresponding application call-back to deliver \( m \) 
end when 

|| Module: SendState
while TRUE do
\( m = (i, \{ \{ m.id \mid m \in Delivered \} \}) \)
send \( m \) \( \forall p_j (j \neq i) \)
waituntil \( \delta_{sendState} \) expires 
end while 

|| Module: ReceiveState
while TRUE do
receive \( s = (sender, Ids) \)
MyDeliveredId = \( \{ m.id \mid m \in Delivered \} \)
for all \( id \in s.Ids \mid id \notin MyDeliveredId \) do 
NotDeliveredId = NotDeliveredId \( \cup \{ (s.sender, id) \} \)
end for 
end while 

|| Module: RecoverMsg
while TRUE do
while NotDeliveredId \( \neq \{ \} \) do 
for all \( (p, id) \in NotDeliveredId \) do
request message \( m | m.id = id \) from \( p \)
end for 
while \( \delta_{retransmission} \) does not expire do 
receive \( m \)
RepliedReq = \( \{ r = (sender, id) \mid r \in NotDeliveredId \wedge r.id = m.id \} \)
NotDeliveredId = NotDeliveredId \( \setminus \{ r \} \)
Delivered = Delivered \( \cup \{ m \} \)
execute the corresponding application call-back to deliver \( m \)
end while 
end while 

|| Module: RetransmitMsg
while TRUE do
receive request \( r = (sender, id) \)
send \( m | m.id = r.id \wedge m \in Delivered \) to \( r.sender \)
end while
Primitive broadcast\((m)\). It starts the reliable broadcast of a message \(m\). Broadcasted messages are delivered to the local application by an invocation to a call-back function that has been registered by the corresponding application. Delivered messages are added to the set \(Delivered\). The identifiers of the messages belonging to the set \(Delivered\) represent the internal state of a process and are periodically diffused to allow recovery of missed messages.

Module for message delivery (DeliverMsg). This module is responsible for the reception and delivery of every message broadcasted by some process in the group.

Module for sending the identifiers of delivered messages (SendState). This module is responsible for, periodically, diffusing (in an unreliable manner) the internal state of each process, i.e. the message identifier of every message delivered by each process. Such information is encapsulated in a message that is sent to all processes every \(\delta_{sendState}\) time units.

Module for receiving the identifiers of delivered messages (ReceiveState). This module is responsible for receiving the internal state of all other processes, i.e. the identifiers of all messages delivered by all other processes. The received information is used to update the contents of the set \(NotDeliveredId\).

Module for message recovery (RecoverMsg). This module is responsible for requesting every message whose identifier belongs to \(NotDeliveredId\). The elements of the set \(NotDeliveredId\) are tuples \((sender, id)\), where \(id\) is the identifier of a message that was not yet received by \(p\) and \(sender\) is the identifier of a process that has already delivered this message. The retransmission of such message is requested to all processes that have already delivered the message. The retransmission of a message is requested every \(\delta_{retransmission}\) units of time, until it is received and delivered locally. After having received requested messages, the contents of the sets \(NotDeliveredId\) and \(Delivered\) are updated accordingly.

Module for message retransmission (RetransmitMsg). This module is responsible for retransmitting missed messages. As soon as a request to retransmit a message \(m\) is received, the corresponding message is sent to the requesting process.

4.1.2 A Failure Detection Service

This service is implemented by a protocol that periodically diffuses \(IamAlive\) messages to the other processes and checks the emission of \(IamAlive\) messages of the other processes.

The service is implemented by Protocol 2, which is divided into three modules (executed in parallel) and has one access primitive, as described below:

Module for sending \(IamAlive\) messages (SendIamAliveMsg). This module is responsible for the diffusion (in an unreliable manner) of \(IamAlive\) messages to the group of processes. Such a diffusion is started every \(\delta_{heartBeat}\) time units.

Module for failure detection (DetectFailure). This module is responsible for detecting crashed processes.

Every \(\delta_{detection}\) time units it checks if there is some process that should be added to the set \(ListOfSuspects\). A process \(p_j\) is suspected by another process \(p_k\) when \(p_j\) spends more than \(\delta_{failure}\) time units without sending \(IamAlive\) messages to \(p_k\). In other words, the elapsed time since the last time process \(p_j\) received an \(IamAlive\) message from \(p_j\) (represented by the value of \(\delta_{exception}[p_j]\)) is larger than \(\delta_{failure}\) time units. In each failure detection round the set \(NewListOfSuspects\) is formed and the contents of the set \(ListOfSuspects\) is updated.

Module for receiving \(IamAlive\) messages (ReceiveIamAliveMsg). When an \(IamAlive\) message is received from a process \(p_j\), the array \(\delta_{exception}\) is updated, at position \(p_j\), with the local clock time. This array stores, for each process, the time when the last \(IamAlive\) message was received from the respective process.
Primitive `suspects()` returns the set `ListOfSuspects` which is composed by the processes suspected of having crashed.

### Protocol 2 Failure Detection Protocol Executed by Process $p_i$

/* Global variables */
`ListOfSuspects` = {}

/* Initialisation of local variables */

for all $p_j, j \neq i$ do
    $\delta_{reception}[p_j] = time_{local\_physical\_Clock}$
end for

|| Module: `SendIamAliveMsg`
while TRUE do
    $m = (p_i, IamAlive)$
    broadcast($m$) \forall $p_j (j \neq i)$
    waituntil $\delta_{heartbeat\_expires}$
end while

|| Module: `DetectFailure`
while TRUE do
    waituntil $\delta_{detection\_expires}$
    `NewListOfSuspects` = {}
    for all $p_j, j \neq i$ do
        if $\delta_{reception}[p_j] < time_{local\_physical\_Clock} - \delta_{failure}$ then
            `NewListOfSuspects` = `NewListOfSuspects` + $\{p_j\}$
        end if
    end for
    `ListOfSuspects` = `NewListOfSuspects`
end while

|| Module: `ReceiveIamAliveMsg`
when receive message $m = (sender, IamAlive)$

    $\delta_{reception}[m, sender] = time_{local\_physical\_Clock}$
end when

`suspects()`:
return `ListOfSuspects`

Due to space limitations we do not present the proofs for the two protocols presented above. These proofs can be found in [14].

### 5 Conclusion

In this paper we have presented the design of fault-tolerant processing services for asynchronous distributed systems. These services either mask hardware and software failures or fail just by crashing. We have performed a comparative study on the level of correctness and promptitude of the fault-tolerant processing services defined. From this effort it was possible to identify the main aspects that influence the reliability and availability level of each service. Although we did not provide a detailed quantitative analysis, the information provided by the values of $PC$ and $PO$ are adequate for helping a developer to decide which service to use to fulfill its needs.

To validate the services we have proposed a hybrid approach that combines two classical asynchronous model: the timed asynchronous model and the asynchronous model augmented with unreliable failure detectors. To validate the ordering protocol required, we used an extended timed asynchronous model, augmented
with a failure detection service and a reliable broadcast service. This allowed the utilisation of known protocols to solve the ordering problem. It is important to notice that these protocols were validated assuming a system model that considers existence of services that are not normally provided by off-the-shelf distributed systems. Thus, we have presented protocols that can be implemented on top of such systems and that provide the extra functionality assumed.

References


