An Overview of Fault-tolerance using Software Replication Techniques

Simin Nadjm-Tehrani and Diana Szentivanyi
Department of Computer and Information Science
Linköping University
[simin/diasz@ida.liu.se]

March 31, 2001

1 Introduction

The revolutionary growth of network-based technologies in the 90's has had a huge impact on the growth of application domains in which distributed computing is a basic premise. These applications are to be found all over the spectrum from information systems, data management and retrieval, communication networks, and computer-based systems for command and control. Distribution of functions combined with added levels of functionality is now an emerging trend in the automotive and aerospace domains as well as air traffic control. In addition to this, distribution as a means of achieving fault-tolerance is prevalent – in some cases where safety is a major concern, e.g. aerospace, or in others where availability is a prime requirement, e.g. telecommunications.

In this paper we review the latest developments in the area of distributed systems with respect to achieving fault-tolerance, and in particular via replication techniques. Since the area of fault-tolerance includes several diverse areas, the report should not be seen as a comprehensive survey about fault-tolerance but a reflection over the state-of-the-art in one area, presented in the larger context of fault-tolerance.

The paper is organised as follows. Section 2 deals with basic terminology and some historical notes. More details about models and terms in a network setting, including various fault-models and ways to treat them can be found in section 3. Section 4 is devoted to consensus as a basic mechanism to achieve fault-tolerance, and section 5 describes an abstraction higher up in the software infrastructure: the group abstraction. In section 6 we review the ways of measuring efficiency. Section 7 relates the notions used in process replication with similar notions in the database area with distributed transaction as a basic mechanism. In section 8 we look at ways of adding fault-tolerance to the middleware in a distributed setting. The discussion here is at a more detailed level and implementation oriented style, as opposed to the earlier sections

where we simply provide an overview. Section 9 begins to wrap up the paper, it describes the only general formalisation we are aware of – a framework for relating fault-tolerant and fault-intolerant computational processes. Section 10 describes some emerging trends and observations, and section 11 concludes the report.

2 Models for fault-tolerance

In this section we present some basic notions, and some earlier attempts to unify the approaches to fault-tolerance by defining restrictions on distributed computational systems.

Fault-tolerance increases the capacity of a system to continue providing its services at the same quality, or at some reduced (acceptable) quality even in presence of faults. Since the area was large and diverse, there was little unifying work to formalize the basic notions. This unfortunately resulted in authors not systematically building on earlier agreed common grounds for a while[1]. An early attempt at defining common agreed upon terms appeared in the work by Laprie [32], which was later on adopted and developed as the working document for the IFIP WG 10.4 on dependable computing.

2.1 A historical perspective

One of the early attempts at distinguishing models for fault-tolerance appeared in the works of Cristian [11]. However, it was not until late 90's that a general framework for formal treatment of fault-tolerance appeared. Prior to that, formal descriptions were common but were typically used for specification and analysis of particular algorithms or protocols. A general framework for developing systems including systematic addition of fault-tolerance to fault-intolerant systems is a recent contribution by Kulkarni et.al., and a survey by Gärtner is a valuable source of formally expressed models and approaches [2, 24].

Some authors see the requirement for fault-tolerance as a part of the specification of the system, i.e. derived from availability, safety, integrity, etc. [34, 3]. Others see distribution itself as a motivation for fault tolerance [41]. However, by and large, one can say that early works on fault-tolerance either did not have distribution as a basic premise or had strong synchrony requirements on a system's components. The distinguishing factor between various works is often the emphasis on hardware or software implementations [43, 35, 48, 11, 5]. and whether the technique is employed in a particular setting, e.g. databases [6] or disk storage [23].

In most cases, however, multiple versions of a process(or) are added to achieve fault-tolerance. The same thing is true for data; multiple copies of a data item are used to provide masking in case of failures. Another possibility is to augment data items with redundant information (for example parity bits

[1] We see for example that basic notions such as fail-safe (crash) and fail-stop are used in one way in some papers [49, 48], and the opposite way in others [19].

and CRC code in data transmission). Thus, for both types of "multiplication" we talk about redundancy, but in the process(or) case we even use the term *replication*.

Thus, in replicated systems we have several copies of some entities contributing to the production of the final result, which should be unique. In this context we introduce the notion of *consensus*, which relates to the need of multiple processing units to "agree" or "reach consensus" on the produced result. This notion has generated a lot of work in the distributed systems community, devising algorithms for solving the problem.

An important early result which possibly affected the future directions considerably is the theoretical result on impossibility of consensus in *asynchronous systems* in presence of failures [20] – a theorem established by Fischer and Lynch in the mid-80's. In a synchronous system every system component takes one step whenever other system components take $n(n \geq 1)$ steps. In asynchronous systems this is no longer the case. Processing units can execute a step without any relation to other nodes' activity. Also, there is no known upper bound on the delay in communication amongst units. Thus, no reference to the notion of global time can be made in an asynchronous system.

There has of course been a large body of work on clock synchronization algorithms over the years (for historical snapshots see [49, 51]). However, the impossibility result tells us that since the asynchronous distributed system has no global notion of time, there is no deterministic algorithm to distinguish between a failed process and an extremely slow process. Hence a failure detector and thereby fault-tolerance measures can not be used to compensate for failures.

If the system, on the other hand, can be equipped with a global clock to which the nodes have access, then we make a system synchronous. Such systems together with associated protocols and fault-tolerance mechanisms have been extensively studied by Kopetz et.al. and described under the general name of time-triggered systems [29]. Almost all papers covered by this survey concentrate on asynchronous systems.

2.2 Recognizing faults

Any system which treats faults in a foreseeable manner has to be able to recognize the symptoms of faults leading to failures at subsystem level.

Failure models

A failure model shows the way a system can fail as perceived from other systems using its services [12]. Before a failure occurs, there is a chain of events. First, a fault (i.e. a defect at a low level) has to exist or appear in one part of the system. This fault *can* lead to an error, if the part of the system is activated under certain conditions. An error is, thus, the manifestation of a fault. We can say that if an error happens the system is in an erroneous state.

Finally, if the error is not "treated", it can lead to a failure. The treatment can be repairing a broken part, switching to a redundant unit, or jumping to an

exception handler in a piece of software. A failure can be seen as the deviation of the behavior of the system from the specified one. Of course, the chain ... - fault - error - failure - ... can be quite long. A fault in a higher level of a system can, in its turn, be caused by a failure in a lower level.

Some classifications of faults and failures are given in the literature [17]. Considering their occurance in time, faults can be:

- transient: such a fault appears in the system for a short while and then it "goes away" - it can cause, of course, an error. An example could be a fault caused by a radiation beam hitting a hardware circuit.
- intermittent: such a fault appears at certain intervals for a short while and repeatedly "goes away"; example: system overload faults, or a loose wire.
- permanent: such a fault is permanently there in the system; the question is when will it cause the error. An example of such a fault is a design fault (e.g. a software bug).

Failures are also classified in the literature, in terms of their aftermath, or the their severity [17]. The class of failures covered by a particular algorithm is called the failure model for it.

A common model is the crash failure model. Failing by crash means that the unit stops executing its computation and communication, and does nothing more. Another name for the crash failure model is the fail-silent model.

A more serious type of failure is the Byzantine failure, where the unintended behaviour of a unit can be arbitrary, and potentially caused by malicious attacks.

Of course, when several processing units collaborate to provide fault-tolerant services, the communication media among these units is also important to analyze. Thus, in a distributed fault-tolerant "service provider", there are two major types of failures. This classification is based on the location of the failure. In addition to the previously mentioned processor failure, we have the class of network failures. These can be due to a broken communication wire, or to congestion of a communication link. In both cases, the two units connected by the "failed" link cannot communicate. In this case, we say that the two nodes are in different *network partitions*, unless there is some other indirect way for them to communicate. Thus, a new failure model, concerning the communication media, is the *partition failure*.

Unreliable failure detectors

If the time-free asynchronous system model is used, in presence of failures, we saw that there is no possibility of reaching agreement in bounded time (not bounded also means infinite).

However, there is a possibility to weaken the time-free assumption, without departing too much from a real setting of a distributed system. What if one simply supposes that there is some bound on message delay, but at the same time is conscious that this supposition can be mistaken?

This aspect is explored by Chandra and Toeug in their formal characterization of failure detectors [10]. They actually play around with "mistake patterns" which are their way of specifying the so-called *unreliable failure detectors*.

For example, when executing some consensus algorithm, processors do not explicitly reason about time – time is simply abstracted away from the algorithm execution level. Instead, they query their failure detectors about the state of the other members of the replicated group.

The failure detectors can make mistakes, by suspecting processors which are correct[2], to be faulty – the so called lack of *accuracy*, or by not detecting an actual failure – the so called lack of *completeness*.

Even though strong accuracy, i.e. the property that a correct processor is never suspected by any correct processor, is difficult to achieve, some weak accuracy or eventual accuracy properties can be satisfied. The same is true about completeness (eventually all processor failures are detected). Strong completeness (all correct processors will eventually suspect all failed processors) is difficult to achieve in an asynchronous setting. However, Chandra and Toeug show that for correctness of some distributed algorithms it is enough that the failure detectors have some weak properties. Moreover, they provide transformation algorithms to make some weak detectors more strong.

The main contribution of their work is to show that the impossibility result by Fischer et al. can be overcome if the failure detectors have enough "reliability" for performing the algorithm. As an example they give a reliable broadcast algorithm and show that it can be used to solve consensus using unreliable failure detectors of a particular class (weakly accurate and strongly complete). We will come back to this in section 4.

3 FT network services: basic notions

Using replication (redundancy) to achieve fault-tolerance is an old trick[3]. However, several non-trivial problems remain to be solved in practical engineering of fault-tolerance by replication. In this section we describe some basic notions necessary for building up replicated services. This will cover replication in *space*; fault-tolerance via redundancy in *time* (e.g. repeating an execution after a transient failure) will not be covered here.

In the following sections we often use two words: process and processor. When talking about fault-tolerance by software replication, they can be used, in general, interchangeably. However, in some cases, one or the other term is more appropriate. In order to give a better understanding of the issues discussed, we use "process" when we only mean computational processes.

[2] i.e. those that have not crashed

[3] An early reference to the topic goes back to 1824 and is cited by Shin and Krishna [30].

3.1 Models of FT networks

Algorithms and applications for distributed systems require a model of the underlying system. The model is built by making assumptions about the physical system consisting of processing units and communication media.

We have already seen the main assumptions of synchrony and asynchrony in relation to time. A third model is based on *partial synchrony*. A partially synchronous system model is one in which there are intervals during the system's life-time when there exist bounds on message delays and on processor execution times. This is the general idea but there are variants to this notion, also referred to as *partially stable*, in which the bounds are not known [16]. Actually, Dwork et al. delimit very precisely two partial synchrony models: when the bound on message delays and/or processor synchrony hold only after a "global stabilization time" (GST), and when the same bounds exist and hold all the time, but are not known. Further, under a certain time-related assumption, different kinds of failure models exist. These include network failures as well as process(or) failures.

Network failures

When considering the communication media, two types of failure models can be considered. One is the *partition* failure model. A partition means that it is possible for the system to split in two (or more) disjoint sets of processes working separately, and each set believing that there are no other working processes in the system. For example, the algorithms in [4] are built starting from this assumption.

On the opposite side we have the *no-partition* assumption, as defined by Ricciardi et al., which means the reverse: there is no such situation when there are at least two disjoint sets of processors in the system, such that members of these sets "believe" that they are the only alive nodes [45]. As a consequence, if there always exists at least one correct processor which is not suspected by other correct processors, then the no-partition assumption holds.

Similar to the no-partition assumption is the *primary-partition* notion. This means that there will always be a majority of correct processors, and progress will be allowed *only* if this majority can participate in decision making.

Processor failures

In connection with processors, the most usual assumption is that one can fail by crashing. Other failures such as commission or Byzantine failures, when processors fail in an arbitrary way are harder to address. In presence of failure detectors, a processor can be suspected to have crashed. Further assumptions can be made about a processor's behaviour after an actual crash, or the reversal of a suspicion.

First, we can have the *recovery* assumption. Usually, recovery is related to a real crash. It simply means that an actually crashed processor is repaired after a while, and joins further decision making steps in one way or another (for

example in algorithm of [1]). Sometimes, the recovery assumption is related with a suspected crash. In this case recovery means that the suspicion can be revoked provided that the processor is found to be correct later on (this assumption is used in [10]).

The opposite model is based on the *no-recovery* assumption. Again, we can relate the assumption to a real crash, in which case we assume that the processor is not repaired and will never participate again in decisions. On the other hand, when no-recovery is related to a crash suspicion, it means that once a processor is suspected, the suspicion is never revoked. This notion is called *failure belief stability* by Ricciardi et al. [45]. The no-recovery assumption is used in both senses in [28].

A failure model is built by combining these assumptions. For example, Babaoglu et al. use a failure model built on crash/no-recovery/partition assumptions [4]. The no-recovery is used in its first sense (related to real crashes, i.e. it is possible to revoke a crash suspicion).

As a summary of the above mentioned models we can build the following table. On the left we list the mentioned papers and in the table we put for each, the assumed failure and network model. Some models do not make sense in the context of the paper considered. For these we will use N/S. For the others, we will use –.

	recovery		
	from crash	suspicion revocation	partition
paper [28]	No	No	No
N/S	No	No	Yes
paper [10]	No	Yes	No
paper [4]	No	Yes	Yes
N/S	Yes	No	No
N/S	Yes	No	Yes
paper [1]	Yes	Yes	No
—	Yes	Yes	Yes

Table 1: Summary of models used in major approaches to group replication

In section 4, we show how the above assumptions affect some of the algorithms.

3.2 Communication in presence of replicas

Because in a replicated reliable service framework, all processes have to be involved in the communication, the sending of messages is done via *broadcasts*[4] [27]. However, since failures are expected to happen, even a broadcast activity can be affected by problems caused by failures. Thus, we need to char-

[4]when the message is sent to only a subset of a "global" destination set, then we talk, instead of broadcast, about *multicast*.

acterise reliable broadcasts, and that needs the notion of message delivery (as distinct from receipt) defined as follows:

- *Receiving a message* means that the message arrived at the destination process. This is similar to a letter arriving in one's post box.
- *Delivering a message* – usually "to an application" or "by a process" – means that the message is ready to be used by the application. Thus, message delivery in this context can be seen as analogous to taking out a letter from one's mailbox and reading it.

A reliable broadcast is a broadcast with the following three properties:

- *agreement*: all correct processes deliver the same set of messages
- *integrity*: a message is not delivered unless it is sent, and it is delivered only once
- *validity*: a message sent by a correct process is eventually delivered by all correct processes

¿From the agreement property it follows that every message is either delivered by all correct processes or by none of them, which is a desirable property specially when broadcast can be affected by failures.

Message ordering

Besides being reliable, sometimes a broadcast primitive has to give also different message ordering guarantees. Thus, even if all sent messages are received by all destinations, the order in which these messages are delivered by all processes is important. Depending on the delivery order restrictions, we can distinguish different types of broadcast primitives.

One possibility is to ask for a FIFO delivery order (thus, related to a sender) - messages from sender A are delivered in first-in-first-out order. There is no restriction about messages from sender A as compared to messages from sender B.

A second possibility is to ask for causal delivery order. In this case it is required that before delivering a message, all messages causally preceding the message should be delivered first. For node B delivering m sent by node A, one method would be to first deliver all messages that were delivered by A before sending m, and then deliver m [7].

A third possibility is to request the same order for delivery of messages. This means that messages should be delivered at all the receiving nodes in a total order, also called *atomic* order. Of course, this atomic order property refers to ordering of messages which otherwise are not restricted by other criteria (like causal or FIFO delivery ordering).

Further, we can have broadcast primitives which can give combinations of ordering guarantees. For example FIFO-causal means that messages are delivered in a causal order and for messages which are not causally dependent on each other, if they are received from the same sender, they are delivered in FIFO order).

4 Consensus as a basic primitive

Consensus was mentioned earlier as a useful operation in distributed systems. The operation is usually defined in terms of the two notions *propose* and *decide*.

Replicas typically need to agree on a joint value within a reasonable time. Besides termination of the process, the value decided upon has to be a valid one, in the sense that it has been proposed by one of the replicas. The consensus operation is invoked by the replicas at a certain moment when there is a need to get a unique result. Each process proposes its value. After a set of steps, the processes arrive to a point when they decide on the final value. Of course, the algorithm is constructed in such a way that it will be guaranteed that processes decide on the same value. Summing up the above, an algorithm which solves consensus has to satisfy the following three properties:

- Agreement - no two (correct) processes decide differently
- Validity - each (correct) process decides at most once and on a value which was previously proposed by one of the processes (or is related to those proposed values)
- Termination - every correct process eventually decides

The relation between a number of problems in distributed systems and consensus has been a major area of research. It has, for example been shown that some problems can be solved by using consensus (and vice-versa). Thus, some problems in distributed systems are reducible to consensus and vice-versa. Examples of such problems are the atomic broadcast problem [10], and the atomic commit problem [25]. For the first case, the authors prove the equivalence of the two problems. The second problem is actually a particular instance of the consensus problem [25]. Thus, here, the reduction is only in one way.

For example, let us take the case of the atomic broadcast problem. Processes have to agree on the order they deliver a set of messages. When all processes have the set of messages in possibly different orders, they will start a consensus algorithm to choose the unique order of delivery. This is an intuitive way of showing that the ABP is reducible to consensus - if there is an algorithm which solves consensus, then the ABP can also be solved. On the other hand, if there is a solution for the ABP, meaning that all processes in a message delivery step will deliver the same message, then we can say that processes agree on the "value" of that message and thus reach consensus.

4.1 Basic algorithms for consensus

No-recovery/no-partition model

Chandra et al. give solutions to the consensus problem in asynchronous systems with unreliable failure detectors [10]. The no-recovery assumption is used here in the sense that after a (real) crash a process is not repaired. It is possible, on

the other hand to "change one's mind" if a process is only suspected to have crashed.

They prove that consensus is solvable with the weakest category of failure detectors. This type of failure detector is called *eventually weak* and satisfies weak completeness and eventual weak accuracy (see below).

Two algorithms are given (in pseudo-code form) to solve consensus: one for the system equipped with *strong* failure detectors (those satisfying strong completeness and weak accuracy), and one for the system equipped with *eventually strong* failure detectors (those which satisfy strong completeness and eventual weak accuracy).

In the first case, consensus can be solved even in the presence of $n-1$ process failures. Given that the weak accuracy property is satisfied, we have that there is a correct process which will never be suspected by any correct process. The consensus algorithm is distributed in the sense that each process executes the same steps. The steps contain also synchronization points when in a certain iteration – also called a *round* – a process waits for messages sent in the same round by the other non-suspected processes. The no-partition assumption is used, even if it is not specified explicitly. By the fact that there exists a correct process which is never suspected as being down, disjoint sets of processes which work on their own, can never appear.

In the second case, consensus can be solved in the presence of at least a majority of correct processes. This algorithm is based on a rotating coordinator approach (this coordinator is the decider). By eventual weak accuracy we have that eventually there is at least one correct process which is not suspected by any correct process. Thus, eventually, the current coordinator will manage to decide the final value. It has to be mentioned that by suspecting the coordinator to have crashed, processes move to a new round with a new coordinator. Thus, the coordinator in the previous round did not manage to decide. However, as mentioned above, this situation will not last forever. In this case, the no-partition assumption is used in its primary-partition form. By always needing the agreement of at least a majority of processes in order to take a decision, it is clear that there cannot be more than one set of processes that decide.

In the article, the authors prove that the given algorithms solve the consensus problem, by satisfying the three mentioned properties.

Recovery/no-partition model

Aguilera et.al adopt a totally different view on solving the consensus problem and come up with a new notion of failure detector suitable for the crash/recovery assumption[1]. A crashed process can be repaired, and can thus recover after its crash. Of course, the algorithm also incorporates suspicion revocation.

The new type of failure detector that is needed is justified as follows. There can be processes which crash and recover infinitely often. Thus, it would be too restrictive for them to be permanently suspected as prescribed by the strong completeness property in the sense of Chandra et al. [10]. They show that the unfounded permanent suspicion in the sense of Chandra et al. can lead to

problems even in synchronous systems whenever the recovery assumption holds. This is because, the future cannot be predicted. A process which has crashed a number of times, cannot be predicted as being up (forever) or down (forever) in the future. Somehow, the failure detector implementation – by following some new specification – together with the new consensus algorithm has to cope with this problem.

Another contribution of this work is to study the need for stable storage under recovery assumption. Stable storage is some storage device (like, e.g. a disk) on which data can be permanently stored, and used even after a processor "looses its memory" (by crashing). The authors prove two theorems which in summary lead to the following result:

In the case when the number (n_a) of always-up processes is less than half the total number of processes, and besides that, the number of bad[5] processes is not less than n_a, the consensus problem is not solvable without stable storage. This impossibility holds even if the failure detector used is eventually perfect[6], there are no lossy links and no *unstable* processes. A process is unstable if it crashes and recovers infinitely often. Further, even though stable storage can be used, if it is only used for storing the process's proposed and decided values, the problem is still not solvable. Thus, the problem is solvable without stable storage, only in case the always-up processes form a majority of the whole set of processes or outnumber the bad processes.

To see this consider a case where the number of bad processes is not less than the number of good processes. Then there can be two groups of decision maker processes. Recall that there is no stable storage to remember full states. Thus, these two groups of actually correct processes can decide two different values if e.g. their common members crash and forget their state. This is contradictory to the agreement property of the consensus.

In the case where the number of bad processes is less than the number of always-up processes, Aguilera et. al. provide an algorithm based on the rotating coordinator approach. The algorithm works with no stable storage. The idea is, that processes which crash and then recover are not used further in the algorithm, i.e. when deciding on the final value. These processes can only receive the final result when it is decided upon.

In case of using stable storage to store full state (not only proposed and decided values) the algorithm is not much different. The number n_a of always-up processes has to form, in this case, a majority of the total number of processes. Every action, e.g. proposing a value, is recorded on stable storage together with some timing information. This is the current timestamp for the process – the round number in which the proposal was made. These are used later, in case the process crashes and then recovers. They indicate where in the algorithm the crashed process stopped; this will be used as the point to continue after recovery.

[5] Here bad means not-always-up.

[6] Eventually perfect detectors satisfy eventual strong accuracy and strong completeness.

4.2 Perfect failure detectors

Sometimes, when solving consensus problems, besides the minimal safety requirements like agreement and validity (i.e. decision on a unique valid value), an important demand is for decision making to be based on using proposed values from *all* correct processes[7]. This requirement can not be satisfied if the failure detector makes mistakes related to accuracy, i.e. it suspects a correct process to have crashed, thus ignoring its value in the decision phase.

Helary et.al discuss the problem of decision based on a global set of values in the presence of perfect failure detectors [28]. This can be seen as a strict(er) form of consensus – called by the authors *global function computation.* Of course, the questionable part of this argument is the method for implementing perfect failure detectors. This relies on a somewhat "perfect real-world" assumption: the possibility of having privileged channels – channels that are never congested and never go down, and therefore have predictable timing behaviour. These channels, thus, could be used by processes for sending "I am alive" messages.

Here, as already mentioned before, the no-recovery assumption is used, in both its senses. First, a process does not recover form a real crash, and secondly, there is no such thing as revoking ones suspicion about a process crash. The reason is that failure detectors used are perfect and they do not suspect a crash if this is not real. The no-partition assumption is quite obvious here - again because of the use of perfect failure detectors. There is no mistaken crash suspicion, thus all correct processes will see the same processes as being up or down. As a consequence, there is no possibility for the appearance of disjoint sets of separately working processes.

The algorithm given is performed in rounds, as in usual consensus solving algorithms. At the beginning of a round, say i, every correct process sends its global data set to all processes expected to be correct in round i (i.e. those processes which participated in previous round). Further, the set of such processes is built up for the next round, from those which were senders of global data set information in round i. The algorithm has a termination condition: agreement is reached when either the number of executed rounds exceeds a certain value (the maximum number of processes that may crash), or all processes expected to be correct in a round are those which actually are correct in that round and, in addition, all these have the same knowledge of the global data set.

5 The process group abstraction

Intuitively, a group can be seen as containing a set of replicas being able to do the same thing. Here we have the difference between *processor groups* running a set of replicated services, and *service groups.*

The service groups (servers) are also processor groups, but they all run the same service. Of course, the issue is that a processor can run more than one

[7] For example, this is needed to solve the strict version of the atomic commit problem, when it is not permitted to ignore any answer (YES or NO) from a correct process.

type of service. To maintain service replication, for each service we must know the number of processors running that service, and if this drops under a certain limit (the replication limit for that service), then the service has to be started on some other processor.

One of the successful experimental platforms around the ideas of software replication were implemented in the Delta-4 project [41]. This work leaves the decision of method of replication as an open question for the application and provides several modes to choose from in the open architecture. The project was discontinued in 1992, but several mechanisms were studied during the project.

5.1 The notion of *group*

In previous sections, although the word *group* was sometimes mentioned, it was used only in an intuitive sense. In this section, the notion will be more precisely delimited. Thus, what is meant by *group* in the fault-tolerance community is the following: a group of processes (or processors) which cooperate with each other, is a *mechanism* for building fault-tolerant service providers. The members of the group are not fixed apriori, and the group mechanisms take care of dynamic changes to the group. A service provider, in this context, is a collection of replicated servers which process requests from clients. The important aspect is the transparency from the client's side. Thus, the client, when sending a request to the replicated server (or now, a process group), perceives it as one addressable unit.

The group actually underlying the service is not visible for the user of the service. It might be used to manage complexity, to balance load, or to provide fault-tolerant services [42]. A consequence is that the group has to manage itself to assure the one-unit image.

An important point becomes the notion of *group member*. A process may or may not be a member of a group at different times. If it is not, it might have been excluded from the group due to a crash (real or suspected), or due to isolation from the group because of communication failures. In the latter case, it is possible that the process is still working, but its view of the group membership differs from that of other processes' views. This case is conforming to the partition failure model. As already mentioned earlier, in a distributed system, it is possible to have several groups of processing units working on their own. Sometimes this is permitted, sometimes not.

5.2 Replication strategies

A replica group can be constructed in several ways:

- Primary-backup: in this approach, there is only one replica which processes requests from clients and answers to them (the primary). The backups in this case, are just receiving state update information from the primary so that in case of a primary crash, take over the job of the primary in a predetermined manner[26].

- Active replication: in this approach, all replicas process requests from the client in an active way (in this case, the notion of group is not used in its above mentioned strict sense - it is not necessary for the replicas to know who is up and current member of the group).

  The restriction here is that all replicas perform deterministic computations in a state machine fashion [47]. Depending on the failure semantics of the replicas the client has to wait for the first answer from any replica, or for an answer resulting from a majority vote performed by the replica group.

  In the latter case, processes do not communicate with each other explicitly – consistency among them is guaranteed by the deterministic processing of the same inputs.

- Semi-active strategy: in this approach, there is a central replica which does all the computation steps and which leads all the other replicas. In case of a non-deterministic choice, the replicas wait for the leader's decision, and they follow it [15].

- Semi-passive strategy: in this ("semi-primary-backup") replication style, the primary is "elected" by a consensus algorithm [41].

The notion of *state* might need some clarification. The state (as used in "state update" or "state machine") refers to the collection of data on which the processes act. It might include actions performed up to a certain point during a computation.

In all types of replication strategies, failures have to be handled such that the states of the replicas remain consistent. In the first case, if the primary fails, this has to be noted by the backups which have to choose a new primary, and continue providing services from where the primary left it. To choose a new primary, a membership service, which indicates the composition of the group, can be used. In case of active replication, failures of members can be masked by using a voting mechanism on the results (if Byzantine failure semantics is considered), or by sending one of the answers (the fastest one) in case of crash/performance failure semantics of the replicas. In the former case, the number of replicas has to be $3f + 1$ if the failure of f replicas has to be masked [30]; whereas in the second case, it suffices with one replica more than the number of failures tolerated [47].

5.3 Typical group services

An undispensable service in a group setting is the one which informs about the *group membership*. This is used by the group members themselves to be able to assure transparency even in presence of failures which should involve masking. A membership service has to ensure that group members have a consistent view of the group.

Another specific group service is the state merging (or transfer) service. This is used when the initial group is split in several (or one majority and some

minority) (sub)groups. When, e.g. links are repaired and partitions disappear then all new group members have to know the state of the rest of members. The state transfer service is also used in case a new node joins the group. The operation of state transfer has to occur as an atomic action.

5.4 Specification and implementation of group services

Extensive work on the group membership problem can be found in the works of Cristian ([11, 13]). He gives a formal description of membership service properties both for synchronous systems and asynchronous systems.

For explaining problems in presence of the asynchrony assumption, he introduces the notions of stability and instability of a system. When a system is stable, then communication between processes either works all the time with constant (bounded) delays, or it does not work. In contrast, during instability periods there are no guarantees of message delay bounds; the communication can be working, but very slow, or not at all.

The important thing is that the membership service has to ensure the agreement of group members on the group view and, of course, to correctly include and exclude members from the view.

Following the line of Cristian's work, is the formal specification and implementation of a membership service for partitionable systems, by the group from Bologna University [4]. They give a formal description – by introducing some quite simple properties – of a communication infrastructure for processes in a partitionable distributed system. Thus, here, the failure model is built using the partition assumption, and partition failure can be also present.

In this setting, a processor does not recover after a real crash. Thus, the no-recovery assumption with its first sense is used here. On the other hand, it is possible to reverse one's suspicion about a crash. This, actually, follows from the fact that the suspicion itself can be related to process crashes as well as process unreachability.

Work by Birman et. al on the ISIS system, has minted the notion of virtual synchrony and view synchronous multicast. The first notion is related to the provision of well chosen "synchronization" points for processes in an asynchronous system. Thus, a process will stop and "synchronize" with other processes, when it has to wait for messages causally preceding other messages ready to be delivered. The second notion is tightly coupled with the former one, and directly relates to groups. Thus, a view synchronous multicast (or broadcast) works as follows: a message sent by a process while the process membership view is V, has to be delivered before any other new view W is registered[8], by all members of V. In this context, one says that a group uses a view-synchronous multicast service.

Although one of the leading groups in the area of group communication in the early 90's, much of the work done on ISIS in those years can be characterised by a "system building" approach. A lot of the detailed algorithms were not

[8] "registered" is referred to as *installed* in [7]

formally (or clearly) presented and the behaviour of the system was analysed pragmatically.

5.5 Application platforms

Besides the ISIS system, there can be found some other fault-tolerant distributed system platforms.

One of these is Totem developed by L.E. Moser et. al. [37]. This system provides a platform for writing distributed applications. It is structured in several layers, deals with sending messages among different process groups, enforces total order on delivery of messages, and also deals with merging of groups. An extension of the notion of virtual synchrony is introduced here: the extended virtual synchrony. In this model, care is taken even of the processes not in the current group of a certain process. Thus, messages which are received and delivered by certain processes are delivered in the same order by all of these, regardless of whether they (currently) have the same view on their group or not.

Messages are delivered either in agreed, or safe total order. Agreed means that a process may deliver a message only if all messages prior to it, sent by processes from the same group were delivered first (classical causal delivery), while safe means that a process delivers a message only if it knows that all processes in its group received it as well. In Totem born-ordered[9] messages are used [37]. This is needed to guarantee extended virtual synchrony.

The application writer can request a certain type of message delivery (agreed, safe or both) and can rely on the fact that the system will enforce the total order delivery. The system is suitable for use in soft real-time systems, since high throughput can be achieved, as well as predictable latency.

Another system, developed again by Birman's group is Horus [50]. It is equipped with group communication (e.g. message passing) primitives, providing flexibility to distributed applications developers.

This system allows flexible composition of protocols to support groups. The application developer can use well-defined protocol components (i.e. with well defined interfaces) to build the needed protocol stacks, while excluding unnecessary overheads. Not all protocol block configurations make sense; only those for which one block's output interface can be connected to the other block's input interface. The authors mention their successful experiences with Horus [44], as well remaining challenges to be taken up later.

6 Measures of efficiency

Algorithms for solving problems in distributed systems are expected to satisfy some specifications. These contain some properties – usually not connected to efficiency – which have to be fulfilled each time the algorithm is executed. Examples are agreement and validity in connection with consensus. It follows

[9] the relative order of two messages can be determined from information provided with the message by its sender

that there is no simple way to compare two algorithms in terms of efficiency. At least not looking at the specification of properties. Using the specification, only correctness of the algorithm can be proved. Thus, an interesting question is: what can one look at when reasoning about the efficiency of a distributed algorithm?

Some criteria could be related to time complexity – meaning the possibility of bounding the execution time. Further, two cases can be considered: the best case (when no failures occur), or the worst case when failures occur. For this criteria, analysis can include, e.g. the consideration of round numbers for an algorithm execution. This, of course, is interesting to be done when the round number is not fixed (like in [10] - the second type of algorithm). Other criteria could be related to resource utilization (like memory or stable storage). The communication media can also be regarded as resource, thus the number of messages exchanged during communications is also a measure of efficiency.

Another way to look at measuring efficiency is to see how an algorithm behaves if the worst case assumption under which it was built does not hold (e.g. the assumption is that n failures can occur, but, in fact, only m<<n failures occur in 90% of the cases).

Cristian is one of the few authors discussing real-time aspects (i.e. bounds on time) in the context of distributed algorithms [11, 13] . Of course, in a synchronous setting it is not so difficult to reason about bounds on, say, agreement time, and thus on response time to the client request. In an asynchronous system, the problem is somewhat more challenging. Thus, as already mentioned before the system model is not totally time-free. There can be some bounds on message delays, but these hold only during stability periods of the system. The interesting part is that the analysis is done by taking in consideration the instability periods of the system, as well. The algorithms described for asynchronous settings do have quite long worst case stabilization times.

Helary et.al. give upper (worst case) bounds for the number of rounds executed in their algorithms [28]. Aguilera et.al. give time estimations and message number complexity for best case executions and compare them with the same condition performances for other algorithms [1].

7 The transaction analogy

An area somewhat related to the above discussions is the one concerned with replicated databases and transaction processing. The notion of *transaction* was developed in the 70's to ensure a consistent view of the system despite a node crash in the middle of an updating activity [6], thus implementing an *atomic multicast* at an application level. A distributed transaction ensures that for a given update either all nodes are updated or none of the nodes.

Mechanisms for *group communication* were developed in the 80's to make convenient use of replication of components (masking failure both in software and hardware) [42]. The incorporation of group services provides separation of concerns, so that the application developer can concentrate on development

of an application, relying on in carefully selected middleware services. In contrast to the transactional approach, when reaching consensus in a replica group, the client application need not be aware of any failures that are transparently dealt with in the replicated server group [8] – a new primary is transparently selected, a consistent group state is transparently maintained, etc. However, transparency costs time – and there are very few trade-off studies to show how this affects the performance [18].

Moreover, recent studies show that an application may pick and choose one mechanism or another (transaction or group communication) depending on own characteristics – or more advantageously, combine transactions with group communication [46, 33]. This aspect is gaining more importance as many distributed network-based applications are realised as three-tier systems where the application tier contacts a data repository via a server tier. Also, many server groups, in order to serve a client transparently need access to address and parameter databases for the server group. A data base which can in turn be replicated for availability reasons.

There are similarities and, of course, differences between the two problems. Wiesmann et al. [52], compare the *replication in distributed systems* and *replication in database systems with transaction processing* issues, by giving them a common denominator. A common denominator can be established by considering the phases of a client request processing. These phases range from receiving the request, to sending the response. In each type of system these phases have particular instances, but their main features are common: that several distributed nodes must coordinate before committing or deciding a value.

Analogous to the replication strategies in distributed systems mentioned in section 5.2, there are different ways of dealing with transaction processing in replicated databases. These are:

- eager primary copy replication - meaning that the transaction is executed at a primary site and then changes are propagated to the backup sites – before sending any result to the client
- eager update everywhere replication - meaning that the transaction is executed at all sites and the result is sent to the client only after all replicas did all the operations in the same order
- lazy replication - meaning that the transaction is executed locally at the site where the client connects, the result is sent to the client and then the rest of the replicas are updated

Thus, *eager primary copy* replication is analogous to passive (primary-backup) replication. *Eager update everywhere* replication is analogous to active replication. Finally, the *lazy* approach does not really have a correspondent because in fault-tolerant distributed systems it is very important to have consistency before answering any request.

Schiper and Raynal [46] relate group communication with transactions in a somewhat different way. They refer to the ACID properties of a transaction

system: Atomicity, Consistency, Isolation, Durability. The discussion is about how such properties are satisfied (relative to a data item update) when an atomic broadcast group communication primitive is used in a process-group setting. Thus, they consider the following scenario: to perform a transaction consisting of, say, two operations is possible to have a group of n (say, n=3) of processes. Atomic broadcast is used. This will assure that either all or none of the processes receive the operation requests and that they receive them in the same order. Thus, it is possible to satisfy the enumerated ACID properties.

8 Adding replication services to the middleware

Little et.al. present a way of integrating a group communication service with transactions [33]. They start with a system which supports transactions, but no process groups. Then, they enhance the use of transactions by introducing process groups. Further, the authors consider the possibility of using a widespread middleware like the Common Object Request Broker Architecture (CORBA) [40], to implement persistent replicated objects.

Thus, there are objects accessible by name. Each object is an instance of some class which defines the instance variables of that object and the set of methods which define the externally visible behavior of that object. The system has a binding service which maps object names to a location where a server for that object is run. Further, the system has to contain an object storage service, which can manage storing and retrieving of object states. finally, there has to be an remote procedure call (RPC) service which provides an object invocation facility.

When an object is looked up, the binding service will indicate the node where the object server is activated. If the object was passive, then it will be activated, and the server will start on some node. Its address will be returned to the application. The address of the node where an object store server runs is also returned (this node is not necessarily the same as the object server's node). A second client which wants to use the same object, will be given the address of the node where the object server is already activated.

In the replicated version, the authors mention two ways of activating an object server:

- the object server is activated on a single node, and the object store server is replicated on several nodes
- the object server, as well as the object store server, is replicated on several nodes

There is some information which has to be kept updated by the binding service in a consistent way. This information is related to which nodes are active for a certain object server, and at which location are the object store servers active.

If the replication is done by only using transactions, then the information is kept in a "central" place (a kind of centralized membership service) and updated in a transactional manner. If the replication is done in the most normal way, by using groups (and group communication), then the data is provided by the membership service distributed at each node in the replicated system.

First, the transaction based replication is analyzed. In this case the server for an object is activated only at a single node, and the object store servers are replicated.

The binding service must record for each object the list of nodes where that object can be activated. Further, the node where the server is actually activated has to be known. If, after the object server was activated, a second client wants to use the services of the object, the binding service will return the same node.

What can happen is that the client, trying to connect to the object server, concludes (maybe wrongly) that the server's node is faulty. In this case, the server will be activated on a new node (from the specified list), and this fact will be recorded in the binding service instead of the previous information. But, if the client's conclusion was wrong, then this is an inconsistency, because the server is not allowed to be activated on two correct nodes at the same time.

Here lies the drawback of managing group information in a system which supports only transactions. Thus, the solving of any inconsistencies is delayed until the client comes to "unbind" itself from the object. If it realizes that the information in the binding service does not reflect reality, it aborts the whole transaction. This is because, if it would not do so, then it would write state information at the object stores and this information would not be the correct one. If there are no inconsistencies detected, then the state of the object server is to be written at the object stores. Those nodes which do not succeed in the update procedure, are deleted from the binding service's list.

If the group based replication is used we come back to the old problems. In this case the membership service is distributed and view changes are initiated by sending "view change" messages along with "normal" messages. In this case, the authors describe the situation where an object server, as well as the object store server, can be replicated on several nodes. In this case, the binding service has a list of nodes where a server for object A can be activated, and the list of nodes where an object store server can be activated. The client, when performing the "bind" operation, obtains a list from where it can choose the group members on which it can activate the server for object A. If primary-backup replication is used, then the primary of the group is registered in the binding service.

In this case, no inconsistency like the one mentioned before, can occur. This is because the membership of the server group (in other words, the identity of the primary to which connections are made) is managed transparently by the group management service (not based on some indications from the client - application).

While this avoids the inconsistency aspect above, other problems can appear. Because no transactions are used, the system can be vulnerable to total group failures. Thus, when the object store updating happens, and the list of the nodes at which this succeeds has to be updated correctly, all group members might

fail, thus leaving the list inconsistent with reality. With transactions, if this operation can not be successfully completed, then the transaction is aborted, thus not leaving inconsistent information. Little et.al. suggest that a good trade-off is to use group communication in a transaction based system. In this way the replicated binding service can be managed as a group. The same thing is valid for the set of object server and object store nodes.

8.1 Standard middleware services

For many applications in which a distributed service has to be used (like Web service), a nice solution can be to use standard middleware (like CORBA) to obtain fault-tolerance.

CORBA accommodates an object transaction system (OTS). Thus, fault-tolerant services can be implemented by using CORBA with its existing transaction module, to replicate objects for fault-tolerance.

Prior to the specification of the FT-CORBA extension, few works had studied alternative augmentations of CORBA with a process (object) group module. Felber et.al. show some possible approaches for introducing "object groups" in CORBA [18]. Three different approaches are presented depending on the "position" of the "group communication module" relative to the ORB.

If the integration approach is used, then the group toolkit is integrated in the ORB. Thus, each request to the ORB directed towards a process group is sent as a multicast message by the group toolkit.

If the interception approach is used, the ORB is totally "detached" from the group toolkit. There is an extra "layer" between the group toolkit and the ORB. This is the interception layer. The client sends an invocation to a group to the ORB which is not aware of groups, further, at a lower level, the interceptor "receives the call" and forwards it to the group toolkit. The latter one transforms the call in multicast messages. On the server's side, the messages are again "intercepted". Further, they are sent to the ORB as calls on the separate objects which form the group.

Finally, the most CORBA-oriented approach is presented. This is the "object service" approach. Thus, the group toolkit is implemented by using a set of classes, as all other object services, such as naming.

The interceptor approach is considered to be more general in the sense that several "deficiencies" of current CORBA systems can be overcome by interceptors: non-application components which can alter the behavior of the application without changing the code or the code of the ORB. For example, Narasimhan et.al. propose that interceptors be used not only to achieve fault-tolerance, but also for several other purposes: real-time scheduling, profiling and monitoring, as protocol adaptors for protocols other than IIOP and for enhancement of security [38]. However, implemented ad hoc, this is the least clean and scalable approach in the sense that every application adds its own patches for dealing with middleware deficiencies. The recent addition of portable interceptors to the CORBA specifications is a step towards avoiding ad hoc solutions.

8.2 FT-CORBA specification

The generic CORBA Specification by the OMG group was recently extended to provide application developers with support for fault-tolerance. The Fault-tolerant CORBA Specification V1.0 was adopted in April 2000 [39].

To obtain fault-tolerant applications the underlying architecture is based on replicated objects (replication in space). Temporal replication is supported by request retry, or transparent redirection of a request to another server. Replicated application objects are monitored in order to detect failures. The recovery in case of failures is done depending on the replication strategy used. Support is provided for use of active replication, primary/backup replication, or some variations of these. The non-active replication styles provided are warm passive (semi-passive), or cold passive (primary/backup).

There is a Replication Manager which inherits interfaces such as PropertyManager, ObjectGroupManager, GenericFactory. The standard also informally refers to the notion of Fault-Tolerance Infrastructure (hereon we refer to it as FTI). This is implicitly the collection of mechanisms added to the basic CORBA to achieve fault-tolerance.

- The PropertyManager interface has methods used to set the replication style, the number of replicas, the consistency style (infrastructure-controlled or application-controlled), the group membership style (infrastructure-controlled or application-controlled).
- The ObjectGroupManager interface has methods which can be invoked to support the application-controlled membership style, at the price of losing transparency.
- The GenericFactory interface has the method *create_object*() which in case of replicated objects is called transparently by the Replication Manager in response to an application *create_object* method call. Each object in a group has its own reference, but the published one is the inter-operable object group reference (IOGR).

To be able to manage large applications, the notion of fault-tolerance domains is introduced. Each fault-tolerance domain contains several hosts and object groups and a separate Replication Manager is associated with it.
Replica consistency is called strong if:

- in case of active replication, after every method call, all object replicas have the same state. Here, strong membership is needed meaning that on every node the FTI has the same view of the membership of the object group.
- in case of passive replication, after every state transfer, all object replicas have the same state. Here, the uniqueness of the primary is needed – at all times only one object replica executes methods invoked on the object group.

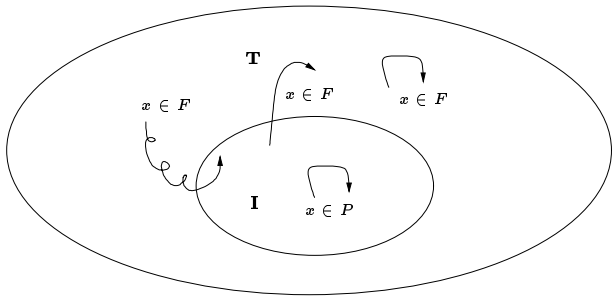

Figure 1: Results of actions on reachable states: normal, faulty, corrective.

Every application object has to implement two interfaces – PullMonitorable and Checkpointtable – for the purpose of fault detection and logging. The logged information is later used if the object fails, and a new replica has to take over its role (and state). The logging and recovery are used only in case of infrastructure controlled consistency.

The main requirements in the specification are related to the compatibility of non fault-tolerant objects with object groups. Thus, it is possible to call a method of an "object group" by using its IOGR by an object which is not replicated (and vice-versa).

There are some limitations in the FT-CORBA specification. For example, the ORBs and FTI on hosts in the same fault tolerance domain have to be from the same vendor. If this is not the case, then full inter-operability is not always possible. Another limitation is related to the types of faults detected and treated. Thus, no correlated (design) faults are treated. Also, no mechanism is provided to treat partition failures. There is a quite simplistic way to protect against Byzantine failures: the *Active-with-voting* replication style can be used. A nice overview of the FT-CORBA mechanisms can currently be obtained from tutorial notes by Moser et.al. [36].

9 General formal model

Obviously research on fault-tolerance has been large and diverse. So none of the earlier works attempted to create an overall theory for fault-tolerance. In this section we mention the beginnings of a general theory of fault-tolerance for the sake of completeness.

In recent years Kulkarni and Arora attempt to come up with such a general framework [2, 31]. They give algorithms for adding fault-tolerance to a system which is fault-intolerant. Given a specification for the system for the no-fault situation, the fault-tolerant system has to be designed such that it does not infringe on the earlier specification. However, it is able to satisfy the original specification even in presence of (a given class of) faults.

The framework by Arora and Kulkarni (also followed by Gärtner [24]) models distributed programs as sets of processes where each process is described as guarded commands [2, 24]. It is proposed that systems be modelled as fault-intolerant (i.e. describe the behaviour in the absence of faults) using guarded actions which correspond to *normal* system behaviour.

The *behaviour* of a process is a set of state sequences. At any state, if the guard for any action is true in the state, the command *may* be executed, after which some state variables may be set to new values. The behaviour of the process (and also of the system) is a set of state sequences based on the interleaving of enabled actions. To avoid the degenerate cases a (strong) fairness restriction is added, whereby it is stipulated that if an action is enabled infinitely often then it will be eventually executed. The state of the distributed is referred to as "the configuration" consisting of elements from the local (process) state.

The addition of fault-tolerance is modelled in two steps: first distinguishing the undesired system behaviours, i.e. adding fault actions to the program. These model state transitions in presence of faults and potentially extend the set of states reachable by the system. What remains of the normal behaviour of the system in presence of faults is dependent on the fault model adopted (e.g. crash model, fail-stop, etc.).

Next, the fault containment or repair actions can be modelled, again adding to the set of behaviours possible by the system. Thus a fault-tolerant program contains at least a detection component, but possibly also correction components. Arora and Kulkarni then go on to formally relate the fault-intolerant behaviour to the fault-tolerant behaviour.

The effects of additional fault-related actions can be viewed as follows. A fault-intolerant program behaviour is verified against a specification which typically consists of both safety and liveness properties. The specification thus prescribes what should be in the set of state sequences for the behaviour (liveness) and what should not (safety). So, how are these properties affected when the program is augmented with fault actions and later by detection and correction actions? One way to formalise this is to consider how the choices the program has, in terms of states reachable in the future from a given state, are affected by the added actions. Fault-tolerance can be formally defined as follows and illustrated by Figure 1 [24].

Definition 1 *A distributed program P is said to tolerate faults from a fault class F for an invariant I iff there exists a predicate T for which the following three requirements hold:*

- *At any configuration at which I holds, T also holds.*
- *Starting from any state where T holds, if any actions of P or F are executed, the resulting state will always be one in which T holds.*
- *Starting from any state where T holds, every computation that executes actions from P alone eventually reaches a state where I holds.*

□

The definition above makes it possible to reason around the effects of adding fault actions to fault-intolerant systems. To summarise, the actions in the fault class F can take the program to the (extended) set of states T. Now, if the effects of these actions should have no bearing on the safety and liveness of the system, then the specification I should be so detailed to cover all the requirements in presence of faults, i.e. it should coincide with T. If that is the case, and the above definition holds, then the program has a *masking* fault-tolerance behaviour in presence of faults from class F.

On the other hand, one may choose a specification which focuses on safety or liveness. In the former case, the program can be designed to show a *fail-safe* fault-tolerant behaviour with respect to F. In such systems, the F actions can take the program to T (which for safety purposes may be not entirely acceptable). Thus, to be fail-safe, one has to ensure that all F-actions only take the system to the I subset of T. Arora and Kulkarni show that detectors alone are sufficient for implementation of safety-preserving additions to fault-intolerant systems.

In the case of liveness specifications, the system has to guarantee that any computations starting in I and ending up in T as a result of F actions, will eventually result in computations which have a suffix in I. The addition of *non-masking* fault-tolerant techniques, can thus preserve liveness in the above sense, but do not guarantee safety. Arora and Kulkarni show that correctors alone are sufficient for implementing non-masking fault-tolerance.

By formally defining safety, liveness, fault, program, in terms of states and transitions the authors manage, for some restricted fault classes to define algorithms which automatically transform a fault-intolerant program into a fault-tolerant one.

10 Some open questions and trends

Guerraoui et.al [9, 22] introduce two low-level primitives used for solving problems in distributed systems. The apparent motivation for the authors is as follows: Every computational paradigm needs its basic models and mechanisms. The primitives employed to model the basic scenarios (algorithms) in an area should be abstract enough to allow the highest level of description for these scenarios and low level enough for different languages (or systems) to be built on top of them. The analogy drawn is for example with the concept of semaphore or monitor in concurrent systems. What are the basic essential primitives in distributed systems?

The authors then go on to propose to primitives on which consensus services for distributed processes can be built: the *weak leader election* primitive and the *weak consensus* primitive. By using them in combination, consensus and thereby other distributed problems can be solved in asynchronous systems. The authors claim that by providing the right level of abstraction one gets an optimal implementation of a consensus service (measured in terms of number of messages and procedure calls, compared to alternative applications).

The argument is based on the observation that behind every failure detector lies the ability to distinguish a faulty process from a correct process (a concept they relate to the notion of election of a leader). Further, both active replication and primary-backup mechanisms require this notion of a "unique leader". For active replication this is baked into the mechanism which feeds the deterministic replicas with a totally ordered message sequence. In their approach, the primitives in themselves fulfill some properties (like the eventual existence of a unique and correct leader). Thus, the failure detection assumptions are lifted up into the primitives, and an additional failure detector level does not exist. A side effect is also that the strong distinction between active replication and primary-back up does not exist any more. There are certain intervals during which several leaders coexist, but the eventuality of a unique leader election ensures that the computations converge. It is also not necessary for the replicas to be deterministic (much more likely to be the case in practical applications).The arguments are appealing. However, more work is needed to establish whether these are *the* basic (and optimal) primitives for building fault-tolerant algorithms for distributed systems.

10.1 Correctness criteria revisited

Frolund and Guerraoui [21] also pose the question of an appropriate notion of correctness for distributed systems. In comparison with concurrent systems where well-established notions like safety and liveness can be used to analyse the correctness of an algorithm, there are no such well-defined notions in distributed systems. The nearest to liveness is the notion of termination for consensus-based algorithms. What is then the equivalent notion to safety?

The notion of *X-ability* is introduced to capture an intuitive requirement implicit in the notion of replication [21]. That is, the idea that a distributed fault-tolerant setting together with its algorithms is "correct" if it is able to execute a computation for an application as if there was only one fault-tolerant computing site.

In this context, the notions of *idempotent* and *undoable* actions are introduced. Idempotent actions are those that the effect of their execution does not change if they are executed several times (writing the same value to a memory location is for example idempotent, incrementing the value of the same location is not). An undoable action is such that the outcome of its execution is only seen after it was committed. Thus, until commit, the action can be aborted (i.e. undone) several times, without any change on the final effect. Most known operations of this type are on databases - all updates to a database item are done temporarily in memory and can be changed several times before their result will be made permanent by access to the database on a disk.

By constructing algorithms from components which have these properties, it is possible to prove the overall algorithm's correctness in terms of X-ability.

11 Final remarks

In this report we have presented an overview of the area of fault-tolerance – with an emphasis on achieving availability in distributed systems via software replication. Thus, fault-tolerant protocols (replication strategies, agreement algorithms) were described from the perspective of their behaviour in a network of communicating processing units in which both crash failures and partition failures are possible.

Since our interest is directed towards the study of real-time and quality of service guarantees which can be given in a fault-tolerant setting, some efficiency measures were also pointed out. This relatively short survey[10] serves to indicate that it is a major challenge to study resource allocation and fault-tolerance demands collectively. Most of the works on the fault-tolerance technique have no resource optimisation perspective. A separate and extensive survey would be needed to show the symmetric situation on the resource allocation side: most quality of service extensions to network middleware ignore the need for high-availability. A short survey studying the extensions of CORBA towards real-time and QoS is provided elsewhere [14].

Replication in time and its connection to scheduling theory have been intentionally left out of the scope of this survey (i.e. feasibility of a schedule in presence of failures and the duplicate running of a process which was subject to transient failures). Our aim has been to provide an introduction to works which are focused on redundancy in space supported via middleware. This can be done in many different ways and the recent CORBA specification with regard to fault-tolerance provides the basic infrastructures to experiment with.

There is also more work to be done on the theoretical side. Both validating the choice of the right building blocks in the middleware for three-tier architectures, and a general framework to place any added measure for achieving fault-tolerance in relation to the original computational processes are interesting open areas.

References

[1] M.K. Aguilera, W. Chen, and S. Toueg. *Failure detection and consensus in the crash-receovery model*, pages 99–125. Springer-Verlag, 2000.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–77, 1998.

[3] O. Babaoglu. *The "Engineering" of Fault-Tolerant Distributed Computing Systems*, volume 448 of *Lecture Notes in Computer Science*, pages 262–273. Springer Verlag, Berlin, 1990.

[4] O. Babaoglu, R. Davoli, and A. Montresor. *Group Communication in Partitionable Distributed Systems*, chapter 48-78. Springer-Verlag, 2000.

[10] It is by no means comprehensive, but most major trends are well-represented.

[5] M. Banatre and P.A. Lee, editors. *Hardware and Software Architectures for Fault tolerance*. Springer-Verlag, 1994.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[8] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53,103, December 1993.

[9] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. Technical report, Ecole Politechnique Federale de Lausanne, January 2001. Available at http://dscwww.epfl.ch/EN/publications/list.asp.

[10] T.D. Chandra and S.Toeug. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[11] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4, 1991.

[12] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 1991.

[13] F. Cristian and F. Schmuck. Agreeing on processor group membership in timed asynchronous distributed systems. Technical report, UCSD, 1995.

[14] C. Curescu and S. Nadjm-Tehrani. Review of support for real-time in corba-based architectures. Technical report, IDA, Linköping University, October 2000.

[15] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 43–50, 1998.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 1988.

[17] J.C. Laprie (Ed.). Dependability:basic concepts and terminology, August 1994.

[18] P. Felber, R. Guerraoui, and A. Schiper. *Replication of CORBA Objects*, volume 1752 of *Lecture Notes in Computer Science*, pages 254–276. Springer Verlag, Berlin, 2000.

[19] M. J. Fischer. *A Theoretician's View of Fault Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 1–9. Springer Verlag, Berlin, 1990.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985.

[21] S. Frolund and R. Guerraoui. X-ability: a theory of replication. Technical report, Ecole Politechnique de Lausanne, January 2000. Available at http://dscwww.epfl.ch/EN/publications/list.asp.

[22] S. Frolund and R. Guerraoui. Democratizing the parliament. Technical report, Ecole Federale Politechnique de Lausanne, January 2001. Available at http://dscwww.epfl.ch/EN/publications/list.asp.

[23] J. Gray. *A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem*, volume 448 of *Lecture Notes in Computer Science*, pages 10–17. Springer Verlag, Berlin, 1990.

[24] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 1999.

[25] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliviera, M. Raynal, and A. Schiper. *Consensus in Asynchronous Distributed Systems: A Concise Guided Tour*, volume 1752 of *Lecture Notes in Computer Science*, pages 33–47. Springer Verlag, Berlin, 2000.

[26] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, pages 68–74, April 1997.

[27] V. Hadzilacos and S. Toueg. *Fault-Tolerant Broadcasts and Related Problems*, pages 97–147. Addison-Wesley, 1993.

[28] J.Helary, M.Hurfin, A.Moustefaoui, M. Raynal, and F. Tronel. Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel and Distributed Systems*, 1999.

[29] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in mars. In *Digest of Papers of the 20th Ineternational Symposium on Fault-Tolerant Computing*, 1990.

[30] C.M. Krishna and K. G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[31] S. S. Kulkarni and A. Arora. *Automating the Addition of Fault-Tolerance*, volume 1926 of *Lecture Notes in Computer Science*, pages 82–93. Springer-Verlag, 2000.

[32] J.C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 15th IEEE Annual International Symposium on Fault-Tolerant Computing*, pages 2–11, June 1985.

[33] M. C. Little and S. K. Shrivastava. *Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects*, volume 1752 of *Lecture Notes in Computer Science*, pages 238–253. Springer Verlag, Berlin, 2000.

[34] C. D. Locke. *Fault Tolerant Application Systems; A Requirements Perspective*, volume 448 of *Lecture Notes in Computer Science*, pages 21–25. Springer Verlag, Berlin, 1990.

[35] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):220–232, February 1988.

[36] L. Moser, M. Melliar-Smith, and P. Narasimhan. Tutorial on Fault Tolerant CORBA, September 2000. University of California at Santa Barbara, currently appears at `http://beta.ece.ucsb.edu/ priya/DOCS/ftctutor.pdf`.

[37] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[38] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Using interceptors to enhance corba. *IEEE Computer*, pages 62–68, July 1999.

[39] OMG. Fault tolerant corba specification, April 2000. available at ftp://ftp.omg.org/pub/docs/ptc/00-04-04.pdf.

[40] OMG. The common object request broker: Architecture and specification. OMG Formal Documentation (formal/01-02-01), February 2001. Available at ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf.

[41] D. Powell. *Distributed Fault Tolerance - Lessons Learnt from Delta-4*, volume 774 of *Lecture Notes in Computer Science*, pages 199–217. Springer Verlag, Berlin, 1994.

[42] D. Powell. Group communication. *Communications of the ACM*, 1996.

[43] B. Randell. Software structure for software fault tolerance. *IEEE Transactions on Software Enginering*, SE-1(2):220–232, June 1975.

[44] R. Van Renesse, K. Birman, R. Friedman, M. Hayden, and D.A. Karr. A framework for protocol composition in horus. In *Proceedings of the 14th Symposium on the Priciples of Distributed Computing ACM*, pages 80–89, August 1995.

[45] A. Ricciardi, A. Schiper, and K. Birman. Understanding partititons and the "no partition" assumption. In *Proceedings of the Fourth Workshop on future Trends of Distributed Computing Systems*, 1993.

[46] A. Schiper and M. Raynal. From group communication to transasctions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[47] F. B. Schneider. Implementing fault-tolerant services using the state machine approach:a tutorial. *ACM Computing Surveys*, 22, 1990.

[48] D. P. Siewiorek. *Faults and Their Manifestation*, volume 448 of *Lecture Notes in Computer Science*, pages 244–261. Springer Verlag, Berlin, 1990.

[49] B. Simons, J. Lundelius, and N. Lynch. *An Overview of Clock Synchronization*, volume 448 of *Lecture Notes in Computer Science*, pages 84–96. Springer Verlag, Berlin, 1990.

[50] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[51] P. Verissimo and M. Raynal. *Time in Distributed System Models and Algorithms*, volume 1752 of *Lecture Notes in Computer Science*, pages 1–33. Springer-Verlag, 2000.

[52] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of The 19th IEEE Symposium on Reliable Distributed Systems*, 2000.