Fundamentals of Fault Tolerant Distributed Computing in Asynchronous Environments

Felix C. Gärtner
Darmstadt University of Technology

Department of Computer Science
Technical Report TUD-BS-1998-02

July 6, 1998

Abstract

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 of computer science. The paper uses a formal approach to define important terms like *fault*, *fault tolerance* and *redundancy*. This leads to four distinct forms of fault tolerance and to the two main phases in achieving them called *detection* and *correction*. It is shown that this can help reveal fundamental structures inherent in the field that contribute to the understanding and unification of methods and terminology. By doing this, many existing methodologies of this area of computer science are surveyed and their relations are discussed. The underlying system model is the close-to-reality asynchronous message-passing model of distributed computations.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance; 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

---

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.
Address: Department of Computer Science, Alexanderstr. 10, D-64283 Darmstadt, Germany, Email: `fcg@acm.org`

1. INTRODUCTION

Research in fault tolerant distributed computing aims at making distributed systems more reliable by catering for the occurence of faults, which are undesirable but still possible in todays complex computing environments. Moreover, it has always been stressed that the increasing dependance of society on well designed and well functioning computer systems has lead to an increasing demand for *dependable* systems. These are systems that have quantifiable properties regarding their reliability. The necessity for such quantifications is especially obvious in mission critical settings like flight control systems or software to control (nuclear) power plants. Until the early years of the 1990s, work in fault tolerant computing has traditionally 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 itself seems to be fragmented". Hence, people approaching the field are often perplexed, or sometimes even annoyed.

Since that time, much progress has been made by viewing the area in a more abstract and formal way. This has lead to a clearer understanding of the essential and inherent problems of the field and has shown what can be done to harness the complexity of systems that are opposed to faults. This paper draws from these results and uses a formal approach to structure the area of fault tolerant distributed computing. It concentrates on an important and intensely studied system environment called the *asynchronous system model*. Informally spoken, this is a model in which processors communicate by sending messages to one another which are delivered with arbitrary delay, and in which the speeds of the nodes can run adrift and get out of synch to an arbitrary amount of time. 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 it is arguably the most realistic for distributed computing in todays large-scale wide area networks.

The purpose of this paper is twofold: Firstly, we want to clearly structure the area using the latest research results on formalizations of fault tolerance [Arora and Kulkarni 1998a]. For people familiar with the area, this perspective may be unusual and at first glance collide with traditional approaches to structure the field [Nelson 1990; Cristian 1991; Jalote 1994], but we think that the approach presented here offers insightful views that in complement to current taxonomies can contribute further to the understanding of fault tolerance phenomena. This is especially true for readers who are new to the field and so this text can also serve as an introductory tutorial.

Our second aim is to survey the focussed area, namely the fundamental building blocks that can be used to construct fault tolerant applications in asynchronous systems. By doing this, we also want show that formalization and abstraction can lead to more clarity and interesting insights. However, to comply with the style of a tutorial, this survey still tries to argue in an informal way while 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 aspects of security, although they are becoming more and more important and are sometimes also dealt with under the heading of fault tolerance.

The structure of the paper follows the two goals: First of all we introduce and define the relevant terms and the basic system model used throughout this paper (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 to achieve fault tolerance: Section 5 shows that there can be no fault tolerance without redundancy. We briefly revisit the topic of 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 this paper and outlines directions for future work.

2. TERMINOLOGY

The benefits of fault tolerance considerations are usually advertised to improve dependability which measures the amount of trust that can justifiably be put in a system. Normally, dependability is defined using mathematical terminology from statistics, 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. This is because we focus on the design phase of fault tolerance in distributed systems, not on the evaluation phase. So while the above terms may stay a little unprecise, it is important that we define the characteristics of a distributed system precisely. This will be done in the following section. For precise definitions of the defining attributes of dependability, the reader is referred to 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 which 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 which results in a sequence of atomic transitions of its local state. Every state transition defines an *event*, which can be either a *send* event, a *receive* event, or an *internal* event. No assumptions regarding the network topology or the 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 which signifies an atomic and instantaneous change of the local state (called the *command*). It is written in the form $\langle guard \rangle \quad \longrightarrow \quad \langle command \rangle$. A guarded command (which for sake of variation will sometimes also be 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 visually by a '$[\!]$' symbol). A process executes a step in its local algorithm by evaluating all its guards and nondeterministicly choosing one enabled action from which it executes the assignment. We assume that the choice of actions is fair (meaning that an action, which is enabled infinitely often is eventually chosen and executed)[1]. Communication is embedded within the notation as follows: The guard can contain a special **rcv** statement which evaluates to **true** iff the corresponding message can be received. A command may contain one or more **snd** statements which 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 will be called a *configuration* and 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 will often use the terms configuration and state synonymously and will explicitly write "local" state when referring to the local state of a process. All processes have a local *starting state* which together define the *starting configuration* of the system. We will not attempt to distinguish exactly between the terms distributed system, distributed program and distributed algorithm. While the latter two will be used synonymously, the former usually refers to the entirety of program, state and execution environment (i.e. hardware).

```
process Ping
  var z : ℕ init 0
   ack : boolean init true
  begin
        ¬ack ∧ rcv(m)   ⟶   ack := true; z := z + 1
     ▯  ack             ⟶   snd(a); ack := false
  end

process Pong
  var wait : boolean init true
  begin
        ¬wait           ⟶   snd(m); wait := true
     ▯  wait ∧ rcv(a)   ⟶   wait := false
  end
```

Fig. 1. A simple distributed algorithm.

To help understand all these definitions, consider Figure 1 that depicts a simple distributed algorithm using guarded command notation. The algorithm is the well-known "ping-pong" example of two processes that alternately keep sending messages to eachother. 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 $z = 0$, $ack = \mathbf{true}$, $wait = \mathbf{true}$ 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' turn to send

[1]This corresponds to *strong fairness* [Tel 1994].

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] assessed that "what one person calls a failure, a second person calls a fault, and a third person might call an error". Usually, the term *fault* is used to name a "defect" at the lowest level of abstraction, e.g. a memory cell that always returns the value of 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, which means that the system deviates from its correctness specification. These terms are admittedly vague, and here again, formalization can help clarification.

Traditionally, faults have been handled by describing the resulting behavior of the system and have been 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 neighbours) or *Byzantine* (in which processors may behave in arbitrary, even malevolent ways). The correctness of a system was always proved with respect to a specific fault model. But unfortunately, slight ambiguities or differences in definitions have failed to establish a common understanding of even simple fault models, which has resulted in more formal methods to describe these matters.

```
process Pong that may crash
   var wait : boolean init true
   error up : boolean init true {* virtual error variable *}
   begin
         {* normal program actions: *}
      ▯  up ∧ ¬wait              ⟶   snd(m); wait := true
      ▯  up ∧ wait ∧ rcv(a)      ⟶   wait := false
         {* fault actions: *}
      ▯  up                      ⟶   up := false
   end
```

Fig. 2. A process which may crash.

A formal approach to defining the term "fault" is usually based on the observation that systems change their state in result of two quite similar event classes: normal system operation and fault occurences [Cristian 1985]. Thus, a fault can also be modelled 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 the common fault classes can be simulated [Arora 1992; Arora and Gouda 1993]. As an example, consider Figure 2. It shows the code from one of the processes in Figure 1 that may be subject to crash failures. The occurence of a crash can be modelled by adding an internal boolean "fault variable" *up* to the processes state which is initially **true**. An additional action $up \quad \longrightarrow \quad up := \mathbf{false}$ which models the crash can now be added to the set of guarded commands of the process. To comply with the expected behavior of the crash model, all other actions of the process must be inhibited from occuring, which can be achieved by adding *up* as an additional conjunct to the guards of these actions. In such cases 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 of faults actions can be viewed as a transformation of the initial program into a possibly faulty program. Investigating and impoving 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 will be called a *fault class*. The advantages of this definition lie not only in the clarity of its semantics: By defining faults in this way, programs that are suspect to faults can be easily reasoned about using the standard techniques aimed at normal fault-free operation. We will 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 will 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 that are 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. Formally, this can be characterized 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 system executions from being safe [Alpern and Schneider 1985] (this is the unwanted "bad thing"). For example consider a traffic light system that controls the traffic at a road intersection. A simple safety property of this system can be stated as follows: At any point in time no two traffic lights shall show green. The system would be not 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 and proved using an invariance argument. By proving that a distributed program is safe, it is assured that the system will always remain within this set of safe states. The safety property therefore unconditionally prohibits the system to switch into configurations not in this set.

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 legally extended to still remain in p. "Legally" here means that the extension must be allowed by the system itself. A liveness property is one for which every partial execution is live [Alpern and Schneider 1985]. Consider the traffic lights example again. The liveness property of the system 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 "eventual 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.

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 means 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 1998a]. As we have seen in the previous section, this has been traditionally done by naming one of the standard fault models (crash, fail-stop, etc.), but it 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 the faults from the fault class.

Example 1. Consider the example program from 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 will only consider a single type auf fault specified by the fault action $\quad \mathbf{true} \quad \longrightarrow \quad x := 0 \quad$. The next step is to provide a protection mechanism against this type of fault, namely the action $\quad x = 0 \quad \longrightarrow \quad x := 1 \quad$.

Now we have refined our intuition to a level where it is possible to formally define what it means that a system tolerates a certain fault class.

```
process from Example 1
  var x ∈ {0,1,2,3} init 1 {* local state *}
  begin
        {* normal program actions: *}
        x = 1    ⟶   x := 2
     ▯  x = 2    ⟶   x := 1
        {* faults that should be tolerated: *}
     ▯  true     ⟶   x := 0
        {* protection mechanism: *}
     ▯  x = 0    ⟶   x := 1
  end
```

Fig. 3. Example of a simple fault tolerant program.

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 of 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 will 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 $F = \{\ \textbf{true} \longrightarrow x := 0\ \}$. The predicate T is called the *fault span* [Arora and Kulkarni 1998a] and can be understood as a limit to perturbations that are 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 catered for 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. And 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.

Note also that the definition does not directly refer 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[2]. We will therefore call states within P *legal*, states within T but not within P *illegal* and — because they are not catered for — states outside of T *untolerated* (see Figure 5). For instance, the correctness specification of the program of Example 1 could be stated as:

[2]Examples of systems that may not be live starting from states in P are given in the next section.

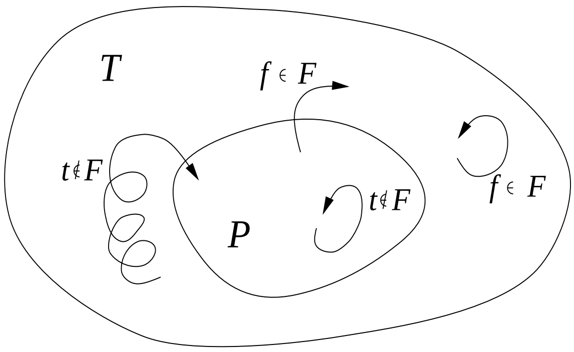

Fig. 4. Schematic overview of Definition 1.

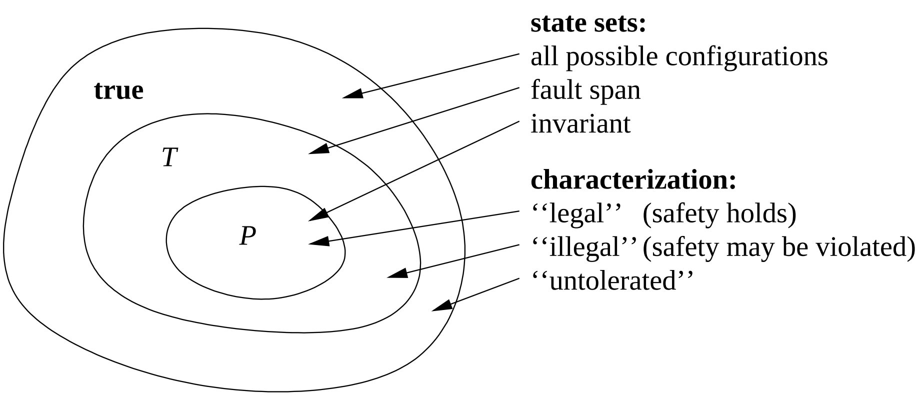

Fig. 5. Overview over state sets (with corresponding predicates) and colloquial characterizations.

(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 referring 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 for the fact that the same fault tolerance ability can be achieved by different means. In the above example, if T was specified as $x \in \{0, 1, 2, 3\}$, an additional "recovery" action $\quad x = 3 \quad \longrightarrow \quad x := 1 \quad$ would have been necessary to comply with the definition. As more states are tolerated, this is a distinct fault tolerance ability.

	live	not live
safe	masking	fail safe
not safe	non-masking	none

Table 1. The four forms of fault tolerance as a result of the properties of a program exposed to faults.

4. FOUR FORMS OF FAULT TOLERANCE

To behave correctly, a distributed program A must satisfy both its safety and its liveness property. This may not be the case anymore if faults from a certain fault class are allowed to occur. So if faults occur, how are the properties of A affected? The four different possible combinations are visualized in Table 1.

If a program A still satisfies both its safety and its liveness property in the presence of faults from a specified fault class F, then we say that A is *masking* fault tolerant for fault class F. This is the strictest, most costly and also most desirable form of fault tolerance since the program is able to transparently tolerate the faults. Formally, this type is an instantiation of Definition 1 from page 6 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 possibly 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 a "fault tolerance" ability here at all, at least not when referring to fault class F.

The two intermediate combinations exist: one guarantees safety, but is not be live (e.g. not terminate properly), the other may still meet their liveness specification but not safe. The former shall be called *fail-safe* fault tolerance and is favourable over the latter whenever safety is much more important than liveness. An example case is the ground control system of the Ariane 5 space missile project [Dega 1996]. The system was masking fault tolerant for a single component failure, but also was designed to stop in a safe state whenever two successive component failures occured [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 the launch site (safety). Formally, it must be possible to divide the fault span into two disjoint sets L and R. Both state sets are safe, but the system is only live 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 a states in R which are safe but not live anymore.[3]

In contrast to masking fault tolerance we will call the latter combination from table 1 (which ensures liveness but not safety) *non-masking*, 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 behavior of the system (i.e. failures). For

[3]This captures the idea of a system being live "as long as possible". Of course, liveness is a "static" property of a piece of code, and so formally A is not live if fault class $F_1 \cup F_2$ is considered.

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. With respect to Definition 1, the fault span T contains states where the safety property does not hold.

Traditionally, research has focussed on forms of fault tolerance that continuously ensure safety. In particular, masking fault tolerance has been the major area of work [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.[4] Also, an invalidation of liveness is in many cases more readily tolerable and observable by the user, because no liveness means no progress. In such cases, the user or an operator usually will reluctantly (but at least safely) restart the application.

Non-masking fault tolerance has for a long time been the "ugly duckling" of the field, as application scenarios for this type of fault tolerance are not readily visible (some are given by Arora, Gouda, and Varghese [1994] and by Singhai, Lim, and Radia [1998]). However, the potential of non-masking fault tolerance lies in the fact that it is strictly weaker than masking fault tolerance and can therefore be applied in cases where masking fault tolerance is too costly to implement or even provably impossible.

It should be noted that recently there has been much work on a specialization of non-masking fault tolerance called *self-stabilization* [Schneider 1993b; Dijkstra 1974], because of 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, the result of which being that self-stabilizing programs can tolerate *any* kinds of transient faults. However, examples show that such programs are quite difficult to construct and to verify [Theel and Gärtner 1998]. Also, their non-masking nature has inhibited them from becoming practically relevant.[5]

It should also be noted that the liveness properties that are guaranteed in masking and non-masking fault tolerance are only eventually guaranteed, i.e. it can happen that the system is inhibited from making progress as long as fault actions 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*.

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 frequent or

[4]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 non-masking fault tolerances can lead to failures. But as at least one of the two necessary correctness conditions holds, we should only speak of a partial failure.

[5]To our knowledge, Singhai, Lim, and Radia [1998] are the first to describe a self-stabilizing algorithm implemented within a commercial system.

too severe. This at first glance frustrating evidence is an immediate consequence of the topic discussed in this section, i.e. that fault tolerance abilities are always limited in space or time.

The second central observation which we will try to substantiate here is that for being able to tolerate faults one must employ a form of redundancy. The usual meaning of redundancy points to the fact that something is there although it 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 will now define what it means for a distributed program to be redundant and will 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 the set of all configurations of A contains configurations that are not reached in any execution where no faults occur.

Dually, A is said to be *redundant in time* iff the set of actions of A contains actions that are never executed in any execution where no faults occur.

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

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

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

To better understand the intuition behind Definition 2 consider the code from 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.

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.

The definition of redundancy in space is quite strong. Most programs we write contain states which are never reached simply because for example the ranges of variables are not properly subtyped. In fact, every program that does anything useful will be 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 certain range $0 \ldots m$) and then stops.[6] The result in s will be in the range of $0 \ldots 2m$ and after calculating the sum, there will be 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 of the fact that redundancy alone does not guarantee fault tolerance. However, in the next section we will see that redundancy is necessary to prevent failures.

5.2 No Fault Tolerance without Redundancy

The code in Figure 6 from the previous subsection employed redundancy. The objective now is to look at programs that do not use 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?

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

Fig. 7. A program that does not employ redundancy.

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 and therefore are safe. However we will see now that even these kinds of programs can still deviate from their specification in the presence of faults if they have no redundancy in time. The argument is valid for all kinds of non-trivial computations. We will define the term "non-trivial computation" shortly. 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 "non-trivial" program here so that the result is as strong as possible and will use the code from Figure 7 as an example. In a *non-trivial 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 liveness property: starting from an initial con-

[6]Nondeterministic values could depend for example on the relative ordering of messages received.

figuration, all specified events must occur in a certain prescribed order.[7] In terms of the code from Figure 7 it 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 (liveness) properties:

(L1) For all i in $\mathbb{N}$ and for all n in $\{0, 1, 2, 3\}$, n_i must eventually occur.

(L2) 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 $n = 2$ and $m = 0$.

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

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

In the context of this claim a ***non-trivial fault*** is a fault action that actually changes program variables or effective error variables[8]. We will now argue for this claim using the example code from Figure 7 and its liveness specifications from above.

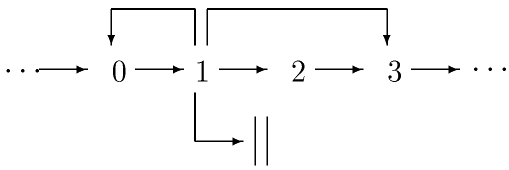

Fig. 8. Results of fault actions to a non-redundant program.

We have already seen that the code from Figure 7 does not employ redundancy and so its set of configurations contains no redundant (and thus possibly illegal) states. So safety cannot be violated. Also, all program actions are used during normal executions, so there are no program actions that are merely inhibited from occuring due to error variables within their guard. Our goal is now to show that this program (and in analogy the program A from Claim 1) does not satisfy its liveness property.

The argument is straightforward: Assume that at some point during an execution a non-trivial fault occurs. In Figure 8 this is visualized for our example program. The configuration in which the fault occurs is one where $x = 1$. As the fault is non-trivial, 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 where both program and

[7]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.

[8]Recall that error variables are called *effective* if they govern the applicability of program actions.

error variables are changed can be ignored, since it is reducible 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 liveness condition L2 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.[9]

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 occurence of $x = 2$ and thus L2 is again violated. 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.[10]

Case 2 (effective error variables have changed): The code from Figure 7 does neither contain fault actions nor (virtual) error variables. This makes reasoning about the properties of the program a little complicated as we do not know the concrete effects of changes to error variables in this case. However, we assume that at least one *effective* error variable e has changed. Without loss of generality let e be a boolean variable which changed from **true** to **false**. Because e is effective it now inhibits regular program actions from occuring (i.e. those actions that have e as a conjunct in the guard). For example, e might inhibit action 1, action 2 or an abribtrary 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 non-redundant nature of the code. Hence, L1 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 non-redundant, the missing functionality may have been vital. This may obviously result in system failures if faults happen at the wrong time.

This argument should be sufficient to support Claim 1.[11] The essential observation from these findings is that redundancy is a necessary prerequisite for fault tolerance.

5.3 Conclusions from Needing Redundancy

The results from investigating non-redundant programs in in the previous subsection are fundamental: While redundancy is not sufficient for fault tolerance, it is a necessary condition and thus must be applied intelligently. However, the argumentation also reveals another essential problem within fault tolerant computing: It must be possible for the system to detect that a fault has occured and thus "know" about this fact (in the precise sense of knowledge as defined for example by Halpern

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

[10] Imagine the missed event being the "commit" operation of a transaction that is the prerequsiste of the state where $x = 3$.

[11] The argument could have been formally stated as a theorem but this seemed unnecessary within a tutorial survey.

and Moses [1990]). Detection mechanims either need state space and/or program actions. As they are not really used in fault free executions, they form part of the systems redundancy.

As we will 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 shall be guaranteed. Correction hereby implies detection and is thus 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.

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 which reside 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 (like processor, disk, etc.), whereas software redundancy can also employ fractions (like 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 doing the same computation again and again within the same system. Practical examples for this kind of behavior are exhibited by systems that roll back to a consistent state once a fault has become visible (*rollback recovery*) or that calculate a result a given number of times to detect transient hardware faults.

Another point that will be 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 which is in the worst case *not* present (inhibited actions). This is a serious restriction in some of the widely used system models of distributed computations and shall be 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 from 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 will clearly be no single correct model for distributed systems. Even the basic definitions of Section 2 are only one way to model the subject matter (although a widely accepted one). Lamport and Lynch [1990] therefore compare different models with different personal views. But despite this fact, there are certain aspects of models that appear repeatedly in the literature. For example, processes are usually modeled as state machines (or transition systems) which 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 which are available (some keywords in this context are shared variables, point-to-point vs. broadcast/multicast transmission and synchronous vs. aysnchronous communication [Charron-Bost et al. 1996]).

One particularly important aspect in which existing models of distributed systems differ is 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]. Often intermediate models are found that have such bounds to a varying degree [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 (Schneider [1993a] thus calls asynchrony a "non-assumption"). Another result from this observation is that every algorithm which works in the asynchronous model will also work 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 theoretical attraction 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 application settings. In todays large scale systems, network congestion and bandwidth shortage in addition to unreliable components have contributed to a significant failure rate concerning reliable message delivery [Golding 1991; Long et al. 1991]. Also, highly variable workloads on network nodes 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 little or better no synchrony assumptions.

However, as mentioned earlier there are severe drawbacks with using this model especially when considering fault tolerance applications: For example it can be shown [Chandy and Misra 1986] that in such models it is impossible to detect if 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 so 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 non-trivial way are impossible [Fischer et al. 1985; Chandra et al. 1996; Fisher et al. 1986; Sabel and Marzullo 1995]. But seen in a positive light, there is a common saying that such impossibility results are a blessing to formal sciences because a lot of work can be done at the fringe [Dolev et al. 1987] of the problem, pushing the edge of the possible further towards the impossible without ever reaching it. And there are achievable cases [Attiya et al. 1987] in such environments too. We will outline the two basic approaches now and reconsider 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 has been proven [Attiya et al. 1987; Hadzilacos and Toueg 1994; Dolev et al. 1986]. This includes standard methods for traditional (non-distributed) 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. This is a sensible approach whenever the system in question actually guarantees certain timing constraints. This is the case for example in real-time operating systems concerning response time of processes. Another example are 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 little synchrony requirements as possible and also to add them only to those parts of the system whose correctness depends on them. We will meet such modular approaches in the following two sections which discuss the two essential properties which must be achieved in order to guarantee fault tolerance.

7. ACHIEVING SAFETY

In Section 4 we have come across the four distinct forms of fault tolerance based on the presence or non-presence 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 related to the invariant P and the fault span T of Definition 1 (for this, recall Figure 5 on page 8). In particular, T can contain "unsafe" configurations, i.e. global states that violate safety. A result from the considerations in Section 5 and especially the argument accompanying Claim 1 was that the first step towards fault tolerance was to aquire knowledge about the fact that a fault has occured. In this section we will 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 Achieving Safety

Confronted with the necessity to build safe applications, system designers, who have read Section 5 may opt for an (at first glance) trivial method of how to ensure safety: The program should not employ redundancy in space. If $T \equiv P \equiv \mathbf{true}$, there are no illegal configurations that 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 if a distributed setting is considered.

Recall that a system configuration was 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 arithmetics are redundant in space) contributes to the fact that systems will practically always contain configurations which 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.

The idea of continuously ensuring safety is to employ detection and subsequently inhibt "dangerous actions" from occuring [Arora and Kulkarni 1998b]. For example, single bit errors over a transmission line can be detected by a parity bit which is sent along with the message. If we want to tolerate single bit errors safely, the 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], where nodes that detect internal faults simply stop working in a way detectable from the outside. Arora and Kulkarni [1998b] view this as eliminating unsafe states from the fault span and thus moving T closer to P.

Detection mechanisms such as parity are commonly met in practical systems: prominent generalizations of parity checking are the well-known error detection codes or cryptographic checksums and fingerprints that are used to guard data integrity. Modules that compare the results from 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 over the extended system state holds. 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 non-distributed 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, answers to the question whether a predicate over the global state does hold or does not hold, are in general not easy. Usually, one would assume a central observer that would watch the execution of a distributed algorithm and instantaneously take action if a given predicate holds. But because there is no central lookout point from which someone could observe the entire system at once, the problem of observing distributed computations in a consistent and "truthful" way is a key issue [Babaoğlu and Marzullo 1993]. In fact, settings are easily constructable where two nodes observe the same computation but arrive at different decisions 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 for 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*.[12]

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 non-distributed setting, one could 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.

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 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 acknowledgement from the observer process.

There exist a couple of 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] which 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 paper by Schwarz and Mattern [1994] for an introductory survey.) However, none of them considers fault tolerance issues. Interestingly, there exists a large body of literature on algorithms that have a close resemblence with possibility detection and the fault tolerance properties of which have been studied to a great detail. Usually, these algorithms run 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]. In it, a set of processes (that each posess an initial value) must all decide on a common value. Again, this is trivial in fault-free scenarios, but

[12] The definitions brought here are rather informal but should be sufficient for the understanding. 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].

it becomes interesting if processes may be 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, where 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 aquires knowledge about the state of all processes and what they want to do next. In effect the central process acts as an observer that (like in possibility detection schemes) can construct all possible observations. If the other processes are willing to wait for an acknowledgement from the observer before they take 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 acknowledgements, 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 behave malevolently, then mechanisms that are even more complicated 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; Fisher et al. 1986; Dwork et al. 1988; Chandra and Toueg 1996]. As remarked above, there even exist solutions to the consensus problem if a limited number of nodes behave maliciously and follow the very unfavourable 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 the 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 contributed this to the "absence" of time in this model. Chandra and Toueg [1996] have proposed a modular way of extending the asynchronous model to be able 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 neighbouring 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*. As weak accuracy is often difficult to achieve, it is often only required 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 not suspected anymore. In effect, an eventually weak failure detector may make infinitely many mistakes on 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], they simplify the task of designing algorithms for asynchronous systems by large, because they encapsulates the notion of time neatly within a program module.[13] Thus, failure detectors can be used to detect *inhibited actions* and thus the change of virtual error variables.

7.5 Summary

To conclude this section, 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 detection is generally difficult and requires intrinisic fault tolerant mechanisms (like consensus algorithms and failure detectors) to be achieved. But sometimes, safety is not the key issue but liveness.

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 also be achieved. It was also noted that though liveness is less important, it is often more difficult to achieve. To be safe, detection sufficed. However, to be live, apart from detecting a fault, it must also be corrected.

8.1 Correction as the Basis for Achieving Liveness

Liveness is tied to the notion of correction as is safety bound to detection. The term correction refers to turning a bad state into a good state 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 is eventually resumed.

[13]Obviously, models using unreliable failure detectors are not truely asynchronous anymore. They merely produce an illusion of an asynchronous system by encapsulating all references to time.

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. This retransmission is the correction action that hopefully will lead to a state where the data is correctly received. Broader notions of parity (that are analogous to error detection codes) are the well known *error correction codes*. Also, notions of a local reset exist that are often called *rollback recovery*. Dually, there also exist methods called *rollforward recovery* where the system state is forcefully advanced to a certain "good" state in the future. All these methods are examples of correction actions.

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. 1994] 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 met method in distributed settings lends itself 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 the data items (*replicas*), a process often has to obtain a majority vote from the other processes to be able to mutually exclusively update the data item. Note that the necessity to take the majority hurdle ensures safety (here: consistency) and the fact that eventually a process will win the 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 decide. This is sometimes done on the first non-trivial 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 hint to the next action they wish to take.

An obvious approach to ensuring liveness in distributed systems which results from the above considerations was proposed by Schneider [1990] and 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 to update 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 hint to the importance and omnipresence of consensus in fault tolerant distributed computing which is often underestimated in practice [Guerraoui and Schiper 1997].

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

8.3 Summary

This section contained a description of fundamental methods to achieve liveness in distributed systems. The basic idea is to impose a state predicate onto the system in cases where possibly illegal or unsafe states are detected. Detection is thus a prerequisite to correction. However, to prevent a 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 1998b]. 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 ensured safety depended on the fact that a fault could be detected before it could 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 usefullness 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 [1998b] have incorporated the idea of detection and correction into a methodology to add fault tolerance properties to intolerant programs in a stepwise and non-interfering manner (the concept of *multitolerance* [Arora and Kulkarni 1998a]). The four forms of fault tolerance discussed in Section 4 are a 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 1998a; Arora and Kulkarni 1998b]. However, their model of computation is based on locally shared variables (i.e. processes have read access to variables of neighbouring nodes) and thus has rather tight synchrony assumptions which results in relatively easy solutions to in general difficult problems [Kulkarni and Arora 1997]. The author is unaware of any attempft to generalize their observations. The currect paper is a first step in this direction.

Interest in the topic of consensus in asynchronous distributed system has been very strong during the 1980s [Barborak et al. 1993] and seemed to fade away after Fischer, Lynch, and Paterson [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. It has also lead to a flurry of research activity on what problems can now be solved with which kind of failure detector [Aguilera et al. 1997b; Sabel and Marzullo 1995; Chandra et al. 1996]. There is strong evidence [Chandra and Toueg 1996] that the different classes of failure detectors capture the notion of time in a more adequate way than different levels of synchrony that were subject of previous research [Dolev et al. 1987; Dwork et al. 1988]. Current work focusses on extending failure suspectors to be able to deal with other node behavior apart from the crash model [Doudou and Schiper 1997; Oliveira et al. 1997; Aguilera et al. 1998; Dolev et al. 1996]. The ideas of Section 5 with the formalization of redundancy and its consequences have to the best of our knowledge not appeared in the literature and are considered novel.

10. CONCLUSIONS AND FUTURE WORK

Despite substantial research efforts, the design and implementation of distributed software remains a complex and intriguing endeavour [Schwarz and Mattern 1994; Hadzilacos and Toueg 1994]. But although there are many inherent limitations of distributed systems (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 they must be catered for in order to achieve the dependability and reliability which is necessary in many practical applications. Therefore, fault tolerance aspects 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 the continuing attempt to formalize important concepts of the area can reveal many aspects of the underlying structure inherent in the topic. In effect, this should lead to a better understanding of the subject matter and naturally to better, more reliable and more dependable systems in practice. The advantages of a formal approach however lie also 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 despite its considerable size, this text could not integrate the entire area of fault tolerant distributed computing. Many aspects still need further attention. These topics cover 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]. Also, we have spared the reader from probabilistic definitions of central terms like reliability and availability [Jalote 1994] and detailed discussions of implementation issues, test methods (such as fault injection), and practical example systems [Siewiorek and Swarz 1992; Spector and Gifford 1984; Dega 1996]. All aspects need to be dealt with to see if the structure proposed in this paper actually can coherently organize most of the aspects of the field. The inherent interrelations that it 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 adequatly 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]. The effect of such considerations can 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 matter can be directly transferred to practical settings, thus easing the task of designing, implementing, and maintaining fault tolerant distributed systems.

Acknowledgements

I am gratefull to Friedemann Mattern and Henning Pagnia for reading a first draft of this paper and for giving vitally helpfull suggestions that substantially improved the presentation.

REFERENCES

Aguilera, M. K., Chen, W., and Toueg, S. 1997a. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320* (Sept. 1997), pp. 126–140.

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

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

Aguilera, M. K., Chen, W., and Toueg, S. 1998. Failure detection and consensus in the crash-recovery model. Technical Report TR98-1676 (April), Cornell University, Computer Science Department.

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Information Processing Letters 21*, 181–185.

Arora, A. and Gouda, M. 1993. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering 19*, 11, 1015–1027.

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

Arora, A., Gouda, M. G., and Varghese, G. 1994. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. In *Proceedings of the 14th International Conference on Distributed Computing Systems* (1994), pp. 424–431.

Arora, A., Gouda, M. G., and Varghese, G. 1996. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks 5*, 3, 293–306.

Arora, A. and Kulkarni, S. S. 1998a. Component based design of multitolerance. *IEEE Transactions on Software Engineering 24*, 1, 63–78.

Arora, A. and Kulkarni, S. S. 1998b. Designing masking fault tolerance via nonmasking fault tolerance. *IEEE Transactions on Software Engineering*. (to appear).

Arora, A. K. 1992. *A foundation of fault-tolerant computing*. Ph. D. thesis, The University of Texas at Austin.

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 Sym-*

posium on the Foundations of Computer Science (Los Alamitos, CA, USA, Oct. 1987), pp. 337–346. IEEE CS Press.

AVIŽIENIS, A. 1976. Fault-tolerant systems. *IEEE Transactions on Computers 25*, 12 (Dec.), 1304–1312.

BABAOĞLU, O. AND MARZULLO, K. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. MULLENDER Ed., *Distributed Systems* (Second ed.)., Chapter 4, pp. 55–96. Addison-Wesley.

BABAOĞLU, ÖZALP., BARTOLI, A., AND DINI, G. 1994. Replicated file management in large-scale distributed systems. In *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857* (1994), pp. 1–16.

BABAOĞLU, ÖZALP., DAVOLI, R., AND MONTRESOR, A. 1997. Partitionable group membership: specification and algorithms. Technical Report UBLCS-97-1 (Jan.), Department of Computer Science, University of Bologna, Italy. Revised May 1997.

BARBORAK, M., DAHBURA, A., AND MALEK, M. 1993. The consensus problem in fault-tolerant computing. *ACM Computing Surveys 25*, 2 (June), 171–220.

BARTLETT, J. F. 1978. A “NonStop” operating system. In *Proceedings of the 11th Hawaii International Conference on System Sciences*, Volume 3 (1978).

BASU, A., CHARRON-BOST, B., AND TOUEG, S. 1996a. Simulating reliable links with unreliable links in the presence of process crashes. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486* (1996), pp. 105–122.

BASU, A., CHARRON-BOST, B., AND TOUEG, S. 1996b. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609 (Sept.), Cornell University, Computer Science Department.

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

BIRMAN, K. P. 1993. The process group approach to reliable distributed computing. *Communications of the ACM 36*, 12, 36–53.

BRACHA, G. AND TOUEG, S. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM 32*, 4 (Oct.), 824–840.

CHANDRA, S. AND CHEN, P. 1998. How fail-stop are faulty programs? In *Proceedings of the 28th Symposium on Fault Tolerant Computing Systems (FTCS-28)* (June 1998), pp. 240–249.

CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1996. The weakest failure detector for solving consensus. *Journal of the ACM 43*, 4 (July), 685–722.

CHANDRA, T. D., HADZILACOS, V., TOUEG, S., AND CHARRON-BOST, B. 1996. On the impossibility of group membership. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, USA, May 1996), pp. 322–330. ACM.

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (March), 225–267.

CHANDY, K. M. AND MISRA, J. 1986. How processes learn. *Distributed Computing 1*, 40–52.

CHARRON-BOST, B., DELPORTE-GALLET, C., AND FAUCONNIER, H. 1995. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems 17*, 1 (Jan.), 157–179.

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

COOPER, R. AND MARZULLO, K. 1991. Consistent detection of global predicates. *ACM SIGPLAN Notices 26*, 12 (Dec.), 167–174.

CRISTIAN, F. 1985. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering 11*, 1 (Jan.), 23–31.

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

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

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

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

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

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

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

DOUDOU, A. AND SCHIPER, A. 1997. Muteness detectors for consensus with Byzantine processes. Technical report, EPFL – Départment d'Informatique, Lausanne, Switzerland.

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

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

FISHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1986. Easy impossibility proofs for distributed consensus problems. *Distributed Computing 1*, 26–39.

GARG, V. K. AND WALDECKER, B. 1994. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems 5*, 3, 299–307.

GARG, V. K. AND WALDECKER, B. 1996. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems 7*, 12, 1323–1333.

GÄRTNER, F. C. AND PAGNIA, H. 1998. Enhancing the fault tolerance of replication: another excercise in constrained convergence. In *Digest of FastAbstracts of the 28th Symposium on Fault Tolerant Computing Systems (FTCS-28)* (June 1998).

GOLDING, R. A. 1991. Accessing replicated data in a large-scale distributed system. Master's thesis, University of California, Santa Cruz. Also published as Technical Report UCSC-CRL-91-18.

GUERRAOUI, R. AND SCHIPER, A. 1997. Consensus: the big misunderstanding. In *Proceedings of the 6th Workshop 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 (May), Cornell University, Computer Science Department. An earlier version appeared in the collection edited by Sape Mullender [Mullender 1993].

HALPERN, J. Y. AND MOSES, Y. 1990. Knowledge and common knowledge in a distributed environment. *Journal of the ACM 37*, 3 (July), 549–587.

JALOTE, P. 1994. *Fault tolerance in distributed systems*. Prentice-Hall, Englewood Cliffs, NJ, USA.

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* (1997).

LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (March), 125–143.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July), 558–565.

LAMPORT, L. AND LYNCH, N. 1990. Distributed computing: models and methods. In *Handbook of Theoretical Computer Science (Volume B: Formal Models and Semantics)*, Chapter 18, pp. 1157–1199. Elsevier. J. van Leeuwen, Editor.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July), 382–401.

LAPRIE, J. C. 1985. Dependable computing and fault tolerance: concepts and terminology. In *FTCS-15, 15th Symposium on Fault Tolerant Computing Systems* (June 1985), pp. 2–11.

LE LANN, G. 1995. On real-time and non real-time distributed computing. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972* (Sept. 1995), pp. 51–70.

LONG, D. D. E., CARROLL, J. L., AND PARK, C. J. 1991. A study of the reliability of Internet sites. In *SRDS91 Proceedings of the 10th Symposium on Reliable Distributed Systems* (Sept. 1991), pp. 177–186.

LYNCH, N. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

MARZULLO, K. AND NEIGER, G. 1991. Detection of global state predicates. In *WDAG91 Distributed Algorithms 5th International Workshop Proceedings, Springer-Verlag LNCS:579* (1991), pp. 254–272.

MULLENDER, S. Ed. 1993. *Distributed Systems* (Second ed.). Addison-Wesley.

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. Technical Report TR-97/239 (Aug.), EPFL – Département d'Informatique, Lausanne, Switzerland.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)* (June 1988), pp. 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)* (1993).

SABEL, L. S. AND MARZULLO, K. 1995. Election vs. consensus in asynchronous systems. Technical Report TR95-1488 (Feb.), Cornell University, Computer Science Department.

SCHIPER, A. 1997. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing 10*, 3, 149–157.

SCHIPER, A. AND RICCARDI, A. 1993. Virtually-synchronous communication based on a weak failure suspector. In *FTCS-23, 23rd Symposium on Fault Tolerant Computing Systems* (1993), pp. 534–543.

SCHLICHTING, R. D. AND SCHNEIDER, F. B. 1983. Fail stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1*, 3 (Aug.), 222–238.

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

SCHNEIDER, F. B. 1993a. What good are models and what models are good? In S. MULLENDER Ed., *Distributed Systems* (Second ed.)., Chapter 2, pp. 17–26. Addison-Wesley.

SCHNEIDER, M. 1993b. Self-stabilization. *ACM Computing Surveys 25*, 1, 45–67.

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

SIEWIOREK, D. AND SWARZ, R. 1992. *Reliable Computer Systems: Design and Evaluation.* Digital Press.

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

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

STOLLER, S. D. 1997. Detecting global predicates in distributed systems with clocks. In M. MAVRONICOLAS AND P. TSIGAS Eds., *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, Number 1320 in Lecture Notes in Computer Science (Sept. 1997), pp. 185–199. Springer-Verlag.

STOLLER, S. D. AND SCHNEIDER, F. B. 1995. Faster possibility detection by combining two approaches. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972* (Sept. 1995), pp. 318–332.

TEL, G. 1994. *Introduction to Distributed Algorithms*. Cambridge University Press.

THEEL, O. AND GÄRTNER, F. C. 1998. On proving the stability of distributed algorithms: self-stabilization vs. control theory. In *Proceedings of SSCC 98, Durban, South Africa* (1998). To appear.

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