Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments

FELIX C. GÄRTNER

Darmstadt University of Technology

Fault tolerance in distributed computing is a wide area with a significant body of literature that is vastly diverse in methodology and terminology. This paper aims at structuring the area and thus guiding readers into this interesting field. We use a formal approach to define important terms like *fault*, *fault tolerance*, and *redundancy*. This leads to four distinct forms of fault tolerance and to two main phases in achieving them: *detection* and *correction*. We show that this can help to reveal inherently fundamental structures that contribute to understanding and unifying methods and terminology. By doing this, we survey many existing methodologies and discuss their relations. The underlying system model is the close-to-reality asynchronous message-passing model of distributed computing.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; *Reliability, availability, and serviceability*

General Terms: Algorithms, Design, Reliability, Theory

Additional Key Words and Phrases: Asynchronous system, agreement problem, consensus problem, failure correction, failure detection, fault models, fault tolerance, liveness, message passing, possibility detection, predicate detection, redundancy, safety

1. INTRODUCTION

Research in fault-tolerant distributed computing aims at making distributed systems more reliable by handling faults in complex computing environments. Moreover, the increasing dependence of society on well-designed and well-functioning computer systems has led to an increasing demand for *dependable* systems, systems with quantifiable reliability properties. The necessity for such quantification is especially obvious in mission-critical settings like flight control systems or software to control (nuclear) power plants. Until the early 1990s, work in fault-tolerant computing focused on specific technologies and applications, resulting in apparently unrelated subdisciplines with distinct terminologies and methodologies. Despite attempts to structure the field and unify its terminology [Cristian 1991; Laprie 1985], Arora and Gouda [1993] subsequently assessed that "the discipline it-

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg ISIA at Darmstadt University of Technology.

Author's address: Department of Computer Science, Darmstadt University of Technology, Alexanderstraße 10, D-64283 Darmstadt, Germany; email: fcg@acm.org.

CONTENTS

self seems to be fragmented." Hence, people approaching the field are often perplexed, sometimes even annoyed.

Since that time, much progress was made by viewing the area in a more abstract and formal way. This has led to a clearer understanding of the essential and inherent problems in the field and shown what can be done to harness the complexity of systems that counteract faults. This paper draws from these results and uses a formal approach to structure fault-tolerant distributed computing. It concentrates on an important and intensely studied system environment called the *asynchronous system model*. Informally, this is a model in which processors communicate by sending messages to one another delivered with arbitrary delay, in which the speeds of the nodes can get out of synch to an arbitrary extent. We use this model as a starting point because it is the weakest model (i.e., methods for this model work in other models, too) and is arguably the most realistic for distributed computing in today's large-scale wide-area networks.

The purpose of this paper is twofold. First, we want to structure the area clearly, using the latest research results on formalizations of fault tolerance [Arora and Kulkarni 1998a; Arora and Kulkarni 1998b]. For people familiar with the area, this perspective may be unusual and, at first glance, collide with the traditional approaches to structuring the field [Nelson 1990; Cristian 1991; Jalote 1994], but we think the approach here offers insights that complement current taxonomies and contributes to the understanding of fault-tolerance phenomena. This paper can also serve as an introductory tutorial.

Our second aim is to survey the fundamental building blocks that can be used to construct fault-tolerant applications in asynchronous systems. By doing this, we also want to show that formalization and abstraction can lead to more clarity and interesting insights. However, to comply with a tutorial style, this survey tries to argue in an informal way, retaining formalizations only where needed. We assume that the reader has some basic understanding of computers, formal systems, and logic, but not necessarily of distributed systems theory. Note that we are not concerned with security aspects, although they are becoming more and more important and are sometimes discussed under the heading of fault tolerance.

The structure of the paper echoes the two goals. First, we define the relevant terms and the basic system model used throughout (Section 2). Next, we formally define what it means for a system to tolerate certain kinds of faults (Section 3) and derive four basic forms of fault tolerance in Section 4. This leads the way to a discussion of methods for achieving fault tolerance: Section 5 shows that there can be no fault tolerance without redundancy. We briefly revisit system models and argue for the asynchronous system model in Section 6 and then dicuss refined and practical examples of fault-tolerance concepts in

Sections 7 and 8. Finally, Section 9 discusses related work and sketches the current state-of-the-art. Section 10 concludes the paper and outlines directions for future work.

2. TERMINOLOGY

The benefits of fault-tolerance are usually advertised as improving dependability—the amount of trust that can justifiably be put in a system. Normally, dependability is defined in statistical terminology, stating the probability that the system is functional and provides the expected service at a specific point in time. This results in common definitions like the well-known *mean time to failure* (MTTF). While terms like dependability, reliability, and availability are important in practical settings, they are not central to this paper because we focus on the design phase of fault tolerance, not on the evaluation phase. So while the above terms may remain a little imprecise, it is important to define the characteristics of a distributed system precisely. This is done in the following section. For precise definitions of the defining attributes of dependability, see other introductory works; e.g., the book by Jalote [1994].

2.1 States, Configurations, and Guarded Commands

We model a distributed system as a finite set of processes that communicate by sending messages from a fixed message alphabet through a communication subsystem. The variables of each process define its *local state*. Each process runs a local algorithm that results in a sequence of atomic transitions of its local state. Every state transition defines an *event*, which can be a *send* event, a *receive* event, or an *internal* event. No assumptions regarding network topology or message delivery properties of the communication system are made, except that sending messages between arbitrary processes must be possible.

We use *guarded commands* [Dijkstra 1975] as a notation to abstractly represent a local algorithm. A guarded command is a pair consisting of a boolean expression over the local state (called the *guard*) and an assignment that signifies an atomic and instantaneous change of the local state (called the *command*). It is written as $\langle guard \rangle \rightarrow \langle command \rangle$. A guarded command (which for sake of variety is sometimes called an *action*) is said to be *enabled* if its guard evaluates to **true**. A *local algorithm* is a set of guarded commands (in the notation the commands are separated by a '[]' symbol). A process executes a step in its local algorithm by evaluating all its guards and nondeterministically choosing one enabled action from which it executes the assignment. We assume that the choice of actions is fair (meaning that an action that is enabled infinitely often is eventually chosen and executed); this corresponds to *strong fairness* [Tel 1994]. Communication is embedded within the notation as follows: The guard can contain a special **rcv** statement that evaluates to **true** iff the corresponding message can be received. A command may contain one or more **snd** statements that, in effect, send a message to another process.

A *distributed program* (or *distributed algorithm*) consists of a set of local algorithms. The overall system state of such a program, called a *configuration*, consists of all the local states of all processes plus the state of the communication subsystem (i.e., the messages in transit). To make this paper more readable, we often use the terms configuration and state synonymously, and explicitly write "local" state when referring to the local state of a process. All processes have a local *starting state* that defines the *starting configuration* of the system. We do not attempt to distinguish exactly among the terms distributed system, distributed program, and distributed algorithm. While the latter two are used synonymously, the former usually refers to the entirety

```
process Ping
  var $z : \mathbb{N}$ init 0
      $ack$ : boolean init true
  begin
        $\neg ack \wedge \mathbf{rcv}(m)$   $\longrightarrow$   $ack :=$ true; $z := z + 1$
     [] $ack$                             $\longrightarrow$   snd($a$); $ack :=$ false
  end

process Pong
  var $wait$ : boolean init true
  begin
        $\neg wait$                        $\longrightarrow$   snd($m$); $wait := true$
     [] $wait \wedge \mathbf{rcv}(a)$      $\longrightarrow$   $wait := false$
  end
```

Figure 1. A simple distributed algorithm.

of program, state, and execution environment (i.e., hardware).

To help in understanding all these definitions, consider Figure 1, which depicts a simple distributed algorithm using guarded command notation. The algorithm is the well-known "ping-pong" example of two processes that keep alternately sending messages to each other. For example, the local state of process *Ping* consists of the values of the variables z and *ack* (which are initialized to 0 and **true**, respectively). The global state in turn consists of the values of all variables (z, *ack* and *wait*), as well as the set of messages that may be in transit (i.e., a or m). The starting configuration is the global state where $z = 0$, $ack =$ **true**, $wait =$ **true**, and where no messages are in transit. Note that, at every point in time, there is always at most one enabled action in both processes: either it is the process's turn to send or the process is waiting to receive. If a guarded command is enabled, it may be executed, resulting in the change of the local state and equally in the change of the global configuration.

2.2 Defining Faults and Fault Models

There is a considerable ambiguity in the literature on the meaning of some central terms like fault and failure. Cristian [1991] remarks that "what one person calls a failure, a second person calls a fault, and a third person might call an error." The term *fault* is usually used to name a defect at the lowest level of abstraction, e.g., a memory cell that always returns the value 0 [Jalote 1994]. A fault may cause an error, which is a category of the system state. An error, in effect, may lead to a failure, meaning that the system deviates from its correctness specification. These terms are admittedly vague, and here again formalization can help clarification.

Traditionally, faults were handled by describing the resulting behavior of the system and grouped into a hierarchic structure of fault classes or *fault models* [Cristian 1991; Schneider 1993a]. Well-known examples are the *crash* failure model (in which processors simply stop executing at a specific point in time), *fail-stop* (in which a processor crashes, but this may be easily detected by its neighbors), or *Byzantine* (in which processors may behave in arbitrary, even malevolent, ways). System correctness was always proved with respect to a specific fault model. But, unfortunately, slight ambiguities or differences in definition have worked against any common understanding of even simple fault models, and thus more formal methods were developed to describe them.

A formal approach to defining the term "fault" is usually based on the observation that systems change their state as a result of two quite similar event classes: normal system operation and fault occurrences [Cristian 1985]. Thus, a fault can be modeled as an unwanted (but nevertheless possible) state transition of a process. By using additional (virtual) variables to extend the actual state space of a process, every kind of fault from common fault classes can be simulated [Arora 1992; Arora and Gouda 1993; Völzer 1998; Gärtner 1998]. As an example, consider Figure 2, which shows code from one of the processes in Figure 1 that may be subject to crash failures. The occurrence of a crash can be modeled by adding an internal boolean "fault variable" *up* to the process's state, which is initially

```
process Pong that may crash
  var wait : boolean init true
  error up : boolean init true {* virtual error variable *}
  begin
      {* normal program actions: *}
   []  $up \wedge \neg wait$              $\longrightarrow$   snd($m$); $wait := true$
   []  $up \wedge wait \wedge$ rcv($a$)   $\longrightarrow$   $wait := false$
      {* fault actions: *}
   []  $up$                          $\longrightarrow$   $up :=$ false
  end
```

Figure 2. A process that may crash.

true. An additional action $up \rightarrow up := \textbf{false}$, which models the crash, can now be added to the set of guarded commands. To comply with the expected behavior of the crash model, all other actions of the process must be stopped, which can be achieved by adding *up* as an additional conjunct to the guards of these actions. Where virtual error variables may inhibit normal program actions, we call the error variable *effective*. (Note that a subsequent repair action must simply set *up* to **true** again.) The addition of virtual error variables and fault actions can be viewed as a transformation of the initial program into a possibly faulty program [Gärtner 1998]. Investigating and improving the properties of such programs is the subject of fault-tolerance methodologies.

We call the state containing such additional error variables the *extended local state* or the *extended configuration* when referring to the entire system. We therefore define a *fault* as an action on the possibly extended state of a process. A set of faults is called a *fault class*. The advantages of this definition are not only in the clarity of its semantics; but by defining faults in this way, programs susceptible to faults can be reasoned about easily by using the standard techniques aimed at normal fault-free operation. We avoid the term *error* and use the term *failure* to denote the fact that a system has not behaved according to its specification.

2.3 Properties of Distributed Systems: Safety and Liveness

We adopt the usual definitions of system properties proposed by Alpern and Schneider [1985], whereby an *execution* of a distributed program is an infinite sequence $e = c_0, c_1, c_2, \ldots$ of global system configurations. Configuration c_0 is the starting configuration and all subsequent configurations c_i (with $i > 0$) result from c_{i-1} by executing a single enabled guarded statement. Finite executions (e.g., of terminating programs) are technically turned into infinite sequences by infinitely repeating the final configuration. In places where we explicitly refer to finite executions, we call them *partial executions*.

A *property* of a distributed program is a set of system executions. A distributed program always defines a property in itself, which is the set of all system executions that are possible from its starting configuration. A specific property p is said to *hold* for a distributed program if the set of sequences defined by the program is contained in p.

Lamport [1977] reported on two major classes of system properties necessary to describe any useful system behavior: safety and liveness. Informally, a *safety property* states that some specific "bad thing" never happens within a system. This can be characterized formally by specifying when an execution e is not safe for a property p (i.e., not contained in a safety property p): if $e \notin p$, there must be an identifiable discrete event within e that prohibits all possible continuations of the execution from being safe [Alpern and Schneider 1985] (this is the unwanted and irremediable "bad thing"). For example, consider a traffic light that controls the traffic at a road intersection. A simple safety property of this system can be stated as follows: At every point in time no two traffic lights shall show green. The system would not be safe if there were an execution of the system where two distinct traffic lights show green.

A safety property is usually expressed by a set of "legal" system configurations, commonly referred to as *invariant*. By proving that a distributed program is safe, we are assured that the system will always remain within this set of safe states. Thus the safety property unconditionally prohibits the system from switching into configurations not in this set. (In general, if the violation of a property p can be observed in finite time, then p is a safety property.)

On the other hand, a *liveness property* claims that some "good thing" will eventually happen during system execution. Formally, a partial execution of a system is *live* for property p iff it can be extended to still remain in p. A liveness property is one for which every partial execution is live [Alpern and Schneider 1985]. Consider the traffic-light example again. The liveness property could be stated as follows: A car waiting at a red traffic light must eventually receive a green signal and be allowed to cross the intersection. This shows that liveness properties are "eventuality" properties (as opposed to "always" properties like safety). Every partial execution must be live, i.e., the system guarantees that a car driver will eventually cross the street whereby the expected "good thing" happens. Note that this good thing must not be discrete: The liveness property refers to the crossing of *all* cars that arrive at the intersection. Thus, liveness properties can capture notions of progress.

The most common example of liveness in distributed systems is *termination.* Examples where the prescribed "good thing" is not discrete are properties like *guaranteed service* (which states that every request will eventually be satisfied). Liveness properties make no forecast as to when the "good thing" will happen, they only keep it possible. The actual proof that a system satisfies a liveness property usually includes a well-foundedness argument.

```
process from Example 1
  var $x \in \{0,1,2,3\}$ init 1 {* local state *}
  begin
       {* normal program actions: *}
       $x = 1 \longrightarrow x := 2$
    [] $x = 2 \longrightarrow x := 1$
       {* faults that should be tolerated: *}
    [] true $\longrightarrow x := 0$
       {* protection mechanism: *}
    [] $x = 0 \longrightarrow x := 1$
  end
```

Figure 3. A simple fault-tolerant program.

A *problem specification* consists of a safety property and a liveness property. A distributed algorithm A is said to be *correct regarding a problem specification* if both the safety and the liveness property hold for A.

3. A FORMAL VIEW OF FAULT TOLERANCE

Informally, fault tolerance is the ability of a system to behave in a well-defined manner once faults occur. When designing fault tolerance, a first prerequisite is to specify the fault class that should be tolerated [Avižienis 1976; Arora and Kulkarni 1998b]. As we saw in the previous section, this was traditionally done by naming one of the standard fault models (crash, fail-stop, etc.), but is done more concisely by specifying a fault class (a set of fault actions). The next step is to enrich the system under consideration with components or concepts that provide protection against faults from the fault class.

Example 1. Consider the example program in Figure 3. It shows a simple process that keeps on changing its only variable x between the values 1 and 2. To make it fault tolerant, we first have to say which faults it should tolerate. To keep things simple, we consider only a single type of fault specified by the fault action **true** $\rightarrow x := 0$. The next step is to provide a protection mechanism against this type of fault, namely the action $x = 0 \rightarrow x := 1$.

We have now refined our intuition to the point that we can formally define what it means for a system to tolerate a certain fault class.

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

—At any configuration where P holds, T also holds (i.e., $P \Rightarrow T$).

—Starting from any state where T holds, if any actions of A or F are executed, the resulting state will always be one in which T holds (i.e., T is closed in A and T is closed in F).

—Starting from any state where T holds, every computation that executes actions from A alone eventually reaches a state where P holds.

If a program A tolerates faults from a fault class F for invariant P, we say that A is F-tolerant for P.

To understand this definition, recall the program from Example 1. The invariant of the process can be stated as $P \equiv x \in \{1,2\}$, and the fault class to be tolerated as $F = \{\mathbf{true} \rightarrow x := 0\}$. The predicate T is called the *fault span* [Arora and Kulkarni 1998b], and can be understood as a limit to perturbations made possible by the faults from F. In our example, T is equivalent to $x \in \{0,1,2\}$. Definition 1 states that faults from F are handled by knowing the fault span T (see Figure 4, where predicates are depicted as state sets). As long as such faults $f \in F$ occur, the system may leave the set P but will always remain within T. When fault actions are not executed for a sufficiently long period of time, but only normal program actions $t \notin F$, the system will eventually reach P again and resume "normal" behavior.

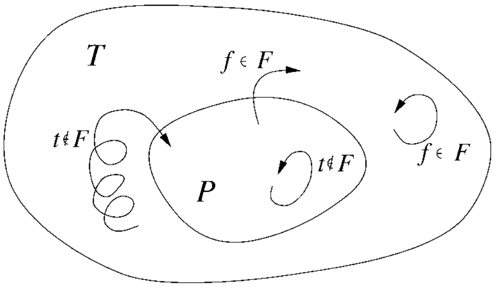

Figure 4. Schematic overview of Definition 1.

Note also that the definition does not refer directly to the problem specification of A. It assumes that the invariant P and the fault span T characterize the program's safety property and that liveness is best achieved within the invariant. (Examples of systems that may not be live starting from states in P are given in the next section.) We therefore call states within P *legal*, states within T but not within P *illegal*, and — because they are not handled — states outside of T *untolerated* (see Figure 5). For instance, the correctness specification of the program in Example 1 could be stated as

(safety) The value of x is always either 1 or 2.

(liveness) At any point in the execution of the program, there is a future point where x will be 1 and there is a future point where x will be 2.

The legal states of the program therefore are those where $x \in \{1,2\}$ and the state $x = 0$ is called illegal. Because the program cannot recover from $x = 3$, this state is called untolerated. In legal states, the program will satisfy safety, whereas safety may or may not be met in illegal states. The reason for not re-

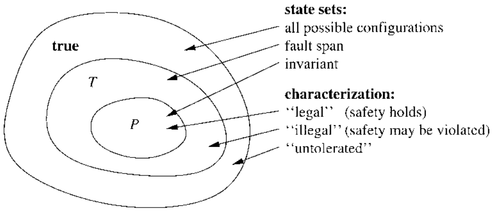

Figure 5. Overview of state sets (with corresponding predicates) and colloquial characterizations.

Table I. Four Forms of Fault Tolerance

	live	not live
safe	masking	fail safe
not safe	nonmasking	none

ferring directly to the correctness specifications of A is to be able to define different forms of fault tolerance, as explained in the next section.

In general, there can be different predicates T that cause A to be fault tolerant. This is an indication that the same fault-tolerance can be achieved by different means. In the above example, if T were specified as $x \in \{0,1,2,3\}$, an additional "recovery" action $x = 3 \rightarrow x := 1$ would be necessary to comply with the definition. As more states are tolerated, this is a distinct fault-tolerance ability.

4. FOUR FORMS OF FAULT TOLERANCE

To behave correctly, a distributed program A must satisfy both its safety and its liveness properties. This may no longer be the case if faults from a certain fault class are allowed to occur. So if faults occur, how are the properties of A affected? Four different possible combinations are shown in Table 1.

If a program A still satisfies both its safety and its liveness properties in the presence of faults from a specified fault class F, then we say that A is *masking* fault tolerance for fault class F. This is the strictest, most costly, and most desirable form of fault tolerance because the program is able to tolerate the faults transparently. Formally, this type is an instantiation of Definition 1, where the invariant P is equal to the fault span T. This means that faults from F and program actions from A cannot lead to states outside of P, thus never violating safety and liveness.

If neither safety nor liveness is guaranteed in the presence of faults from F, then the program does not offer any form of fault tolerance. This is the weakest, cheapest, most trivial, and most undesirable form of fault tolerance. In fact, one shouldn't speak of fault tolerance ability here at all, at least not when referring to fault class F.

Two intermediate combinations exist: one guarantees safety, but is not live (e.g., does not terminate properly), the other may still meet the liveness specification but is not safe. The former is called *fail-safe* fault tolerance and is preferable to the latter whenever safety is much more important than liveness. An example is the ground control system of the Ariane 5 space missile project [Dega 1996]. The system was masking fault tolerance for a single component failure, but was also designed to stop in a safe state whenever two successive component failures occurred [Dega 1996]. For the latter type of faults, the launch of the missile (liveness) was less important than the protection of its precious cargo and launch site (safety). It must be formally possible to divide the fault span into two disjoint sets, L and R. Both state sets are safe, but the system is live only in L. This is guaranteed if faults from a certain fault class F_1 are considered. However, if faults from another, and more severe, fault class F_2 occur, the system may reach states in R that are safe but not live anymore.[1]

[1] This captures the idea of a system being live as long as possible. Of course, liveness is a static

In contrast to masking fault tolerance, we call the latter combination from Table I (which ensures liveness but not safety) *nonmasking*, as the effect of faults are revealed by an invalidation of the safety property. In effect, the user may experience a certain amount of incorrect system behavior (i.e., failures). For example, a calculation result will be wrong or a replication variable may not be up to date [Gärtner and Pagnia 1998]. But at least liveness is guaranteed, e.g., the program *will* terminate or a request *will* be granted.[2] With respect to Definition 1, the fault span T contains states where the safety property does not hold.

Research has traditionally focused on forms of fault tolerance that continuously ensure safety. In particular, masking fault tolerance has been a major area of research [Avižienis 1976; Jalote 1994; Nelson 1990]. This can be attributed to the fact that in most fault-tolerance applications, safety is much more important than liveness.[3]In many cases an invalidation of liveness is also more readily tolerated and observed by the user because no liveness means no progress. In such cases, the user or an operator usually reluctantly (but at least safely) restarts the application.

For a long time nonmasking fault tolerance has been the "ugly duckling" in the field, as application scenarios for this type of fault tolerance are not readily visible (some are given by Arora et al. [1996] and by Singhai et al. [1998]). However, the potential of nonmasking fault tolerance lies in the fact that it is strictly weaker than masking fault tolerance, and can therefore be used in cases where masking fault tolerance is too costly to implement or even provably impossible.

There has recently been much work on a specialization of nonmasking fault tolerance, called *self-stabilization* [Schneider 1993b; Dijkstra 1974] due to its intriguing power: Formally, a program is said to be *self-stabilizing* iff the fault span from Definition 1 is the predicate **true**. This means that the program can recover from *arbitrary* perturbations of its internal state, so that self-stabilizing programs can tolerate *any* kind of transient faults. However, examples show that such programs are quite difficult to construct and verify [Theel and Gärtner 1998]. Also, their nonmasking nature has inhibited them from yet becoming practically relevant.[4]

It should be noted that the liveness properties that are guaranteed in masking and nonmasking fault tolerance are guaranteed only eventually, i.e., the system may be inhibited from making progress as long as faults occur. This can be viewed as a temporal stagnation outside of the invariant but within the fault span. Thus, the system may slow down, but will at least eventually make progress. This can model aspects of a phenomenon known as *graceful degradation* [Herlihy and Wing 1991].

5. REDUNDANCY AS THE KEY TO FAULT TOLERANCE

5.1 Defining Redundancy

When dealing with fault-tolerance properties of distributed systems, a first observation is immediately at hand: No matter how well designed or how fault tolerant a system is, there is always the possibility of a failure if faults are too

property of a piece of code, and so formally A is not live if fault class $F_1 \cup F_2$ is considered.

[2] Nonmasking fault tolerance usually requires the safety property to hold eventually [Arora and Kulkarni 1998b]. It is an open question whether programs that continuously violate safety are of any practical use.

[3] Interestingly, we are more liable to call the fact that a system has not met its safety property a failure than when it invalidates liveness. Note that in the strict sense of a failure, both fail-safe and nonmasking fault tolerances can lead to failures. But since at least one of the two necessary correctness conditions holds, we should speak of a partial failure only.

[4] To our knowledge, Singhai et al. [1998] are the first and only ones to describe a self-stabilizing algorithm implemented in a commercial system.

```
process Redundancy Example
   var $x \in \{0,1,2\}$ init 1 {* local state *}
   begin
        {* normal program actions: *}
        $x = 1 \longrightarrow x := 2$ {* 1 *}
   [] $x = 2 \longrightarrow x := 1$ {* 2 *}
   [] $x = 0 \longrightarrow x := 1$ {* 3 *}
        {* fault action: *}
   [] true $\longrightarrow x := 0$
   end
```

Figure 6. A program with redundancy in space and in time.

frequent or too severe. At first glance this frustrating evidence is an immediate consequence of the topic discussed in this section, i.e., fault-tolerance is always limited in space or time.

The second central observation, which we try to substantiate here, is that to be able to tolerate faults, one must employ a form of redundancy. The usual meaning of redundancy implies that something is there but is not needed because something else does the same thing. For example, in information theory, the bits in a code that do not carry information are redundant. We now define what it means for a distributed program to be redundant and distinguish two forms of redundancy, namely in *space* and in *time*.

Definition 2. A distributed program A is said to be *redundant in space* iff for all executions e of A in which no faults occur, the set of all configurations of A contains configurations that are not reached in e.

Dually, A is said to be *redundant in time* iff for all executions e of A in which no faults occur, the set of actions of A contains actions that are never executed in e.

A program is said to *employ redundancy* iff it either is redundant in space or in time.

To better understand the intuition behind Definition 2, consider the code in Figure 6. The process in Figure 6 is redundant in space because in normal operation the state $x = 0$ is never reached. Redundancy in space refers to the superfluous part of the state of a system, i.e., states never reached when faults do not occur. (A program that is not redundant in space must eventually visit every state in every possible execution.)

On the other hand, redundancy in time refers to superfluous state transitions of a system, i.e., the superfluous work it performs. For example, in Figure 6 action 3 is never executed during normal operation because the state with $x = 0$ is never reached. Thus, according to Definition 2, the program is redundant in time. (A program that is not redundant in time must eventually execute every action in every possible execution.)

The definition of redundancy in space is quite strong. Most programs we write contain states that are never reached simply because (for example) the ranges of variables are not properly subtyped. In fact, every program that does anything useful is redundant in space. This is obvious from a simple example: Imagine a program that first calculates the sum s of two variables x and y (which contain nondeterministic integer values from a range $0 \ldots m$) and then stops. (Nondeterministic values could depend, for example, on the relative ordering of messages received.) The result in s is in the range of $0 \ldots 2m$, and when the sum has been calculated, there are many local states that are never reached during the computation, and thus are redundant. By this argument, one can show that all programs that do some simple form of arithmetics contain unsafe states that may be reached if faults occur. If some of these states are outside the fault span, then the program cannot tolerate these kinds of faults. This is an indication that redundancy alone does not guarantee fault

```
process Non-Redundancy Example
  var $x \in \{0,1,2,3\}$ init 1 {* local state *}
  begin
      $x = 0 \longrightarrow x := 1$ {* 1 *}
    ▯ $x = 1 \longrightarrow x := 2$ {* 2 *}
    ▯ $x = 2 \longrightarrow x := 3$ {* 3 *}
    ▯ $x = 3 \longrightarrow x := 0$ {* 4 *}
  end
```

Figure 7. A program that does not employ redundancy.

tolerance. However, in the next section we see that redundancy is necessary to prevent failures.

5.2 No Fault Tolerance Without Redundancy

The code in Figure 6 from the previous subsection employs redundancy. The objective now is to look at programs lacking any form of redundancy and study their fault-tolerance properties. Figure 7 contains an example of such a program. It shows a process that periodically runs through the states $x = 0,1,2,3,0,1,2,3, \ldots$ forever. What can we say about the fault-tolerance properties of this code?

As a first observation, the reader will notice that the usual invariant of the program, namely $P \equiv x \in \{0,1,2,3\}$, cannot be violated, because in effect it reduces to the predicate **true**. This is because no redundancy in space implies that all states are reached during normal operation of the program. So such a program cannot violate safety properties defined as state sets. However, we see now that even these kinds of programs can still deviate from their specification in the presence of faults. The argument is valid for all kinds of nontrivial computations (we give a short definition of the term "nontrivial computation"; (informally) it rules out trivial programs that do nothing and thus can tolerate any kinds of faults).

We use a very weak notion of a "nontrivial" program here, so that the result is as strong as possible, and use the code in Figure 7 as an example. In a *nontrivial program*, every event on each individual process is assumed to be meaningful to the progress of the computation. So, in order to be correct, the program must satisfy the following specification: starting from an initial configuration, all specified events must occur in a certain prescribed order.[5] For the code from Figure 7, this means that every execution must be of the form 0_0, 1_0, 2_0, 3_0, 0_1, 1_1, 2_1, 3_1, 0_2, 1_2, 2_2, 3_2, 0_3, $\ldots$ (where n_i denotes the event of the i-th occurrence of the state $x = n$). Thus, the program must satisfy the following two properties:

(liveness) For all i in $\mathbb{N}$ and for all n in $\{0,1,2,3\}$, n_i must eventually occur.

(safety) If n_i and m_j are two successive events within the computation, either $i = j$ and $n + 1 = m$ or $i + 1 = j$ and $n = 3$ and $m = 0$.

The central point of this section is formulated in the following claim.

Claim 1. If A is a nontrivial distributed program that does not employ redundancy, then A may become incorrect regarding its correctness specification in the presence of nontrivial faults.

In the context of this claim, a *nontrivial fault* is a fault action that actually changes program variables or effective error variables (recall that error variables are called *effective* if they govern the applicability of program actions). We now argue for this claim using the example code in Figure 7 and its correctness specification above.

We have already seen that the code in

[5] The order can be thought of as a form of intransitive causality relation [Lamport 1978; Schwarz and Mattern 1994]. This notion includes normal sequential programs as well as distributed computations.

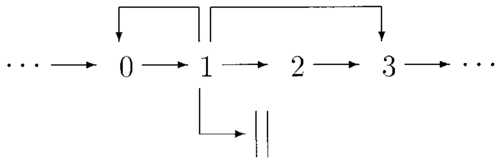

Figure 8. Results of fault actions to a nonredundant program.

Figure 7 does not employ redundancy, and so its set of configurations contains no redundant (and thus possibly illegal) states. So safety properties that are expressible as sets of legal states cannot be violated. Also, all program actions are used during normal executions, so there are no program actions that are merely inhibited from occurring due to error variables within their guard. Our goal is now to show that this program (and analogously the program A from Claim 1) does not satisfy its specification.

The argument is straightforward: Assume that at some point during an execution a nontrivial fault occurs; this is shown for our example program in Figure 8. The configuration in which the fault occurs is one where $x = 1$. Since the fault is nontrivial, it results in a change of the extended system configuration. Here, two cases can occur: (1) the fault changes the normal program variables, or (2) the fault changes any (virtual) error variables. (A third case, in which both program and error variables are changed, can be ignored, since it can be reduced to one of the first two cases.) Let's look at these two cases in turn.

Case 1 (program variables have changed). For the example code, this means that the value of x must have changed, and thus the program finds itself in a new configuration with a different value of x. Let's assume that x was decremented and changed to 0. Now a state where $x = 0$ follows a state where $x = 1$, and thus the safety condition is violated. In general, such changes can cause a program to be "time-warped" into the past and execute events that causally precede events that have long since taken place, thus violating causality.[6]

Something similar occurs if the change to x is an incrementation. If x is set to 2, nobody could distinguish the transition from a normal program action, but we cannot assume that faults are always so benign. So let's assume x was changed to 3. Now the program misses an occurrence of $x = 2$, and thus safety is violated again. In general, this corresponds to a "time warp" into the future that jumps over important events that are necessary for the program to function correctly.[7]

Case 2 (effective error variables have changed). The code in Figure 7 contains neither fault actions nor (virtual) error variables. This makes reasoning about the properties of the program a little complicated, since, in this case, we do not know the concrete effects of changes to error variables. However, we assume that at least one *effective* error variable e has changed. Without loss of generality, let e be a boolean variable that changed from **true** to **false**. Because e is effective, it now inhibits regular program actions from occurring (i.e., those actions that have e as a conjunct in the guard). For example, e might inhibit action 1, action 2, or an arbitrary set of the four actions from Figure 7. It is clear that inhibiting a single action will inhibit progress of the complete process and that this fact is due to the nonredundant nature of the

[6] Imagine the state $x = 1$ as the receipt of a message that the program is about to send when $x = 0$.

[7] Imagine the missed event as the "commit" operation of a transaction that is the prerequisite of the state where $x = 3$.

code. Hence, the liveness condition is violated.

In general, fault actions can result in a degraded functionality of a system (components fail partially or complete processors may crash totally). If the system is nonredundant, the missing functionality may have been vital. Obviously, this may result in system failures if faults happen at the wrong time.

This argument should be sufficient to support Claim 1 (it could have been formally stated as a theorem, but this seemed unnecessary in a tutorial survey). The essential observation from these findings is that redundancy is a necessary prerequisite for fault tolerance.

5.3 Conclusions from the Necessity of Redundancy

The results of investigating nonredundant programs in the previous section are fundamental: While redundancy is not sufficient for fault tolerance, it is a necessary condition and thus must be applied intelligently. However, the argument also reveals another essential problem within fault-tolerant computing: It must be possible for the system to detect that a fault has occured and to "know" about this fact (in the precise sense of knowledge, as defined by e.g., Halpern and Moses [1990]). Detection mechanisms need either state space and/or program actions. As such mechanisms are not really used in fault-free executions, they form part of the system's redundancy.

As we see later in this text, detection mechanisms alone may suffice to ensure safety. But it should already have become clear that on detection of a fault, some action must be taken to correct it if liveness is to be guaranteed. Correction hereby implies detection, and thus is a hint that detection (and thus safety) is easier to achieve than correction (which in effect means liveness). This is promising, since in Section 4 we noted that in practical settings safety is more important than liveness. However, adding detection and correction mechanisms is a complicated matter, because these mechanisms must themselves withstand the faults against which they should protect (the problem of faults occuring in a fault-exception mode). To state it more clearly: detection and correction mechanisms must themselves be fault tolerant.[8]

In practical fault-tolerant systems, redundancy in space is very widespread. It is normally achieved by supplying a component more than once. Practical examples are pervasive in the literature and comprise systems that multiply the software space (e.g. a parity bit in data transmission) or increase the hardware space (e.g. the concept of process pairs residing on different processor boards in the Tandem non-stop kernel [Bartlett 1978]). Of course, the border between hardware and software redundancy is not very sharp [Avižienis 1976]. In fact, hardware redundancy nearly always employs software redundancy because identical components usually run identical programs with identical states (see for example the process group concept of Isis [Birman 1993]). However, hardware redundancy is applied nearly always in fixed multiples of a certain base unit (processor, disk, etc.), whereas software redundancy can also employ fractions (as in error detection codes or the popular RAID systems [Patterson et al. 1988]).

On the other hand, redundancy in time can be thought of as repeating the

[8] *A technical note*: The safety property of the example program of Claim 1 is not expressible as an invariant (i.e., predicate over the system state). However, by adding a variable i that keeps track of the execution, a safety violation can be detected [Lynch 1996]. But now the program may be exposed to faults manipulating i and, by the same argument, there are (new) safety properties (involving i) that are again not expressible as state sets. So if a safety property p is expressible as a set of configurations, "always safe" programs exist. If p is of other forms, we must rely on the fault tolerance of the detection mechanism to ensure safety.

same computation again and again within the same system. Practical examples of this behavior are exhibited by systems that roll back to a consistent state once a fault becomes visible (*roll-back recovery*) or that calculate a result a given number of times to detect transient hardware faults.

Another point of importance in the next section is the distinction between changes in the program's state and changes in the error variables. If a fault changes normal program variables, redundancy makes it possible to detect this change and often also to correct the change. In such cases, redundancy can be seen as a form of "immediate repair." On the other hand, if only error variables change, the fault is somewhat more difficult to detect, since knowledge must be derived from something that is in the worst case *not* present (inhibited actions). This is a serious restriction in some of the widely used system models of distributed computation and is investigated in the next section.

6. MODELS OF COMPUTATION AND THEIR RELEVANCE

To be able to reason formally about properties of distributed systems and to refine our intuition about them, we must abstract those aspects that are unnecessary for investigating a specific phenomenon. We then normally design a set of attributes and associated rules that define how the object in question is supposed to behave. By doing this, we build a model of the object, and the difficulty in doing so is to define a model which is both accurate and tractable [Schneider 1993a].

Because building a model divides important from unimportant issues, there is clearly no single correct model for distributed systems. Even the basic definitions in Section 2 are only one way to model the subject (albeit a widely accepted one). Lamport and Lynch [1990] compare different models with different personal views. But despite this fact, certain aspects of models appear repeatedly in the literature. For example, processes are usually modeled as state machines (or transition systems) that in turn are used to model the execution of larger distributed systems. Other aspects comprise assumptions about the network topology, the atomicity of actions, or the communication primitives available (some key words in this context are shared variables, point-to-point vs. broadcast/multicast transmission, and synchronous vs. aysnchronous communication [Charron-Bost et al. 1996]).

Existing models of distributed systems differ in one particularly important aspect, their inherent notion of real time. This is usually expressed in certain assumptions about process execution speeds and message delivery delays. In *synchronous* systems, there are real-time bounds on message transmission and process response times. If no such assumptions are made, the system is called *asynchronous* [Schneider 1993a; Lamport and Lynch 1990]. Intermediate models that have such bounds to a varying degree are often found [Lynch 1996; Dolev et al. 1987; Dwork et al. 1988]; these are often called *partially synchronous*.

It is well known that the asynchronous model is the weakest one, meaning that every system is asynchronous (thus Schneider [1993a] calls asynchrony a "non-assumption"). Another result of this observation is that every algorithm that works in the asynchronous model also works in all other models. On the other hand, algorithms for synchronous systems are prone to incorrect behavior if the implementation violates even a single timing constraint. This is why the asynchronous model is so attractive and has attained so much interest in distributed systems theory.

Apart from its theoretical attractions, it has been argued [Chandra and Toueg 1996; Babaoğlu et al. 1994; Le Lann 1995; Guerraoui and Schiper 1997] that the asynchronous model is also very realistic in many practical applications. In today's large-scale systems, network congestion and bandwidth shortage, in addition to unreliable components, have contributed to a significant failure rate in reliable message delivery [Golding 1991; Long et al. 1991]. Highly variable workloads on network nodes also make reasoning based on time and timeouts a delicate and error-prone undertaking. All these facts are sources of asynchrony and contribute to the practical appeal of having few or, better, no synchrony assumptions.

However, as mentioned earlier, this model has severe drawbacks, especially for fault-tolerance applications: For example, it can be shown [Chandy and Misra 1986] that in such models it is impossible to detect whether a process has crashed or not. Intuitively, this is a result of allowing processes to be arbitrarily slow [Chandra and Toueg 1996]. This fact is important because we saw in the previous section that detection is the first step towards achieving fault tolerance. As a consequence, solutions to many problems that require *all* processes to participate in a nontrivial way are impossible [Fischer et al. 1985; Chandra et al. 1996; Fischer et al. 1986; Sabel and Marzullo 1995]. But in a positive light, it is commonly said that such impossibility is a blessing to the formal sciences because a lot of work can be done at the fringes [Dolev et al. 1987] of the problem, pushing the edge of the possible further towards the impossible. There are achievable cases [Attiya et al. 1987] in such environments too. We outline the two basic approaches now and consider them in detail in the following two sections.

The two approaches can be characterized as either restricting the system or extending the model. The former way considers only systems and problems in which the existence of solutions is proven [Attiya et al. 1987; Hadzilacos and Toueg 1994; Dolev et al. 1986]. This includes standard methods for traditional (nondistributed) uniprocessor fault tolerance because, as explained earlier, faults that manifest themselves as perturbations of local state variables are usually easily detectable, especially if they occur regularly, and thus can be anticipated.

The second approach deals with extending the model. This means that some synchrony assumptions are added to the model, e.g., a maximum message-delivery delay.[9] This is a sensible approach when the system in question actually guarantees certain timing constraints. This is the case, for example, in real-time operating systems concerned with process response times. Another example is networks of processors that share a common bus for which a maximum message latency can be guaranteed. However, in this approach it is obviously desirable to add as few synchrony requirements as possible and also to add them only to those parts of the system whose correctness depends on them. We meet such modular approaches in the following two sections, which discuss the two essential properties that must be achieved in order to guarantee fault tolerance.

7. ACHIEVING SAFETY

In Section 4 we saw the four distinct forms of fault tolerance based on the presence or absence of the system's safety and liveness properties when faults are allowed to occur. It was also noted that a program's safety property was usually related to the invariant P and the fault span T of Definition 1 (recall Figure 5). In particular, T can contain unsafe configurations, i.e., global states that violate safety. A result of the considerations in Section 5, and especially the argument accompanying Claim 1, is that the first step towards

[9] The original work on partial synchrony [Dwork et al. 1988] assumes that upper bounds to message-delivery delay or relative processor speeds exist, but they are either unknown or only hold eventually. Other prominent models are the *timed asynchronous* model by Cristian and Fetzer [1998] and the *quasi-synchronous* model of Almeida et al. [1998] and Almeida and Veríssimo [1998]. The former mainly assumes a bounded drift rate of local hardware clocks while the latter postulates that only *part* of the network is truly synchronous.

fault tolerance is to acquire knowledge about the fact that a fault has occurred. In this section, we consider basic methods for fault detection and see that in most cases they suffice to ensure safety. Detection mechanisms can therefore be used to implement fail-safe fault tolerance or as a first step towards masking fault tolerance.

7.1 Detection as the Basis for Safety

Confronted with the need to build safe applications, system designers who have read Section 5 may opt for a (at first glance) trivial method to ensure the usual safety properties: The program should not employ redundancy in space. If $T \equiv P \equiv$ **true**, there are no illegal configurations the program can be in, and safety (and thus fail-safe fault tolerance) is trivially always satisfied, no matter what kind of faults occur. This is, however, not so trivial in a distributed setting.

Recall that a system configuration is defined as the collection of all the local states of the processes plus the state of the communication subsystem. Thus, even for small systems, the number of different states that a system can be in is enormous. In fact, if there is no bound on the number of messages that the communication subsystem can buffer, there are infinitely many configurations. This finding, together with the observation from Section 5 that all programs that do some form of arithmetic are redundant in space, contributes to the fact that systems almost always contain configurations that lie outside of P or T, and thus may not satisfy safety. However, T characterizes the set of states reachable by faults from the fault class we want to tolerate, and so we can restrict our attention to states within T.

To continuously ensure safety, we should employ detection and subsequently inhibit "dangerous actions" [Arora and Kulkarni 1998c]. For example, single-bit errors over a transmission line can be detected by a parity bit that is sent along with the message. If we want to tolerate single-bit errors safely, parity is computed and checked against the parity bit. If the fault is detected, the system must be inhibited from doing anything important with the transferred data, e.g., updating a local database or printing the message to the screen. Inhibiting such a dangerous action can be modeled by adding the result of the detection to its guard. An instructive application of this method is known as *fail-stop processors* [Schlichting and Schneider 1983] in which nodes that detect internal faults simply stop working in a way detectable from the outside. Arora and Kulkarni [1998c] view this as eliminating unsafe states from the fault span 1, thus moving T closer to P.

Detection mechanisms such as parity are common in practical systems: prominent generalizations of parity checking are the well-known error detection codes or cryptographic checksums and fingerprints used to guard data integrity. Modules that compare the results of repeated executions of a computation (comparators, voters) are also good examples. Other, broader, notions are exception conditions and acceptance or plausibility tests of computed values.

Taken formally, detection always includes checking whether a certain predicate Q holds over the extended system state. If the type and effect of faults from F are known, it is in general easy to specify Q. Detection is also easy as long as the fault classes are not too severe (because the detection mechanism itself must be fault tolerant) and as long as nondistributed settings are considered. In distributed settings, the detectability of Q depends on different factors and is constrained by system properties, especially if the asynchronous model is chosen (see Section 6).

7.2 Detection in Distributed Settings

In distributed systems without a common time frame, deciding whether a predicate over the global state does or does not hold is in general not easy. One would usually assume a central observer that would watch the execution of a distributed algorithm and instantaneously take action if a given predicate were to hold. But because there is no central lookout point from which to observe the entire system at once, observing distributed computations in a consistent and truthful way is a key issue [Babaoğlu and Marzullo 1993]. In fact, settings can easily be constructed in which two nodes observe the same computation but arrive at different decisions on whether a global predicate held or not [Schwarz and Mattern 1994; Babaoğlu and Marzullo 1993]. This means that in general it makes no sense to ask about the validity of a global predicate without referring to a specific observer or (more generally) to a specific set of observations.

To simplify this matter, Cooper and Marzullo [1991] introduced two predicate transformers called *possibly* and *definitely*.[10]

Definition 3. Let Q be an arbitrary predicate over the system state. The predicate *possibly(Q)* is true iff there exists a continuous observation of the computation for which Q holds at some point. The predicate *definitely(Q)* is true iff for all possible continuous observations of the computation Q holds at some point.

Recall that in a nondistributed setting, one can eliminate an unsafe point from the fault span by adding a detection term to a possibly dangerous action. To ensure safety in a distributed setting, the analogy is that all processes watch out for an unwanted situation specified by Q that may lead to an unsafe state. In terms of the predicate transformers of Definition 3, the system must take action if *possibly(Q)* is detected [Garg 1997].

Algorithms that detect *possibly(Q)* usually take the following approach: Every time a node within the network undergoes a state transition that might affect the validity of Q, it sends a message to a central observer process. The observer process assembles the incoming information in such a way that it has an overview over *all possible* observations. Because it is not known which of these observations is true, this scheme still doesn't suffice to check if a predicate Q *actually* holds. However, there is now enough information to conclude whether the predicate *possibly* holds or not. Note that this scheme must be implemented in a fault-tolerant manner if it is to be of any use in fault detection. In addition, it must be possible to inhibit some process from doing something bad if *possibly(Q)* is detected. This usually requires all processes to wait for an acknowledgment from the observer process.

There are some algorithms that can be used to detect *possibly(Q)* for general predicates [Stoller 1997; Stoller and Schneider 1995; Cooper and Marzullo 1991; Marzullo and Neiger 1991] or restricted ones [Garg and Waldecker 1994; Garg and Waldecker 1996] that consist of a conjunction of so-called *local predicates*. (A predicate q is called *local to process p* iff the truth value of q depends only on the local state of p [Charron-Bost et al. 1995]. See the papers by Schwarz and Mattern [1994] and Chase and Garg [1998] for introductory surveys.) Garg and Mitchell [1998] were the first to consider fault-tolerance issues along with possibility detection. Interestingly, there is a large body of literature on algorithms that have a

[10] The definitions used here are rather informal, but should be sufficient for clarity. Interested readers are referred to background work by Babaoğlu and Marzullo [1993] and Schwarz and Mattern [1994], as well as the initial papers by Cooper and Marzullo [1991] and Marzullo and Neiger [1991].

close resemblance to possibility detection, whose fault-tolerance properties have been studied in great detail. These algorithms usually appear under the heading of "consensus" [Turek and Shasha 1992].

7.3 Adapting Consensus Algorithms for Fault-Tolerant Possibility Detection

The problem of consensus can be viewed as a general form of agreement [Guerraoui and Schiper 1997]. Here, a set of processes (each posesses an initial value) must all decide on a common value. Again, this is trivial in fault-free scenarios, but becomes interesting if processes are faulty and if the asynchronous system model is used.

The similarity between detection of predicates and consensus is visible when a very weak form of consensus is considered in which only one process must eventually decide [Fischer et al. 1985]. Algorithms for this task must diffuse the initial values within the system and collect the information at a single node. Now assume that the initial value of every process is a specific step it wants to take next. Then the central node that collects these values acquires knowledge about the state of all processes and what they want to do next. In effect, the central process acts as an observer that (as in possibility detection schemes) can construct all possible observations. If the other processes are willing to wait for an acknowledgment from the observer before taking their anticipated action, the process can subsequently influence the system.

Of course, this scheme is not very fault tolerant. For example, if the central observer crashes before sending acknowledgments, the system can be blocked forever. Or even worse: if the observer goes haywire by sending arbitrary messages to the other nodes, anything can happen to the system.

The idea of making consensus algorithms fault tolerant is to diffuse information to *all* nodes. If a certain subset of processes is also allowed to behave malevolently, then even more complicated mechanisms are used [Lamport et al. 1982]. However, it should be clear that consensus algorithms need a diffusion phase to be able to work correctly: Information must be received from all correct nodes by all correct nodes in order to arrive at a common decision [Bracha and Toueg 1985; Barborak et al. 1993; Chandra and Toueg 1996]. Thus, in algorithms for stronger versions of the problem, where all correct nodes must eventually decide, every process acts as an observer independently. If the scheme is repeated every time a process step may influence the validity of a predicate Q, a consensus algorithm can be used to detect *possibly(Q)*.

An important result states that consensus is impossible in asynchronous systems where at least one process may crash [Fischer et al. 1985]. However, the problem becomes solvable if some weak forms of synchrony are introduced [Dolev et al. 1987; Fischer et al. 1986; Dwork et al. 1988; Chandra and Toueg 1996]. As we remarked previously, there even exist solutions to the consensus problem if a limited number of nodes behave maliciously and follow the very unfavorable Byzantine fault model [Lamport et al. 1982]. Using such algorithms, detection can be achieved in a very fault-tolerant manner.

7.4 Detecting Process Crashes

Up to now we have implicitly considered predicates that are formed over the system state without virtual error variables. How can predicates that include the latter items be detected in asynchronous systems? We have already noted that in the fully asynchronous model it is impossible to detect a process crash [Chandy and Misra 1986] and have attributed this to the absence of time in this model. Chandra and Toueg [1996] proposed a modular way of extending the asynchronous model to detect process crashes.

In their theory of *unreliable failure detectors*, they propose a program module that acts as an unreliable oracle on the functional states of neighboring processes. The main property of failure detectors is their accuracy: In its weakest form, a failure detector will never suspect at least one correct process of having crashed. This property is called *weak accuracy*. Because weak accuracy is often difficult to achieve, it is often required only that the property eventually holds. Thus, an *eventually weak* failure detector may suspect *every* process at one time or another, but there is a time after which some correct process is no longer suspected. In effect, an eventually weak failure detector may make infinitely many mistakes in predicting the functional states of processes, but it is guaranteed to stop making mistakes when referring to at least one process.

Requiring weak accuracy alone, however, doesn't ensure that a crashed node is suspected at all — it merely prohibits the detection mechanism from wrongly suspecting a correct node. So a second property of failure detectors is necessary. Chandra and Toueg call it *completeness*. Informally, completeness requires that every process that crashes is eventually suspected by some correct process.

It can be shown that different forms of unreliable failure detectors (sometimes simply called *failure suspectors* [Schiper and Riccardi 1993]) are sufficient to solve important problems in asynchronous systems [Chandra and Toueg 1996; Schiper 1997; Schiper and Riccardi 1993]. While there are still unresolved implementation issues [Aguilera et al. 1997a; Chandra and Toueg 1996; Guerraoui and Schiper 1997], these failure detectors simplify the task of designing algorithms for asynchronous systems at large, because they encapsulate the notion of time neatly within a program module.[11] So failure detectors can be used to detect *inhibited actions,* and thus the change of virtual error variables.

7.5 Summary

We have seen that detection is the key to safety: faults must be detected and actions must be inhibited to remain safe. Detection is easy locally, but in distributed settings it is generally difficult and requires intrinisic fault-tolerant mechanisms (like consensus algorithms and failure detectors) to be achieved.

8. ACHIEVING LIVENESS

As explained in Section 4, safety is more important than liveness in many practical situations. But to be truly (i.e., masking) fault tolerant, liveness must be achieved also, and it was noted that though liveness is less important, it is often more difficult to achieve. Detection is sufficient for safety. However, to be live, a fault must not only be detected but also corrected.

8.1 Correction as the Basis for Achieving Liveness

Liveness is tied to the notion of correction as safety is bound to detection. The term correction refers to turning a bad state into a good one, and thus implies a notion of recovery. The hope is that while liveness may be inhibited by faults, subsequent recovery will eventually establish a good state from which liveness will eventually be resumed.

Again, correction is easy in local situations: for example, once a single-bit error over a communication line is detected via parity check, the situation can be corrected by requesting a retransmission of the data. Retransmission is the correction that will lead to a state where the data is received correctly. Broader notions of parity (analogous to error-detection codes) are the

[11] Obviously, models using unreliable failure detectors are no longer truly asynchronous; they merely produce the illusion of an asynchronous system by encapsulating all references to time.

well-known *error-correction codes*. Notions of a local reset also exist and are often called *rollback recovery*. Dually, there also exist methods called *rollforward recovery* by which the system state is forcefully advanced to some future good state. All these methods are examples of actions of correction.

While these methods are easy to design and add in centralized settings, the distributed case is again a little more complicated. In general, on detecting a bad state via a detection predicate Q, the system must try to *impose* a new target predicate R onto the system. This can be seen as a general form of *constraint satisfaction* [Arora et al. 1996] or *distributed reset* [Arora and Gouda 1994].

The choice of the target predicate R depends on the current situation within the system. In general, it is not easy to decide which predicate to choose, since the view of individual processes is restricted and a good choice for one process could be a bad choice for others. The most frequently used method in distributed settings arises from democracy: processes take a majority vote and impose the result of the vote onto themselves. Common examples of voting systems can be found in algorithms that manage multiple copies of a single data item in a distributed system. General methods to keep the copies consistent and to guarantee progress are known as *data replication schemes* [Bernstein et al. 1987]. In systems that enforce strong consistency constraints on data items (*replicas*), a process must often obtain a majority vote from the other processes to be able to update the data item mutually exclusively. Note that the necessity for obtaining a majority ensures safety (here, consistency) and the fact that a process will eventually win a majority guarantees liveness (here, updating progress).

8.2 Correction via Consensus

The notion of correction also corresponds to the decision phase of consensus algorithms (see Section 7). When initial values are received from all functional processes, a node must irrevocably make a decision. This is sometimes done on the first nontrivial value received (for example, if processes can merely crash), or on a majority vote of all processes (if, for example, it is assumed that a certain fraction of processes may run haywire and upset others by sending arbitrary messages). In any case, the decision value is the same in all participating functional processes, and thus can be used as a clue to the next action they wish to take.

An obvious approach to ensuring liveness in distributed systems, which arises from the considerations above, was proposed by Schneider [1990], and is called the *state machine approach*. In it, servers are made fault tolerant by replicating them and coordinating their behavior via consensus algorithms. The idea is that servers act in a coordinated fashion: the servers diffuse their next move and agree on it using consensus. Next moves can be internal actions (for example, updating local data structures) or answers to requests by clients. The fault-tolerance properties of this approach depend on the type of consensus algorithm used.

Other methods that implement fault-tolerant services are based on several forms of fault-tolerant broadcasts [Hadzilacos and Toueg 1994]. It is interesting that a refined version of such communication primitives, called *atomic broadcast*, is closely related to consensus [Hadzilacos and Toueg 1994; Chandra and Toueg 1996]. This is another clue of the importance and omnipresence of consensus, often underestimated in practice, in fault-tolerant distributed computing [Guerraoui and Schiper 1997].

In this section, we briefly surveyed the basic methodologies for achieving liveness in distributed systems. While ensuring safety requires merely inhibiting dangerous actions, guaranteeing liveness requires coordinated correction actions, which are more liable to faults than mere detection. Again, consensus protocols play an important role in asynchronous systems.

8.3 Summary

This section describes fundamental methods for achieving liveness in distributed systems. The basic idea is to impose a state predicate on the system in cases where possibly illegal or unsafe states are detected. Detection is thus a prerequisite to correction. However, to prevent misunderstanding, it should be remarked that the methods presented in the previous two sections do not imply that masking fault tolerance is always possible [Arora and Kulkarni 1998c]. Depending on the fault class in question, there can be faults that lead a system directly into a state where safety is violated. Methods that ensure safety depend on detecting a fault before it can cause transition to an unsafe state. This is obviously not possible for all fault classes. However, it may still be possible to guarantee liveness in such cases [Gärtner and Pagnia 1998; Schneider 1993b]. Thus, safety and liveness in the presence of faults may be achieved both at the same time or one without the other, supporting the utility of the four forms of fault tolerance introduced in Section 4.

9. RELATED AND CURRENT RESEARCH

The idea of dividing fault-tolerance actions into detection and correction phases goes back as far as 1976 to a basic survey by Avižienis [1976]. It is notable that relevant surveys of the area of fault-tolerant distributed computing [Jalote 1994; Siewiorek and Swarz 1992] contain this structure implicitly, without explicitly mentioning it. Lately, Arora and Kulkarni [1998a] have incorporated the idea of detection and correction into a methodology to add fault-tolerance properties to intolerant programs in a stepwise and noninterfering manner (the concept of *multitolerance* [Arora and Kulkarni 1998b]). The four forms of fault tolerance discussed in Section 4 result from their continuing efforts to find adequate formalizations of fault tolerance and related terms [Arora 1992; Arora and Gouda 1993; Arora et al. 1996; Kulkarni and Arora 1997; Arora and Kulkarni 1998b; 1998c; 1998a]. However, their model of computation is based on locally shared variables (i.e., processes have read access to variables of neighboring nodes) and so has rather tight synchrony assumptions that result in elegant solutions to difficult problems [Kulkarni and Arora 1997]. The author is unaware of any attempt to generalize their observations. The present paper is a first step in this direction.

Interest in the topic of consensus in asynchronous distributed systems was very strong during the 1980s [Barborak et al. 1993] and seemed to fade away after Fischer et al. [1985] published their famous impossibility theorem. However, the concept of unreliable failure detectors [Chandra and Toueg 1996] has made practical solutions to this problem a little more feasible, and has also led to a flurry of research on which problems can now be solved with which kind of failure detector [Aguilera et al. 1997b; Sabel and Marzullo 1995; Chandra et al. 1996; Guerraoui et al. 1995; Schiper and Sandoz 1994]. There is strong evidence [Chandra and Toueg 1996] that the various classes of failure detectors capture the notion of time more adequately than the different levels of synchrony that were the subjects of previous research [Dolev et al. 1987; Dwork et al. 1988]. Current work focuses on extending failure suspectors to deal with node behavior other than the crash model [Doudou and Schiper 1997; Oliveira et al. 1997; Aguilera et al. 1998; Dolev et al. 1996; Hurfin et al. 1997]. The ideas in Section 5, dealing with the formalization of redundancy and its consequences have, to the best of our knowledge, not appeared in the literature.

10. CONCLUSIONS AND FUTURE WORK

Despite substantial research effort, the design and implementation of distributed software remains a complex and intriguing endeavor [Schwarz and Mattern 1994; Hadzilacos and Toueg 1994]. However, although distributed systems have many inherent limitations (such as the lack of a global view or common time frame), many problems are easy to solve in ideal and fault-free environments. The possibility of faults complicates the matter substantially, but faults must be handled in order to achieve the dependability and reliability necessary in many practical applications. Therefore, aspects of fault tolerance are of considerable importance today.

In this paper, we have tried to use a formal approach to structure the area of fault-tolerant distributed computing, survey fundamental methodologies, and discuss their relations. We are convinced that continuing attempts to formalize important concepts in this area can reveal many aspects of its underlying structure. This should lead to a better understanding of the subject, and naturally to better, more reliable, and more dependable systems in practice. The advantages of a formal approach, however, also lie in the fact that it reveals the inherent limitations of fault-tolerance methodologies and their interactions with system models. Stating the impossibility of unconditional dependability is as important as trying to build systems that are increasingly reliable.

It is clear that this paper could not integrate the entire area of fault-tolerant distributed computing. Many topics still need further attention, such as aspects of fault diagnosis [Barborak et al. 1993], reliable communication [Hadzilacos and Toueg 1994; Basu et al. 1996b; Basu et al. 1996a], network topology and partitions [Ricciardi et al. 1993; Babaoğlu et al. 1997; Aguilera et al. 1997c; Dolev et al. 1996] and software design faults [Jalote 1994]. We spared the reader probabilistic definitions of central terms like reliability and availability [Jalote 1994] and detailed discussions of implementation issues, test methods (such as fault injection [Hsueh et al. 1997]), and practical example systems [Siewiorek and Swarz 1992; Spector and Gifford 1984; Dega 1996]. All these issues need to be dealt with to see if the structures proposed in this paper can actually coherently organize most of the aspects of the field. The inherent interrelations that this paper reveals (e.g., between consensus and fault-tolerant possibility detection) also need to be pursued further.

Work can continue along other lines as well. For example, it is extremely important to characterize current information technology in terms of models and fault classes (i.e., to know which fault models to follow in which situations). Here, evaluation and simulation of practical systems is a key area [Chandra and Chen 1998; Kuhn 1997]. The effect of such considerations may be to embed practical systems into theoretical models more adequately (and thus more reliably). To this end, the benefits of a formal treatment of the subject can be directly transferred to practical settings, thus easing the task of designing, implementing, and maintaining fault-tolerant distributed systems.

ACKNOWLEDGMENTS

I am indebted to Friedemann Mattern and Henning Pagnia for reading a first draft of this paper and for their suggestions that substantially improved the presentation. Hagen Völzer helped me clarify my understanding of safety and liveness during the final revision phase. I also wish to thank the anonymous referees for their encouraging comments and the Deutsche Forschungsgemeinschaft for making this work possible.

REFERENCES

AGUILERA, M. K., CHEN, W., AND TOUEG, S. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th Interna-*

tional Workshop on Distributed Algorithms (WDAG97, Sept. 1997). 126–140.

Aguilera, M. K., Chen, W., and Toueg, S. 1997. On the weakest failure detector for quiescent reliable communication. Technical Report TR97-1640. Department of Computer Science, Cornell University, Ithaca, NY.

Aguilera, M. K., Chen, W., and Toueg, S. 1997. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report TR97-1632. Department of Computer Science, Cornell University, Ithaca, NY.

Aguilera, M. K., Chen, W., and Toueg, S. 1998. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing* (DISC, Sept. 1998). 231–245.

Almeida, C. and Veríssimo, P. 1998. Using light-weight groups to handle timing failures in quasi-synchronous systems. In *Proceedings of the 19th IEEE Symposium on Real-Time Systems* (Madrid, Spain, Dec.). IEEE Computer Society Press, Los Alamitos, CA.

Almeida, C., Veríssimo, P., and Casimiro, A. 1998. The quasi-synchronous approach to fault-tolerant and real-time communication and processing. Technical Report CTI RT-98-04. Instituto Superior Técnico, Lisboa, Portugal.

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Inf. Process. Lett. 21*, 4 (Oct.), 181–185.

Arora, A. and Gouda, M. 1993. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Trans. Softw. Eng. 19*, 11 (Nov. 1993), 1015–1027.

Arora, A. and Gouda, M. G. 1994. Distributed reset. *IEEE Trans. Comput. 43*, 9 (Sept.), 1026–1038.

Arora, A., Gouda, M., and Varghese, G. 1996. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *J. High Speed Netw. 5*, 3, 293–306.

Arora, A. and Kulkarni, S. 1998. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems* (ICDCS98, May).

Arora, A. and Kulkarni, S. S. 1998. Component based design of multitolerant systems. *IEEE Trans. Softw. Eng. 24*, 1 (Jan.), 63–78.

Arora, A. and Kulkarni, S. S. 1998. Designing masking fault tolerance via nonmasking fault tolerance. *IEEE Trans. Softw. Eng. 24*, 6 (June).

Arora, A. K. 1992. A foundation of fault-tolerant computing. Ph.D. Dissertation. University of Texas at Austin, Austin, TX.

Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. 1987. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science* (Oct. 1987). IEEE Computer Society Press, Los Alamitos, CA, 337–346.

Avižienis, A. 1976. Fault-tolerant systems. *IEEE Trans. Comput. 25*, 12 (Dec.), 1304–1312.

Babaoğlu, Ö., Bartoli, A., and Dini, G. 1994. Replicated file management in large-scale distributed systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms* (WDAG94). Springer-Verlag, Berlin, Germany, 1–16.

Babaoğlu, Ö., Davoli, R., and Montresor, A. 1997. Partitionable group membership: specification and algorithms. Technical Report UBLCS-97-1. Department of Computer Science, University of Bologna, Bologna, Italy.

Babaoğlu, Ö. and Marzullo, K. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. In *Distributed Systems (2nd ed.)*, S. Mullender, Ed. Addison-Wesley Longman Publ. Co., Inc., Reading, MA, 55–96.

Barborak, M., Dahbura, A., and Malek, M. 1993. The consensus problem in fault-tolerant computing. *ACM Comput. Surv. 25*, 2 (June 1993), 171–220.

Bartlett, J. F. 1978. A “NonStop” operating system. In *Proceedings of the 11th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, Los Alamitos, CA.

Basu, A., Charron-Bost, B., and Toueg, S. 1996. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms* (WDAG96). 105–122.

Basu, A., Charron-Bost, B., and Toueg, S. 1996. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609. Department of Computer Science, Cornell University, Ithaca, NY.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

Birman, K. P. 1993. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec. 1993), 37–53.

Bracha, G. and Toueg, S. 1985. Asynchronous consensus and broadcast protocols. *J. ACM 32*, 4 (Oct. 1985), 824–840.

Chandra, S. and Chen, P. 1998. How fail-stop are faulty programs?. In *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-28, June). IEEE Computer Society Press, Los Alamitos, CA, 240–249.

Chandra, T. D., Hadzilacos, V., and Toueg, S. 1996. The weakest failure detector for solving consensus. *J. ACM 43*, 4, 685–722.

Chandra, T. D., Hadzilacos, V., Toueg, S., and Charron-Bost, B. 1996. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (PODC '96, Philadelphia, PA, May 23–26, 1996), J. E. Burns and Y. Moses, Eds. ACM Press, New York, NY, 322–330.

CHANDRA, J. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (Mar.), 225–267.

CHANDY, K. M. AND MISRA, J. 1986. How processes learn. *Distrib. Comput. 1*, 1 (Jan. 1986), 40–52.

CHARRON-BOST, B., DELPORTE-GALLET, C., AND FAUCONNIER, H. 1995. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst. 17*, 1 (Jan. 1995), 157–179.

CHARRON-BOST, B., MATTERN, F., AND TEL, G. 1996. Synchronous, asynchronous, and causally ordered communication. *Distrib. Comput. 9*, 173–191.

CHASE, C. M. AND GARG, V. K. 1998. Detection of global predicates: Techniques and their limitations. *Distrib. Comput. 11*, 4, 191–201.

COOPER, R. AND MARZULLO, K. 1991. Consistent detection of global predicates. *SIGPLAN Not. 26*, 12 (Dec. 1991), 167–174.

CRISTIAN, F. 1985. A rigorous approach to fault-tolerant programming. *IEEE Trans. Softw. Eng. 11*, 1 (Jan.), 23–31.

CRISTIAN, F. 1991. Understanding fault-tolerant distributed systems. *Commun. ACM 34*, 2 (Feb. 1991), 56–78.

CRISTIAN, F. AND FETZER, C. 1998. The timed asynchronous distributed system model. In *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-28, June). IEEE Computer Society Press, Los Alamitos, CA, 140–149.

DEGA, J.-L. 1996. The redundancy mechanisms of the Ariane 5 operational control center. In *Proceedings of the 26th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-26). IEEE Computer Society, New York, NY, 382–386.

DIJKSTRA, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM 17*, 11, 643–644.

DIJKSTRA, E. W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM 18*, 8 (Aug.), 453–457.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan. 1987), 77–97.

DOLEV, D., FRIEDMAN, R., KEIDAR, I., AND MALKHI, D. 1996. Failure detectors in omission failure environments. Technical Report TR96-1608. Department of Computer Science, Cornell University, Ithaca, NY.

DOLEV, D., LYNCH, N. A., PINTER, S. S., STARK, E. W., AND WEIHL, W. E. 1986. Reaching approximate agreement in the presence of faults. *J. ACM 33*, 3 (July 1986), 499–516.

DOUDOU, A. AND SCHIPER, A. 1997. Muteness detectors for consensus with Byzantine processes. Technical report, EPFL. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland.

DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (Apr. 1988), 288–323.

FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

FISCHER, M. J., LYNCH, N. A., AND MERRITT, M. 1986. Easy impossibility proofs for distributed consensus problems. *Distrib. Comput. 1*, 1 (Jan. 1986), 26–39.

GARG, V. K. 1997. Observation and control for debugging distributed computations. In *3rd Int. Workshop on Automated Debugging* (AADEBUG 97, Linköping, Sweden, May). 1–12.

GARG, V. K. AND MITCHELL, J. R. 1998. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems* (ICDCS98, May).

GARG, V. K. AND WALDECKER, B. 1994. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst. 5*, 3, 299–307.

GARG, V. K. AND WALDECKER, B. 1996. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst. 7*, 12, 1323–1333.

GÄRTNER, F. C. 1998. Specifications for fault tolerance: A comedy of failures. Technical Report TUD-BS-1998-03. Darmstadt University of Technology, Darmstadt, Germany.

GÄRTNER, F. C. AND PAGNIA, H. 1998. Enhancing the fault tolerance of replication: another excercise in constrained convergence. In *Digest of Fast Abstracts of the 28th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-28, June). IEEE Computer Society Press, Los Alamitos, CA.

GOLDING, R. A. 1991. Accessing replicated data in a large-scale distributed system. Technical Report UCSC-CRL-91-18. Computer Research Laboratory, University of California, Santa Cruz, CA.

GUERRAOUI, R., LARREA, M., AND SCHIPER, A. 1995. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems* (SRDS95, Bad Neuenahr, Germany, Sept. 1995). IEEE Press, Piscataway, NJ.

GUERRAOUI, R. AND SCHIPER, A. 1997. Consensus: the big misunderstanding. In *Proceedings of the 6th Work-*

shop on Future Trends of Distributed Computing Systems (FTDCS-6, Oct. 1997).

Hadzilacos, V. and Toueg, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425. Department of Computer Science, Cornell University, Ithaca, NY.

Halpern, J. Y. and Moses, Y. 1990. Knowledge and common knowledge in a distributed environment. *J. ACM 37*, 3 (July 1990), 549–587.

Herlihy, M. P. and Wing, J. M. 1991. Specifying graceful degradation. *IEEE Trans. Parallel Distrib. Syst. 2*, 1, 93–104.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. 1997. Fault injection techniques and tools. *IEEE Computer 30*, 4, 75–82.

Hurfin, M., Mostefaoui, A., and Raynal, M. 1997. Consensus in asynchronous systems where processes can crash and recover. Technical Report 1144. IRISA, Rennes Cedex, France.

Jalote, P. 1994. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ.

Kuhn, D. R. 1997. Sources of failure in the public switched telephone network. *IEEE Computer 30*, 4, 31–36.

Kulkarni, S. S. and Arora, A. 1997. Compositional design of multitolerant repetitive Byzantine agreement. In *Proceedings of the 18th International Conference on the Foundations of Software Technology and Theoretical Computer Science* (Kharagpur, India).

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3*, 2 (Mar.), 125–143.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 558–565.

Lamport, L. and Lynch, N. 1990. Distributed computing: models and methods. In *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, 1157–1199.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July), 382–401.

Laprie, J. C. 1985. Dependable computing and fault tolerance: concepts and terminology. In *Proceedings of the 15th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-15, June 1985). IEEE Computer Society Press, Los Alamitos, CA, 2–11.

Le Lann, G. 1995. On real-time and non real-time distributed computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms* (WDAG95, Sept.). Springer-Verlag, Berlin, Germany, 51–70.

Long, D. D. E., Carroll, J. L., and Park, C. J. 1991. A study of the reliability of Internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems* (Pisa, Italy, Sept.). IEEE Press, Piscataway, NJ, 177–186.

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, California.

Marzullo, K. and Neiger, G. 1991. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms* (WDAG91). 254–272.

Mullender, S., Ed. 1993. *Distributed Systems (2nd ed.)*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

Nelson, V. P. 1990. Fault-tolerant computing: fundamental concepts. *IEEE Computer 23*, 7 (July), 19–25.

Oliveira, R., Guerraoui, R., and Schiper, A. 1997. Consensus in the crash-recover model. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland. Technical Report TR-97/239

Patterson, D. A., Gibson, G., and Katz, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the Conference on Management of Data* (SIGMOD '88, Chicago, IL, June 1-3, 1988), H. Boral and P.-A. Larson, Eds. ACM Press, New York, NY, 109–116.

Ricciardi, A., Schiper, A., and Birman, K. 1993. Understanding partitions and the "no partition" assumption. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems* (FTDCS-4).

Sabel, L. S. and Marzullo, K. 1995. Election vs. consensus in asynchronous systems. Technical Report TR95-1488. Department of Computer Science, Cornell University, Ithaca, NY.

Schiper, A. 1997. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput. 10*, 3, 149–157.

Schiper, A. and Riccardi, A. 1993. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-23). IEEE Computer Society Press, Los Alamitos, CA, 534–543.

Schiper, A. and Sandoz, A. 1994. Primary partition "virtually-synchronous communication" harder than consensus. In *Proceedings of the 8th International Workshop on Distributed Algorithms* (WDAG94). Springer-Verlag, New York, 39–52.

Schlichting, R. D. and Schneider, F. B. 1983. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug.), 222–238.

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

Schneider, F. B. 1993. What good are models and what models are good? In *Distributed Systems (2nd ed.)*, S. Mullender, Ed. Addison-Wesley Longman Publ. Co., Inc., Reading, MA, 17–26.

SCHNEIDER, M. 1993. Self-stabilization. *ACM Comput. Surv. 25*, 1 (Mar. 1993), 45–67.

SCHWARZ, R. AND MATTERN, F. 1994. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput. 7*, 149–174.

SIEWIOREK, D. P. AND SWARZ, R. S. 1992. *Reliable computer systems (2nd ed.): design and evaluation*. Digital Press, Newton, MA.

SINGHAI, A., LIM, S.-B., AND RADIA, S. R. 1998. The SunSCALR framework for internet servers. In *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems* (FTCS-28, June). IEEE Computer Society Press, Los Alamitos, CA, 108–117.

SPECTOR, A. AND GIFFORD, D. 1984. The space shuttle primary computer system. *Commun. ACM 27*, 9, 874–900.

STOLLER, S. D. 1997. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (WDAG '97, Sept. 1997), M. Mavronicolas and P. Tsigas, Eds. Lecture Notes in Computer Science, vol. 1320. Springer-Verlag, Berlin, Germany, 185–199.

STOLLER, S. D. AND SCHNEIDER, F. B. 1995. Faster possibility detection by combining two approaches. In *Proceedings of the 9th International Workshop on Distributed Algorithms* (WDAG95, Sept.). Springer-Verlag, Berlin, Germany, 318–332.

TEL, G. 1994. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY.

THEEL, O. AND GÄRTNER, F. C. 1998. On proving the stability of distributed algorithms: self-stabilization vs. control theory. In *Proceedings of the International Computers Conference on Systems, Signals, Control* (SSCC'98, Durban, South Africa, Sept.), V. B. Bajic, Ed. 58–66.

TUREK, J. AND SHASHA, D. 1992. The many faces of consensus in distributed systems. *IEEE Computer 25*, 6 (June), 8–17.

VÖLZER, H. 1998. Verifying fault tolerance of distributed algorithms formally: An example. In *Proceedings of the International Conference on Application of Concurrency to System Design* (CSD98, Fukushima, Japan, Mar. 1998). IEEE Computer Society Press, Los Alamitos, CA, 187–197.

Received: June 1998; revised: January 1999; accepted: January 1999