Distributed consensus and hard real-time systems

Citation for published version (APA):

Document status and date:
Published: 01/01/1994

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Distributed Consensus
and
Hard real-Time Systems
by
Dick Alstein
94/40

ISSN 0926-4515
All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 94/40
Eindhoven, September 1994
Distributed Consensus
and
Hard Real-Time Systems

Dick Alstein

September 8, 1994
Abstract

The first part gives an overview of the literature on Distributed Consensus, focusing on deterministic algorithms that tolerate benign failures. These algorithms are intended for lock-step synchronous systems, which have easy timing characteristics. The second part shows that algorithms presented in the first part, can be used in practical hard real-time systems. This is made possible by a simulation algorithm. We determine under what conditions a simulation is possible, and what its efficiency is.
Chapter 1

Introduction

When constructing algorithms for distributed systems, the most important difficulty that must be faced is the lack of global knowledge. Processors only have immediate access to their own state, and when they need to know the state of other processors, this information must be sent to them. Since the transmission usually takes some time, the knowledge of the global state may not be up-to-date, since the other processors' states may have changed during the transmission.

Distributed Consensus can be seen as a formalization of this problem. Each processor is assumed to hold a private initial (numerical) value, which is at first unknown to the other processors. The processors have to reach a common decision, based on the initial values. When a processor decides, its decision is irrevocable. Whatever the behavior of the faulty processors, the decisions of the correct ones must be unanimous.

The Distributed Consensus problem has long been regarded as fundamental to fault-tolerant distributed computing. Consequently, it has been studied extensively and in many variants, resulting in a huge amount of publications.

The goal of this paper is twofold. The first is to provide the reader with an overview of the subject, which is presented in chapter 3 — after introducing the necessary definitions in chapter 2. The overview does not try to cover the whole area in the same amount of detail: it concentrates on benign (i.e. non-Byzantine) failures, and on deterministic algorithms. The main reason for this restriction is that we are interested in applying Consensus algorithms in a real-time system. Randomized algorithms have an unpredictable (worst-case) termination time, and therefore do not lend themselves very well for this purpose. Algorithms that tolerate Byzantine failures, though very robust, have a low efficiency: typically, less than a third of the processors may fail, and execution takes a number of messages that is exponential in the number of processors involved.

In chapter 4 we discuss the suitability of algorithms from the literature for application in practical hard real-time systems such as DEDOS [SRL+94]. The difficulty in this is that said algorithms are often designed for an abstract distributed system. The timing characteristics of such a system (e.g. lock-step synchrony) do not match exactly with realistic distributed systems. We distinguish several classes of synchrony, called timing models, and determine if a system with this timing can simulate the abstract model.
Chapter 2

Basic definitions and models

The first section of this chapter contains definitions for the basic concepts. The second section introduces the notion of failures, and gives a (rather informal) failure classification. The next section provides a more formal model of computations and failures. The final section of this chapter contains the notations and assumptions that are used throughout this paper.

2.1 Basic concepts

A distributed system is a computer system consisting of a number of independent processing elements, called processors, cooperating to perform a common task. A processor has access to an amount of local memory, which can not be accessed by other processors.

For information exchange, the processors are connected to a communication network. Through this network the processors send and receive information packets, or messages. For the network topology, the way that processors are interconnected, there are myriads of variations possible. In this paper, we view the network as complete: there is a link between every pair of processors. Note that this is only an abstraction. In practice, there may be a communication layer that transmits messages with the help of intermediate processors.

Naturally, we demand of any computer system that the result of a computation is logically correct. But in a real-time system, it is required not only that the value of the result is according to some specification, but also that the time at which this result is delivered lies in a given interval. In practice, this means that a computation has a certain deadline: a time before which the computation must be completed. If the consequences of violating a deadline are very severe, we speak of a hard real-time (HRT) computation; in such a case, deadlines must be met at all cost. The other case is called soft real-time (SRT). Here, violating a deadline means that the usefulness of the result is reduced (possibly to zero or a negative value).

In real-time systems, the timing constraints that must be satisfied are stated in terms of global time. By global time, we mean a time base that is a linear function of physical time. In order to satisfy these timing constraints, a processor must have some way of measuring the passing of time. Each processor is therefore equipped with a real-time clock. ¹ A processor can read the time from its clock, but in most systems this time is not equal to global time. Due to hardware imperfections, it may deviate from global time. Also, in many systems, the clock value is generated locally, at each processor. Therefore, the clock values may differ

¹Here we use the word clock to indicate this real-time clock. It should not be confused with the machine clock, the piece of hardware that determines the pace at which a CPU executes machine instructions.
between processors. To keep the differences small, the clocks must be synchronized. This is usually done by a separate clock synchronization algorithm, through which each processor periodically compares its clock value with others and adapts its clock speed.

2.2 Failures and failure classification

Every component of the system, be it a processor or a communication link, is expected to function according to some specification. This specification will probably prescribe its logical behavior, i.e. the responses that it gives to the inputs that it gets. The specification can – and in real-time systems it should – also prescribe its timing behavior, i.e. the time at which the responses are given. Whenever, during an execution, the behavior does not meet the specification, we say that a failure has occurred; the component is faulty. We classify failures according to the way in which the behavior deviates from the specification:

- If the component omits to respond to an input, we call it an omission failure. The omission can be a send omission, where the processor generates a response internally but fails to send it, or it can be a receive omission, where the processor fails to receive the input.

- If the component at a certain time omits a response, and from that time on omits every response, it has experienced a crash failure.

- If the response is either omitted or given at a moment that is not according to specification, then this is a timing failure. The response can be too early (early timing failure), or it can be too late (late timing failure).

- If the component gives an arbitrary incorrect response, we call this a Byzantine failure. This means that it may not only omit the response or give it at a wrong time, but that the value of the response can also be wrong. The behavior may even be steered by some malevolent intelligence, trying to confuse the correct components.

- A subclass of the latter are authenticated Byzantine failures. Authentication is the ability for a component to append a non-forgeable signature to its output. When another component sees the response, it can check if this output was indeed given by the component. This means that when faulty components pass on a value from another component, they can still change the value, but the falsification can be detected by other components. Public-key cryptography systems are a good example of such an authentication method.

The above classification was introduced by Cristian (see e.g. [Cri91]). The classes are divided into benign failures (crash, omission and timing) and malicious (Byzantine and authenticated Byzantine). The benign failures have in common that a faulty component keeps on calculating its response, but may fail to deliver it correctly. However, if another component receives a response, then it can be sure that the value was obtained according to the program. A maliciously faulty component may give an arbitrary response, and may even work according to some malicious plan, in order to evoke erroneous behavior from the other components. The failure classification forms a scale of failures of increasing severity, which is depicted in Figure 2.1. From left to right, the failure classes cover a wider and wider range of behavior;
a crash failure is a special case of an omission failure, any omission failure can also be seen as a timing failure, etc.

Another important distinction is between transient and permanent failures. When we view a failure as transient, the duration of the faulty behavior is short in relation to the total time of the execution. A faulty component will soon stop its incorrect output, and resume normal behavior. An example of transient failures is a communication channel that occasionally drops a message; if it resumes correct operation, its output can be used again.

With permanent failures, we consider 'faulty' to be a stable property of a component: once it has failed, it remains faulty for the rest of the execution. Its output should be ignored, to prevent it from causing disorder in the rest of the system. A crash failure is an obvious example of a permanent failure.

These two views lead to different methods for treating failures. Transient failures can be countered by using retries (time redundancy) or by applying self-stabilizing algorithms (e.g. [Dij74]). Permanent failures can be treated with information redundancy or component redundancy (see e.g. [AL90]). When using information redundancy, extra data is added so that 'damaged' information can be reconstructed (by applying error-correcting codes). With component redundancy, the component is replicated, and the outputs of the replicas are combined by voting or comparing. In practice, a system should utilize methods of all kinds, in order to obtain an optimal resilience to failures. Note also that methods for treating transient failures assume that either an incorrect output can be detected (so we can start a retry), or that a temporarily incorrect output is acceptable (so we can wait until the system has stabilized again). Methods for permanent failures perform masking, i.e. they prevent the incorrect output from influencing the system.

We are interested in constructing algorithms that are fault-tolerant: an algorithm should meet its requirements despite the occurrence of failures. Usually, it is not possible to tolerate any number and any type of failure. A measure for how well we have succeeded is resilience, which is defined in the next section.

2.3 Algorithms and executions

In this section, we give a computation model. Since this paper focuses on real-time systems, it is important that we are able to verify that computations satisfy both logical and timing constraints. It is therefore inevitable to reason about computations as they take place in global time. The first model sees computations as timed executions, being sequences of configurations (system states) alternated by timed events. In some cases this model is too elaborate. If there is a certain amount of synchrony in the system, we do not have to deal
explicitly with local clocks and real time. The second model is a simplification of the first, introducing the notion of computation rounds.

2.3.1 Basic computation model

A processor is a state machine over a finite set of states. For communication, each processor has a receive buffer, which is an ordered set of messages. The messages are taken from a finite alphabet. Two actions can be executed on a receive buffer: Send and Receive. By Send\((p,m)\) a processor appends messages \(m\) to the receive buffer of the destination processor \(p\). The message is at first undeliverable, i.e. \(p\) can not access it. After some time it becomes deliverable. When a processor executes a Receive, it reads all deliverable messages, thereby removing them from the receive buffer.

The state of the system or configuration consists of all processor states and the contents of the receive buffers. The configuration can only be changed by an event. There are two types of events:

- In a computation event, a processor can receive the set of deliverable messages, change its state, and send a number of messages. This is also called a processor step.

- A delivery event makes a message that resides in a receive buffer deliverable.

We do not view failures as an event of some kind; failures are said to occur when computation or delivery events do not have the expected result. (A proper definition of failures is given below.)

A timed event is an ordered pair consisting of an event and a time. The time is the instant in global time at which the event takes place.

Definition 2.1 An execution \(E\) is a sequence of configurations \(C\), alternated by timed events \((e_i,t_i)\):

\[
E = C_0 (e_1, t_1) C_1 (e_2, t_2) C_2 \ldots
\]

The timed events appear in nondecreasing time order: \(t_k \leq t_{k+1}\) \((k \in \mathbb{N}, t_j \in \mathbb{R})\). Configuration \(C_0\) is called the initial configuration of the execution.

So far, there has been no mention of local clocks. Instead of being added explicitly to the model, they are made part of the processor state. Each processor has a counter, a variable that is initialized to 0 at the start of an execution and incremented by 1 at each computation event. Thus, ticks of the local clock and processor steps are equivalent. If we say that a processor \(p\) takes a certain action at local time \(k\), it means that this is done in the \(k^{th}\) computation event of \(p\). Ideally, local time progresses at the same speed as global time: if a processor had a perfect clock, it would take the next step after exactly one unit of global time. But in real life clocks are never perfect. A more realistic approach is to define a function to prescribe, for each processor step, the time interval within which the step should be taken.

Definition 2.2 A timing model is a pair of monotonously nondecreasing functions \(\tau_e : \mathbb{N} \mapsto \mathbb{R}\) and \(\tau_l : \mathbb{N} \mapsto \mathbb{R}\), where for each \(k \in \mathbb{N}\) : \(0 \leq \tau_e(k) \leq \tau_l(k)\).

The functions \(\tau_e\) and \(\tau_l\) indicate the earliest and latest global time at which a processor step can be taken. In other words, if a processor satisfies a timing model \((\tau_e, \tau_l)\) then it takes its \(k^{th}\) step at some point in the time interval \([\tau_e(k), \tau_l(k)]\).
The definition of an execution is fairly loose. For instance, one could construct executions in which an infinite number of events happen at the same global time. However, in the following we will only consider legal executions, executions that satisfy a number of restrictions. Some restrictions are obvious, and must hold for all timing models:

- In the initial configuration, all receive buffers are empty. A Send action always precedes its corresponding delivery event.

- Processor steps take a certain minimum amount of time\(^2\): there is a constant \(\phi\) such that for every processor \(p\), if \((e_a, t_a)\) and \((e_b, t_b)\) are two computation events of \(p\), then \(|t_b - t_a| > \phi\).

Other restrictions may or may not hold, depending on the timing model. For example, if there is a bound \(\delta\) on message delay, then for every message, the time between the computation event in which it was sent and the delivery event in which it was made deliverable is smaller than \(\delta\). Another assumption commonly made is that messages are not duplicated or generated spontaneously: for every Send action, there is at most one corresponding delivery event, and for every delivery event, there is exactly one Send action.

For computation events, a processor naturally is not free to determine its state change and message output. This is prescribed by the algorithm it is running:

**Definition 2.3** A distributed algorithm is a specification for each processor \(p_i\) of a function

\[
\alpha_i : \mathcal{P}(M) \times S \times \mathbb{N} \rightarrow \mathcal{P}(M) \times S
\]

where \(S\) is the set of processor states and \(M\) is the message alphabet.

In this definition, the meaning of \(\alpha_i(m, s, k) = (m', s')\) is that if at local time \(k\) \(p_i\) is in state \(s\) and receives a set \(m\) of messages, it should change its state to \(s'\), sending out the messages in set \(m'\). Note that the state change is deterministic; we do not consider randomized algorithms here. Also, set \(m'\) contains at most \(n\) messages: since the message size is not a priori limited, there is no need to send more than one message to the same processor at the same time. In some cases the size of \(m'\) is restricted to one: a processor can only send a message to one processor at a time.

An execution in which every processor is supposed to execute algorithm \(A\) is called a run of \(A\). In some proofs, we will use the fact that two runs of an algorithm are indistinguishable.

**Definition 2.4** Two runs are indistinguishable to a certain processor \(p_i\) iff in the two runs

1. the state of \(p_i\) in the initial configurations is equal,
2. the computation events of \(p_i\) occur at the same global times, and
3. the same sets of messages are sent to \(p_i\) and delivered at the same global times.

\(^2\)This lower time bound for processor steps is justified by the existence of a minimum switching time for transistors.
correct step \( \text{iff } \alpha_i(m, s, k) = (m', s') \land t_i(k) \leq t \leq r_i(k) \)
receive-omission failure \( \text{iff } \exists l: (m', s') = (\alpha_i(l, s, k)) \)
send-omission failure \( \text{iff } \exists l': (m', s') = (\alpha_i(m, s, k)) \)
omission failure \( \text{iff } \sigma \text{ is a send-omission or receive-omission failure} \)
crash failure \( \text{iff } \sigma \text{ is a send-omission failure and } \sigma \text{ is the last step of } p_i \)
early timing failure \( \text{iff } t_i(k) < t \)
late timing failure \( \text{iff } t > r_i(k) \) or \( \sigma \) is an omission failure
timing failure \( \text{iff } \sigma \text{ is an early or late timing failure} \)
Byzantine failure \( \text{iff } \sigma \text{ is not a correct step} \)

Table 2.2: Definition of failures in a processor step \( \sigma \)

The state changes prescribed by the algorithm only depend on a processor’s local state, the local time and the incoming messages. Thus, if two runs are indistinguishable to \( p_i \) and \( p_i \) is correct, then we can conclude that \( p_i \) goes through the same state changes.

With the above definitions, the failure classes from the previous section can be defined in a more formal way. First, fix some algorithm \( A = \{ \alpha_i \} \) to specify the logical behavior of the processors, and a timing model \( (\tau, \tau) \) to specify their timing behavior. Let \( R \) be a run of \( A \), and let \( \sigma \) be the \( k \)th step of processor \( p_i \) in \( R \), taken at global time \( t \). The step changes \( p_i \)'s state from \( s \) to \( s' \) and sends a set of messages \( m' \). Let \( m \) be the contents of \( p_i \)'s receive buffer in the configuration preceding \( \sigma \). Thus, we have an observed behavior (input conditions of \( (m, s, k) \), output conditions of \( (m', s') \)) which should be checked against the behavior specified by \( \alpha_i \) and \( (\tau, \tau) \). The definition of the failure types is listed in Table 2.2.

Definition 2.5 A processor is correct in a run \( R \) iff it only takes correct steps in \( R \). Otherwise it is faulty in \( R \).

The ultimate goal of running an algorithm is, of course, to satisfy the requirements posed by a certain problem. This problem will usually be stated as a set of of predicates, containing configurations and global time as variables. We say that a run solves the problem iff the predicates evaluate to true.

Definition 2.6 Let \( F \) be a failure class and \( k \geq 0 \). A run \( R \) is a \( k \)-\( F \)-run iff there are at most \( k \) processors faulty in \( R \), and every failure is of type \( F \).

Definition 2.7 Let \( P \) be a problem \( P \), let \( F \) be a failure class and \( k \geq 0 \). We say that \( A \) is a \( k \)-\( F \)-resilient algorithm for \( P \) if every \( k \)-\( F \)-run of \( A \) solves \( P \).

2.3.2 Round-based models

The above model is intended to serve as a general execution model. For some systems however, it is needlessly complicated. An alternative is the round-based or lock-step synchronous model.

Definition 2.8 A system is lock-step synchronous iff an execution consists of rounds, where each round consists of
2.4 NOTATIONS AND DEFAULT ASSUMPTIONS

- A send phase, in which the processors can send messages to other processors.
- A receive phase, in which the processors receive messages. Messages between correct processors are sent and received in the same round.
- A state change phase, in which the processors take a step, based on the messages received in the previous phase.

The rounds are usually numbered consecutively, starting at 1.

This model has a rather high degree of synchrony, as it is guaranteed that messages, exchanged by correct processors, are sent and received in the same round. A big advantage of this model is that reasoning about executions is relatively simple, and that many results from the literature are based on this model.³

In some cases we will prove that a certain model can simulate a lock-step synchronous system. Such a simulation is an algorithm that divides the processing time into rounds. The current round number is kept as a counter. The simulation tells a processor

1. when the round counter should be increased
2. when a message for a certain round should be sent
3. whether any overhead data should be attached to outgoing messages (e.g. timestamp or current round number)
4. how incoming messages should be processed, based on the overhead data

A simulation is correct if it can be proved that whenever a correct processor receives a message from another correct processor, it is in the same local round that the sender was in when it sent the message.

A variation on this is the eventually lock-step synchronous system. In some cases, it may not be possible to simulate a lock-step synchronous system. However, there may be simulations in which processors operate in a round-based way, where there is no guarantee that messages always arrive. Message loss may occur even for transmission between correct processors, but that can only happen in the first rounds. After a certain round, transmission is reliable.

Definition 2.9 A system is eventually lock-step synchronous if it is lock-step synchronous, with the exception that there is an integer g such that in rounds 1, 2, ..., g-1, messages between correct processors can be lost.

2.4 Notations and default assumptions

This section lists the notations that are used throughout this paper, as well as the default assumptions.

There are n processors, \( p_1, p_2, \ldots, p_n \). Communication is failure-free: every message that has been sent will eventually arrive at its destination, and the message contents are not corrupted during transmission.⁴ Processor \( p_i \) has a local clock, with clock function \( C_i \).

³In some publications this model is simply called synchronous, but the term ‘synchronous’ has become highly polluted. For clarity, we have added the word ‘lock-step’.
⁴This assumption is of course not very realistic. It should be seen as a working hypothesis; we assume that communication failures are masked by an underlying layer of system services, which also masks any failures.
Processor failures are assumed to be permanent: the processors are divided into correct ones, that operate correctly during the entire execution, and faulty processors, which may experience a failure at any time. This implies that we do not consider methods that implement recovery of some kind (e.g. rebooting of processors, reconfiguring the system or self-stabilizing algorithms). The type of failure varies, but we concentrate on benign failures. It is assumed that the number of faulty processors in any execution is bounded. This bound is denoted by \( f \).
Chapter 3

A short history of Distributed Consensus

The Distributed Consensus problem has its origins in the field of distributed databases, where it was first known as the Interactive Consistency problem or Byzantine Agreement problem. The problem situation is as follows: there are a number of data managers (processors), each managing a copy of a replicated data item. Basically, a processor can modify the data item by sending an update message to all processors, but failures may cause problems. For example, if the processor that generates such a set of updates crashes while sending, the data managers do not all get the update, so replicas get different values. For this reason, the managers cannot simply process the updates that they receive. They must agree which updates are to be processed and which ones should be ignored. This problem is further enhanced by the possibility of a failure of data managers. In the worst case, a processor can fail maliciously, and send conflicting messages (e.g. reporting the receipt of an update to one processor while denying the fact to another).

This problem became known as the Byzantine Generals problem [LSP82]. The story: a Byzantine army camps outside an enemy city. The army consists of \( n \) divisions, each commanded by a general. To avoid a catastrophic defeat, all divisions must take the same action, which can either be to attack or retreat. The commander of the army can send a message to his generals, containing 'attack' or 'retreat'. The generals can exchange messages, in order to reach agreement on the action to be taken. However, some generals may be traitors, and may try to confuse the other, loyal ones. Even the commander himself can be a traitor, trying to lure the army into defeat. But if the commander is loyal, all loyal generals must obey his command.

In a more abstract way: there is a sender that is supposed to send the same message to all processors. The processors must take a decision, which is irreversible, and the decisions of correct processors must be unanimous.

**Definition 3.1** A sender \( s \) sends a message \( M \) to a set of \( n \) processors. The processors must take a decision, which is irreversible. An algorithm solves Byzantine Agreement iff it satisfies

1. **Agreement:** Whenever two correct processors decide, their decisions are the same.
2. **Validity:** If the sender is correct, all decisions of correct processors are equal to \( M \).
3. **Termination:** All correct processors eventually decide.
Note that the failures are not necessarily Byzantine. For easier failure classes, the problem becomes simpler, but certainly not trivial.

The Byzantine Agreement problem is also known as the Reliable Broadcast problem. We now switch focus to Consensus, which is only slightly different from Reliable Broadcast. One reason to concentrate on Consensus instead of Broadcast is that the former problem seems to be somewhat easier. This connection is further discussed in section 3.5.

In Reliable Broadcast, the sender is the only source of basic information: he must transmit his message. By contrast, in Distributed Consensus the basic information is distributed among all the processors. This information is kept by each processor as its initial value, and is not known to the other processors. The goal is to bundle all these values into a group decision, or in other words to reach consensus.

**Definition 3.2** Each processor $p_i$ has an initial value $v_i$, taken from a totally ordered set $V$. An algorithm solves Strong Consensus iff it satisfies

1. Agreement: Whenever two correct processors decide, their decisions are the same.
2. Validity: The decision of a correct processor is equal to the initial value of some processor.
3. Termination: All correct processors eventually decide.

Although several variants of the Consensus problem are known, the above definition seems to be the most widely accepted. We will view this as the standard definition for Consensus.

Some remarks can be made about the formulation:

- The database example at the start of this section suggests that a decision is binary. The above definition, however, only demands that the values be taken from a totally ordered set. Indeed, in some cases this set can be $\mathbb{R}$ [DLP+86]. When $|V|$ is finite we speak of Finite-valued Consensus; if $V = \{0, 1\}$ it is called Binary Consensus. The two problems are equivalent, but this is not trivial. A simple but incorrect approach would be to represent an initial value as a binary number, and execute Binary Consensus for every bit of the initial value in a Finite-valued Consensus problem. However, this does not satisfy the Validity requirement. Equivalence is proved by

**Theorem 3.3** For any number $k \leq n$ and any benign failure type $T$, if there is a $k$-T-resilient algorithm for Binary Strong Consensus, then there is a $k$-T-resilient algorithm for Finite-valued Strong Consensus.

**Proof.** Let $A$ be a $k$-T-resilient Binary Strong Consensus algorithm. We show how an algorithm for Finite-valued Strong Consensus can be constructed using $A$ as a building block. First, each processor $p_i$ keeps a variable $w_i$, a binary number initially set to its initial value $v_i$. The processor then sequentially executes $A$, as many times as there are bits in $w_i$. Processor $p_i$ also maintains a set $W_i$ of all $w_i$-values that it 'has heard of' during the current execution of $A$. This set is initialized to $\{w_i\}$ at the start of $A$. Now, let us say that we are in the $k^{th}$ 'round' of $A$, i.e. we are executing $A$ for the $k^{th}$ time. The $k^{th}$ bit of $w_i$ is used as input value. To every message that a processor sends, it appends $W_i$. When $p_i$ receives a message it merges the attached set $W_j$ into its own set $W_i$. However, it must check that the elements of $W_j$ match those already in $W_i$ in the first $k - 1$ bits. If this is not the case, they are discarded. At the end of the round
the value of $w_i$ may change, depending of the outcome of the Binary Consensus. If the outcome is equal to the input bit, then $w_i$ is not changed. Otherwise, the processor copies some value $w_i$ from its set $W_i$, such that the $k$th bit of $w_i$ is equal to the decision reached in $A$.\footnote{Again, it is nontrivial to prove that when $A$ has completed, set $W_i$ actually contains such a message. It is left to the interested reader to verify that any correct algorithm $A$ has this property.} If processors $p_i$ and $p_j$ are correct, then after this round

- the first $k$ bits of $w_i$ and $w_j$ are the same, and
- $w_i$ is equal to some initial value.

The first of the two properties follows from the fact that $A$ satisfies Agreement. The second can be checked by verifying that any value in $W_i$ is equal to some initial value. Finally, after the last round, $w_i$ is taken as decision for the Finite-valued Consensus problem. From the above properties, correctness of the Consensus algorithm is easily verified.  

Thus, we can solve Finite-valued Consensus if we can solve Binary Consensus, but at a price: time complexity and message complexity are higher by at least a factor $\log |V|$. Fortunately, many algorithms solve Finite-valued Consensus, which implies solvability for Binary Consensus. Conversely, impossibility proofs are often stated for Binary Consensus (such as in [FLP85]) so they are equally valid for Finite-valued Consensus. In this paper, we take Finite-valued Consensus as default: $V = \{0, 1, \ldots, v_{max}\}$, unless stated otherwise.

- The Validity requirement prevents the trivial solution in which the processors always take a certain decision (say 0), regardless of the initial values.

- As it is defined here, the Validity requirement creates a problem if we are to deal with malicious failures. In that case, processors can 'make up' an initial value, and otherwise behave like a correct processor. The outside world can never know the real initial value. If this causes other, correct, processors to decide on that value, the Validity requirement is not met. This is e.g. possible in Algorithm 3.1.

Because of this, we see in some publications a different formulation: for every $v \in V$, there is an execution in which some correct processor decides $v$. This definition has the disadvantage that we can't differentiate between Strong and Weak Consensus anymore. An alternative approach is to state that for a maliciously faulty processor, every value in $V$ is an initial value; but this is a bit dodgy. The definition chosen here is adequate for benign failures — the classes that are the main concern in this paper — and renders proofs somewhat easier.

An important variation is Weak Consensus, where Validity need only be satisfied in executions in which all processors are correct.

**Definition 3.4** Each processor $p_i$ has an initial value $v_i$, taken from a totally ordered set $V$. An algorithm solves Weak Consensus iff it satisfies

1. Agreement: Whenever two correct processors decide, their decisions are the same.
2. Weak Validity: If all processors remain correct, the decision of a processor is equal to the initial value of some processor.

3. Termination: All correct processors eventually decide.

Weak Consensus is often used in lower bounds and impossibility proofs, so as to give the theorem the widest applicability. For practical applications Strong Consensus is preferable: we don't want the behavior of a faulty processor to lead to a decision that has no relation with the initial values.

The definitions of Strong and Weak Consensus do not put any restrictions on the decision of a faulty processor. It may take no decision, but it may also decide on a value different from the decision of the correct processors. For some applications, this is undesirable. For example, if the processors manage a database (as in the example at the start of this section), data items can get incorrect values, which may have unwanted results if the values are used in further processing. Another example is a control system, where each processor controls an actuator (e.g., a valve). A decision by a faulty processor may result in an unsafe situation in the controlled environment.

What would be needed is a Consensus algorithm in which faulty processors either take the same decision as the correct ones, or take no decision at all. This we name Uniform Consensus, after a similar definition for Broadcast in [HT93].

**Definition 3.5** Each processor $p_i$ has an initial value $v_i$, taken from a totally ordered set $V$. An algorithm solves Uniform Consensus if it satisfies

1. Uniform Agreement: Whenever two processors decide, their decisions are the same.
2. Uniform Validity: The decision of any processor is equal to the initial value of some processor.\(^2\)
3. Termination: All correct processors eventually decide.

This problem is, of course, only relevant if the failures are benign: a maliciously faulty processor is free to take any decision it wants. Also, for crash failures the problem may be no harder than Strong Consensus, since in that failure model processors either execute correctly or do nothing at all.

### 3.1 Synchronous Consensus

The earliest results on Consensus [PSL80, LSP82] considered a system in which the processors operate in lock-step synchrony. The first algorithms solved Consensus for the case where the faulty processors exhibit Byzantine behavior. These algorithms used a number of messages that grew exponentially with $n$. Furthermore, the problem was proved to be solvable only if $f < n/3$.

Consensus appeared to be much easier if the failures are authentication-detectable instead of plain Byzantine. In [LSP82] an algorithm was given that solves Consensus for $f < n$, in other words, it tolerates an arbitrary number of failures. The communication complexity was improved from exponential to polynomial by a variation on this algorithm, published in [OS83]. This is shown in a simplified form, as Algorithm 3.1. This version is not tolerant of authenticated Byzantine failures, but only of omission failures.

\(^2\)Note that Uniform Validity is not strictly necessary: it follows from Uniform Agreement and Validity.
3.1 SYNCHRONOUS CONSENSUS

**Algorithm 3.1:** CONSENSUS ALGORITHM FOR PROCESSOR $p_i$

```
var $M_i \in V$ init $v_i$ (※ the minimum of all initial values known to $p_i$)

In round 1, 2, ... $f + 1$:
  if $M_i$ changed in the previous round
    then send $M_i$ to all other processors
  when receiving $M_j$ from processor $j$:
    if $M_j < M_i$ then $M_i \leftarrow M_j$

After round $f + 1$:
  decide $M_i$
```

**Theorem 3.6** Algorithm 3.1 is an $f$-omission-resilient Strong Consensus algorithm.

**Proof.** **Agreement:** By contradiction. Suppose $p_i$ and $p_j$ are both correct processors, and decide on different values. Without loss of generality, let $p_i$'s decision be the smallest of the two, say $x$. Now we look at the round in which $p_i$ received $x$ (if $x$ was its own initial value then this round number is 0). Suppose, first, that $p_i$ received $x$ in or before round $f$. By the algorithm, it would have sent this value to $p_j$ in the next round. Since both $p_i$ and $p_j$ are correct, the message would be received, so $p_i$ must also decide $x$.

Hence, $p_i$ must have received $x$ in round $f + 1$, say from processor $p_k$. This processor must be faulty, else $p_j$ would have received $x$ from $p_k$ as well. Furthermore, as $p_k$ is sending $x$ in round $f + 1$, it must have received $x$ in round $f$ from a fourth processor, say $p_l$. Again, $p_k$ must be faulty, otherwise $p_l$ would also have received $x$.

We can repeat this argument for rounds $f - 1$, $f - 2$, ... , 1. This results in a series of $f + 1$ processors that must be faulty. All have sent a value of $x$ in some round. The processor identities in this series are all different, because a processor sends a value of $x$ only once. That means that there are $f + 1$ faulty processors, which is impossible by assumption. Therefore, all correct processors take the same decision.

**Validity:** It can easily be checked (by induction on the number of rounds) that for any processor $i$, $M_i$ is always equal to the initial value of some processor.

**Termination:** By construction of the algorithm, all correct processors decide after $f + 1$ rounds.

The message complexity of this algorithm is $O(n^2)$: each processor will change and transmit a new $M_i$ at most $|V|$ times.

Algorithm 3.1 can be modified to make it tolerant of authentication-detectable Byzantine failures. To do this, the processors add their signature to a message before sending it. Thus, if a value has been passed on by several processors in a row, it should contain all their signatures. Processors may only process and retransmit a value if it is acceptable. A message is acceptable if the signatures attached to it are valid, come from different processors and if the number of signatures is equal to the number of the round in which it is received.

Note that it is assumed that a falsification of a processor’s own initial value is also detectable. Otherwise, the algorithm is incorrect: a faulty processor may transmit a number lower than its initial value, and otherwise function normally. If the lower value becomes the common decision value, the Validity requirement is not met.

For crash failures, Algorithm 3.1 also solves Uniform Consensus:
\begin{algorithm}
\begin{align*}
\text{var } & M_i \in V \text{ init } v_i; \, (*) \text{ the minimum of all initial values known to } p_i; * \\
\text{In round } & 1, 2, \ldots, f + 1: \\
& \text{if } M_i \text{ changed in the previous round} \\
& \quad \text{then send } \langle M_j \rangle \text{ to all other processors} \\
& \quad \text{when receiving } \langle M_j \rangle \text{ from processor } p_j: \\
& \quad \text{if } M_j < M_i \text{ then } M_i \leftarrow M_j \\
\text{In round } & f + 2: \\
& \text{send } M_j \text{ to all other processors.} \\
\text{After round } & f + 2: \\
& \text{if } 3^v \text{ such that it received at least } f + 1 \text{ values } v \text{ in round } f + 2 \\
& \quad \text{then decide } v
\end{align*}
\end{algorithm}

\textbf{Algorithm 3.2: Uniform Consensus Algorithm for Processor } p_i

\textbf{Theorem 3.7} Algorithm 3.1 is an $f$-crash-resilient Uniform Consensus algorithm.

\textbf{Proof.} Taking a decision is the last step in the algorithm. If a processor crashes before this step, it does not decide and so meets the requirements. If it crashes after the decision, it has completed the algorithm before failing, and is therefore a correct processor. \hfill \Box

For omission failures, however, the algorithm does not solve Uniform Consensus. A faulty processor may omit to receive any messages sent to it in the first $f + 1$ rounds, and thus decide on its own initial value. A small modification suffices to make it correct: we add an extra round in which all processors exchange their values of $M_i$, and then decide on a value that is held by at least $f + 1$ processors (which is unique, as proved below).

\textbf{Theorem 3.8} Algorithm 3.2 is an $f$-omission-resilient Uniform Consensus algorithm if $f < n/2$.

\textbf{Proof.} \textbf{Uniform Agreement:} The proof of Algorithm 3.1 shows that after round $f + 1$, all correct processors $p_i$ have the same value for $M_i$. Let this value be $w$. If a processor receives in round $f + 2$ at least $f + 1$ values equal to some $v$, then there must be at least one correct processor among the ones that sent these values. So $v = w$, or in other words: whenever a processor decides, this decision must be $w$.

\textbf{Uniform Validity:} As in Algorithm 3.1, $M_i$ of any processor (faulty or not) is always equal to the initial value of some processor.

\textbf{Termination:} There are $n - f \geq f + 1$ correct processors. As argued to prove Agreement, these all send out the same value in round $f + 2$. Hence, all correct processors decide. \hfill \Box

We see that to reach Uniform Consensus, this algorithm can tolerate less omission failures than 3.1 for Strong Consensus ($f < n/2$ versus $f < n$). The following theorem shows that this is inevitable.

\textbf{Theorem 3.9} There is no $f$-omission-resilient Uniform Consensus Algorithm if $f \geq n/2$.

\textbf{Proof.} Assume there exists an algorithm for Uniform Binary Consensus. Divide the $n$ processors in two disjoint groups $P$ and $Q$, each of size at least 1 and at most $f$. We construct three scenarios, and will arrive at a contradiction.
3.2 Asynchronous Consensus

- Scenario A: All initial values are 0, the processors in Q immediately crash, and those in P remain correct. Since Q contains at most \( f \) processors, the processors in P must be able to eventually reach a decision. According to the Uniform Validity condition this decision must be 0.

- Scenario B: Analogously to scenario A: all initial values are 1, the processors in P immediately crash, the ones in Q remain correct. The processors in Q decide 1.

- Scenario C: The initial values are 0 for processors in P, 1 for processors in Q, and all processors in Q are faulty. The omission failures are such that messages between processors in the same group arrive, but all messages between processors in different groups are omitted. To the processors in P this scenario is indistinguishable from scenario A, so they decide 0. To those in Q it is indistinguishable from B, so these must decide 1.

Thus, in scenario C the two groups take conflicting decisions, violating the Uniform Agreement condition.

3.2 Asynchronous Consensus

In the previous section we saw that Consensus is possible in a lock-step synchronous system, even in the presence of Byzantine failures, the hardest class of failures. When we view the algorithms for synchronous Consensus, we see that they use the synchrony between correct processors: the rounds are counted, giving the processors a measure of time. Thanks to the synchrony, the correct processors are all in the same round at a certain moment.

Without this synchrony, Consensus proved to be much harder. In 1985 Fischer, Lynch and Paterson [FLP85] published a ground-breaking and surprising result: Consensus, even Weak Consensus, is impossible in an asynchronous system, even if there is only a single crash failure to be tolerated. Their system model is completely asynchronous:

- Message transfer between processors can take an arbitrary (though finite) amount of time.

- Processors can 'go to sleep' for an arbitrary time. However, to distinguish faulty from correct processors, the correct ones take an infinite number of steps in any infinite execution.

- Messages need not arrive in the same order that they were sent in.

**Theorem 3.10** In an asynchronous system, there exists no deterministic 1-crash-resilient Weak Binary Consensus algorithm.

For the long and complicated proof, we refer the reader to [FLP85]. Note that this result states impossibility for deterministic algorithms. It is possible to reach Consensus with randomized algorithms, as was shown in several papers. Bracha and Toueg [BT85] published an \( f \)-crash-resistant algorithm for \( f < n/2 \), and an \( f \)-Byzantine-resistant algorithm for \( f < n/3 \). In both cases they also proved that there is no algorithm that tolerates more failures.

A different approach was taken by Chandra and Toueg in [CT91, CT92]: every processor is assumed to run a failure detector, which maintains a set of processors that are suspected to
have crashed. If the failure detector is perfect then Consensus is of course easy; the goal is to find the weakest properties that a failure detector must have in order to make Consensus possible.

3.3 Partially Synchronous Consensus

Looking at the results in the previous sections, there appears to be a large gap between the synchronous and the asynchronous. On the one hand we have the high resiliency that has been achieved for Consensus in lock-step synchronous systems, as in [DS83]. On the other hand there is the strong impossibility result in [FLP85] for asynchronous systems. Of course, the system models used in these publications have large differences. One may wonder what the determining factor(s) may be for reaching Consensus.

3.3.1 Synchrony parameters

The above question was in part answered by Dolev, Dwork and Stockmeyer, in [DDS87]. They distinguished five parameters that may influence the achievability of Consensus:

1. **Synchronous or asynchronous processors.** If processors are synchronous, there is a constant $\Phi$ such that in any interval of global time, if some processor takes $\Phi$ steps, then every other correct processor takes at least 1 step.

2. **Synchronous or asynchronous communication.** If communication is synchronous, there is a constant $\Delta \geq 1$ such that every message is delivered within $\Delta$ real-time steps after sending. With asynchronous communication, messages can be delayed for an arbitrary, but finite, amount of time.

3. **Ordered or unordered communication.** In case of ordered communication, messages are delivered in the same real-time order in which they were sent. With unordered communication, messages can be reordered.

4. **Broadcast or point-to-point transmission.** With broadcasts, a processor can send messages to all processors in one atomic step. This implies that if a processor wants to send a message to all other processors, but crashes while doing so, then it sends them either to all or to none.

5. **Atomic receive/send or separate receive and send.** With atomic receive/send, processors can execute the sequence of receiving messages, calculating the next state and sending other messages, as one atomic step.

These parameters represent five choices, resulting in 32 combinations. For each of these combinations, Dolev et al. gave the maximum resiliency, i.e. the maximum number of processor crashes for which a Consensus algorithm exists. The results are shown in Table 3.3. In this scheme, the result from [FLP85] is found back as the case with broadcast and send/receive atomicity ($\rho, \tau, m, b, s$). We have lock-step synchrony if both processors and communication

---

3 An important thing to note is that this 'real-time step' is not equal to a unit of global time. Instead, one should imagine a clock, advancing by one unit each time some processor takes a step. At each clock tick, exactly one processor takes a step.
3.3 PARTIALLY SYNCHRONOUS CONSENSUS

Table 3.3: MAXIMUM RESILIENCY FOR EACH COMBINATION OF SYNCHRONY PARAMETERS

<table>
<thead>
<tr>
<th></th>
<th>( \overline{m}, \overline{b} )</th>
<th>( m, b )</th>
<th>( m, b )</th>
<th>( m, \overline{b} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p, \overline{c} )</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>( p, c )</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>( p, c )</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>( p, \overline{c} )</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

\( p \) = synchronous processors
\( c \) = synchronous communication
\( b \) = broadcast
\( s \) = send/receive atomicity

are synchronous, with \( \Phi = 1 \) and \( \Delta = 1 \). For other values of \( \Phi \) and \( \Delta \), Algorithm 3.1 can also be used, in a modified form that is described below.

As in Algorithm 3.1, each processor \( p_l \) stores the minimum \( M_l \) of the values that it has seen. Whenever \( M_l \) changes, \( p_l \) relays the value to all other processors. However, the processors do not operate in rounds. Instead, the retransmission is done immediately. The algorithm terminates after a 'timeout', i.e. after a certain number of steps. This timeout is such that a value can be received and retransmitted \( f \) times, and will still be received by any correct processor before the timeout expires. Sending takes \( n - 1 \) steps, receiving takes 1 step, and message delay takes at most \( \Delta \) steps, so the timeout is set to \( f(\Phi n + \Delta) \).

3.3.2 Partial synchrony

Table 3.3 shows that in the case with synchronous processors and synchronous communication, the boundaries are sharp: if either of the synchrony parameters is dropped (i.e. changed to asynchronous), Consensus becomes impossible. Dwork, Lynch and Stockmeyer [DLS88] investigated variations on this model, under the name of partial synchrony. First, note that in order to simulate a lock-step synchronous system, we need to know the value of \( \Phi \) and \( \Delta \). If one or both of these values are unknown, the result from [DDS87] does not hold. Another type of partial synchrony is that a bound does not hold initially, but only after a certain (unknown) time. Dwork et al. showed that Consensus is still possible, but that the resiliency is in most cases lower.

The idea behind the algorithms in [DLS88] is to first define a basic round model, which is eventually lock-step synchronous. Then construct algorithms and prove them correct for this model. After that, it is shown that other models can simulate the basic model, so that the algorithms are equally applicable in those models.

In the basic model, processors operate in rounds, starting at round 1. In each round processors can send and receive messages, and compute the next state, as in the lock-step synchronous model. However, the processors do not necessarily go to the next round at exactly the same instant. Messages either arrive in the same round or are lost. Even messages sent between correct processors may be lost, but there is a certain time GST (Global Stabilization Time) after which those messages always arrive.

The difficulty in this model is that initially, messages between correct processors can be
lost. Whenever a processor decides, it must be sure that no other processor will take a
different decision. Algorithm 3.1 is clearly not usable here: if in the first \( f + 1 \) rounds all
messages are lost, each processor decides on its own initial value, so Agreement is unlikely.
The algorithm may be therefore called 'unsafe'.

By contrast, the algorithms [DLS88] are 'safe', in that processors will not decide if too
many messages are lost. Instead, they try again; ultimately, at time \( \text{GST} \), message loss
between correct processors no longer occurs and Consensus will be reached.

We now describe the algorithm for crash and omission failures (see also Algorithm 3.4).
The processors operate in \textit{phases}, each phase consisting of 4 rounds. A phase \( h \) \((h \geq 0)\) has a
\textit{coordinator}, processor \( p_c \), where \( c = 1 + (h \mod n) \). The other processors let the coordinator
know which values are possible decision values; the coordinator then tries to select a value
to propose. If a processor receives such a proposed value, it will \textit{lock} the value: it stores the
value in a variable, together with the phase number \( \text{LockVal} \) and \( \text{LockPhase} \), respectively.
The processor acknowledges the lock by sending a message back to the coordinator. If the
coordinator receives enough acknowledgments (more than \( f \)), it decides on the locked value.

In order to determine the possible decision values, each processor also maintains a set
\textit{Proper}, containing all the initial values that are known to this processor. It
attaches the current contents of \textit{Proper} to every message that it sends, and whenever it receives a message,
it merges the attached \textit{Proper} set with its own. To a processor, a value \( v \) is acceptable as a
decision value if \( v \) is in \textit{Proper}, and it has not locked a value other than \( v \).

\textbf{Lemma 3.11} It is impossible for two distinct values to be locked in the same phase.

\textbf{Proof.} If two distinct values \( v \) and \( w \) are locked in phase \( k \), then the coordinator has sent out
messages \( (\text{lock} \ v, k) \) and \( (\text{lock} \ w, k) \), which is impossible. \hfill \Box

\textbf{Lemma 3.12} Let \( k \) be the first phase in which some processor decides, and let this decision be \( v \).
Then at least \( f + 1 \) processors lock \( v \) at phase \( k \). Let \( p_i \) be such a processor. From that time on,
\text{LockVal}_i = v, and \text{LockPhase}_i \geq k.

\textbf{Proof.} It is easy to see that at least \( f + 1 \) processors lock \( v \) at phase \( k \). To prove the second part
of the lemma, let \( t \) be the earliest phase in which the some processor removes the lock. This
can only happen in the last round of phase \( t \), when it receives a message \( (\text{locked} \ x, h) \) from
another processor. Here, \( x \neq v \) and \( h \geq k \); Lemma 3.11 implies that \( h > k \). That means that
the coordinator of phase \( h \) sent out \( (\text{lock} \ x, h) \), so that at the first round of this phase at least
\( n - f \) processors have locked either \( x \) or no value at all. But, by the first part of the lemma,
at least \( f + 1 \) processors have locked \( v \). Thus, there are at least \( (n - f) + (f + 1) = n + 1 \)
processors, which is impossible. \hfill \Box

\textbf{Theorem 3.13} Algorithm 3.4 is an \( f \)-omission-resilient Uniform Consensus Algorithm if \( f < n/2 \).

\textbf{Proof.} \textit{Uniform Agreement:} Suppose \( p_i \) is the first processor that decides, say on value \( v \) in
phase \( k \). Lemma 3.12 states that in any later phase, at least \( f + 1 \) processors will have locked \( v \).
Therefore, there can never be \( n - f \) or more processors that (in the first round of a phase)
send a value other than \( v \) to the coordinator; the coordinator will not send \text{lock} messages for
an other value, let alone decide on it. Thus, no processor will decide a different value.

\textit{Uniform Validity:} It is easily verified that in the first round of a phase, only values from
\textit{Proper} are sent, and that the values in this set are the initial value of some processor.
### 3.3 PARTIALLY SYNCHRONOUS CONSENSUS

**Algorithm 3.4:** A PHASE IN THE ALGORITHM FOR PARTIALLY SYNCHRONOUS CONSENSUS

**Termination:** Let $k$ be the first phase after GST, and let $p_c$ be the coordinator of phase $k + 1$. Assume that $p_c$ is correct and has not yet decided. We will show that $p_c$ decides in this phase. First, consider the set of values locked by correct processors at the end of phase $k$. Since in the last round of phase $k$ all messages between correct processors arrive, we know that this set contains at most one value. Also, by the way the *Proper* sets are distributed, we know that there are values that are in the *Proper* set of every correct processor, and that locked values are always proper. Thus, at the end of the first round of phase $k + 1$, $p_c$ can find a value $v$ to propose. There are at least $t + 1$ correct processors, who will receive $(\text{lock } v, k + 1)$ and reply with $(\text{ack } k)$. Thus, $p_c$ decides $v$. Since the processors become coordinator in round-robin fashion, at the end of phase $k + n$ all correct processors will have decided.

It is hard to determine the performance of this algorithm. The number of messages sent per phase is at most $n^2 + 3n$, but the number of phases needed depends on GST, which is not known. Note also that processors keep on executing the algorithm after deciding, so in a sense the algorithm does not terminate, and the maximum number of messages is unbounded.\(^4\) But every correct processor eventually decides, at the latest in the $n^{th}$ phase.

\(^4\)This can be solved by letting a processor broadcast its decision. Other processors, when receiving this
Dwork et al. showed that the resiliency of this algorithm is optimal, by the following impossibility result:

**Theorem 3.14** There is no \( f \)-crash-resilient Weak Consensus Algorithm if \( f \geq n/2 \).

**Proof.** Assume there exists an algorithm for Binary Consensus. Divide the \( n \) processors in two disjoint groups \( P \) and \( Q \), each of size at least 1 and at most \( f \). We construct three scenarios, and will arrive at a contradiction.

- **Scenario A:** All initial values are 0, the processors in \( Q \) immediately crash, and messages between processors in \( P \) take 1 unit of global time to be delivered. Since \( Q \) contains at most \( f \) processors, the processors in \( P \) must be able to eventually reach a decision, say within \( T_A \) time units. This decision must be 0. For if it were 1, we could modify it to create scenario \( A' \), in which the processors in \( Q \) remain correct, but messages between processors in \( P \) and \( Q \) take more than \( T_A \) time. To processors in \( P \), scenario \( A' \) is indistinguishable from \( A \) in the first \( T_A \) time units. Therefore, in \( A' \) the processors in \( P \) would also decide within time \( T_A \). They would decide 1, violating the Weak Validity condition.

- **Scenario B:** All initial values are 1, the processors in \( P \) immediately crash, and messages between processors in \( Q \) take 1 unit of global time to be delivered. By a similar argument, the processors in \( Q \) decide 1, within \( T_B \) time.

- **Scenario C:** The initial values are 0 for processors in \( P \), 1 for processors in \( Q \), and all processors remain correct. Messages between processors in the same group take time 1, but messages between processors in different groups take more than \( \max(T_A, T_B) \) time. To processors in \( P \) this scenario is indistinguishable from scenario \( A \), at least for the first \( T_A \) time units. Therefore, they decide 0 within time \( T_A \). Similarly, for the processors in \( Q \) this scenario is indistinguishable from \( B \), so these must decide 1 within time \( T_B \).

Thus, in scenario \( C \) the two groups take conflicting decisions, violating the Agreement condition.

The paper [DLS88] also contains algorithms and impossibility proofs for all failure classes including malicious failures, but since benign failures are the main concern in this paper, we limit this description to those classes.

### 3.4 Lower Bounds

Fischer and Lynch [FL82] proved that reaching Consensus in the presence of Byzantine failures takes at least \( f + 1 \) rounds. This result was later extended to crash failures by Moses and Tuttle [MT88]. Note that this holds only for deterministic algorithms, and that it is a bound on the worst-case execution time. The worst case is usually an execution in which the maximum number of failures occur, which is of course highly unlikely. It is possible to improve on the average number of rounds by using so-called early stopping algorithms, as message, also decide and rebroadcast the decision. After deciding, a processor can terminate. Through this modification the execution time is reduced as well: correct processors terminate within \( f + 1 \) phases after GST.
treated e.g. in [DRS90]. In such algorithms, the execution time is proportional to the number of failures that actually occur.

The lower bounds on the number of rounds concern lock-step synchronous systems, but they have a wider application. For instance, suppose we have a model with a bound of $\Delta$ time units on communication delay. Then no algorithm will have an execution time that is better than $(f + 1)\Delta$ time units.

### 3.5 Distributed Consensus versus Reliable Broadcast

In this section we take a look at how Consensus and Reliable Broadcast are related. To begin, a reduction from Consensus to Broadcast is straightforward, as expressed by the following theorem:

**Theorem 3.15** For any number $k \leq n$ and failure type $T$, if there is a $k$-$T$-resilient algorithm for Reliable Broadcast, then there is a $k$-$T$-resilient algorithm for Strong Consensus.

**Proof.** Let $A$ be an algorithm for Reliable Broadcast. We can construct an algorithm for Consensus as follows: each processor broadcasts its initial value to the others, using algorithm $A$. The received values (i.e. the outcome of the broadcasts) are stored in an array, with $n$ elements that are initialized to a null value. The broadcast received from processor $p_i$ is stored as the $i^{th}$ element of the array. When all broadcasts have been completed, each processor decides on the first non-$\perp$ value in the array. Such a non-null element can be found, provided that there is at least one correct processor. This proves Validity and Termination. Agreement is also satisfied: given the correctness of $A$, all correct processors will end up with the same array of values, so their decisions will be the same.

A general reduction from Broadcast to Consensus, however, is more complicated to do. The idea is to let the sender transmit its message to all processors, and let them use the received value as input to a Consensus algorithm. When a processor receives nothing from the sender, it chooses a default value to indicate the absence. For example, Algorithm 3.1 would be suitable algorithm, provided we order the default value to be higher than any other value. The difficulty lies in a tacit assumption for the Consensus problem: all processors start the algorithm at the same instant. In a lock-step synchronous system this is no problem, as the message from the sender must arrive in the first round. But with a less strict synchrony it may be impossible to start the Consensus algorithm simultaneously.

Thus, in lock-step synchronous systems we have an equivalence of Consensus and Reliable Broadcast, in other systems this may not be the case. For asynchronous systems they are not equivalent, as Hadzilacos and Toueg [HT93] have pointed out: Consensus can be solved with a randomized algorithm, while Broadcast can not be solved, even with randomization.

---

2In fact, the original version of Algorithm 3.1 in [DS83] was presented in the form of a Broadcast algorithm.
Chapter 4

Distributed Consensus in Hard Real-Time systems

Let us now consider how well the results in the previous chapter can be applied in hard real-time systems. Obviously, since the programs that use the algorithms must meet their deadlines, the algorithms themselves must terminate within an a priori bounded amount of time.

**Definition 4.1** A run of an algorithm is timely if all correct processors complete the algorithm within a bounded amount of global time. An algorithm is timely iff all legal runs are timely.

In an algorithm there are two time-consuming activities: processor steps and message transmission. As a consequence there must be time bounds on the execution of both. Without these bounds there can be no guarantee that deadlines will be met. We therefore explicitly state them as assumptions:

**Assumption 4.2** In a hard real-time system, there exist a priori known upper bounds (in global time) on the execution time of a processor step, and on the transmission delay of a message.

With these properties it is sufficient for the number of processor steps to be bounded, in order to make every run timely.

The existence of both upper and lower bounds on processor speed means that we have synchronous processors according to the definitions in [DDS87]. However, we do not, by those definitions, have synchronous communication as well: Dolev et al. define this as a bound on communication delay as measured in processor steps, whereas the above bounds are stated for global time.

It does not seem logical to assume that the transmission speed depends on the speed at which processors take steps, as Dolev et al. do. In practice, communication facilities are often provided by dedicated hardware; a processor that wants to send a message hands the data to the communication subsystem, and some time later the destination processor is notified of the arrival of a message. The communication subsystem usually has its own timing hardware to regulate the speed of transmission.

A further limitation of [DDS87] is that it only considers crash failures. As we are also interested in omission and timing failures, we would need to extend the results to other failure classes.
The general approach we take here is that the execution is governed by the local clock; the algorithm specifies at what local time a certain step should be taken. We distinguish six timing models, plus a 'clock-less' model for comparison.

1. **Perfect clocks.**
   Local clocks exactly follow global time: \( \tau_s(k) = \tau_l(k) = k \).

2. **Bounded precision.**
   Local clocks have a bounded difference \( \epsilon \) from global time: \( \tau_s(k) = k - \epsilon, \tau_l = k + \epsilon \).

3. **Bounded-drift precision.**
   Local clocks remain within a linear envelope of size \( \rho < 1 \) from global time: \( \tau_s(k) = k(1 - \rho), \tau_l = k(1 + \rho) \).

4. **Perfect accuracy.**
   Local clocks are exactly equal: \( \tau_s(k) = \tau_l(k) \).

5. **Bounded accuracy.**
   Local clocks have a bounded difference \( \epsilon \) from each other: \( \tau_s(k) = \tau_l(k) - \epsilon \).

6. **Bounded-drift accuracy.**
   Local clocks have a bounded drift \( \rho \) from each other: \( \tau_s(k) = \tau_l(k) - k \cdot \rho \).

7. **No accuracy.**
   Processor timing is only restricted by the minimum and maximum execution time (\( \phi \) and \( \Phi \), respectively) for a processor step: \( \tau_s(k) = k\phi \) and \( \tau_l(k) = k\Phi \).

The first three models are precise: they state a relation between global time and local time. The next three only state the accuracy: the local clocks may be closely, even perfectly, synchronized, but their behavior in global time is not known.

We are considering timing failures, so the behavior of incorrect processors need not follow these timing models. The times at which an incorrect processor takes steps are only restricted by a minimum on the step time.

For each of these models, we show that a lock-step synchronous model can be simulated.

### 4.1 Simulations of lock-step synchrony

What is common to all simulations is that each processor keeps a round counter, starting at 1. The round counter is incremented after a certain number of steps, a number which depends on the timing model in question. Each message that is sent must be tagged with the current round number of the sender. When a message is received, the processor compares this round number with its own current round counter. If the message carries a lower round number, it is discarded. Messages with a higher round number are stored, to be processed when the receiver has entered that round. In this way, we have guaranteed that a message either is processed by the receiver in the same (local) round that the sender was in when it sent the message, or the message is lost. What remains to be checked is that messages between correct processors are never lost.

The resulting simulation algorithm is shown as Algorithm 4.1. The function \( \text{Round} \) returns for any \( k \geq 1 \) the local clock time at which round \( k \) should start. Of course \( \text{Round} \) depends
4.1 SIMULATIONS OF LOCK-STEP SYNCHRONY

```plaintext
var R ∈ N init 1 (* round counter *)
r ∈ N
msg : message
Buf : set of message init ∅

repeat
(* simulate round R *)
At time Round(R):
    send messages according to simulated algorithm, tagged with R
repeat
    receive message (msg, r)
    if r ≥ R then Buf ← Buf ∪ {msg, r}
until time Round(R + 1)
process { (msg, r) ∈ Buf | r = R } according to simulated algorithm
    R ← R + 1
until simulated algorithm terminates
```

Algorithm 4.1: SIMULATION OF LOCK-STEP SYNCHRONY

on the timing model; it will be discussed hereafter. In these simulations, we assume that there is no broadcast transmission, and that send and receive are separate. If these types of synchrony are present, the simulations can be made somewhat more efficient. The time bounds mentioned earlier are denoted by \( \phi \) for minimum step time, \( \Phi \) for maximum step time and \( \delta \) for message delay.

Perfect clocks

For this model, there is an easy simulation of a lock-step synchronous system. Each round takes \( n - 1 + \lceil \delta \rceil \) clock ticks, with round 1 starting at time 0. The first \( n - 1 \) steps are for sending messages, according to the round-based algorithm that is being run. The next \( \delta \) time units are spent waiting for the messages to arrive. This requires \( \lceil \delta \rceil \) steps. After the last of these steps the transition to the next round is calculated. It is easy to verify that messages between correct processors are sent and received in the same round.

Bounded precision

A slight modification from the previous case suffices to produce a simulation of lock-step synchrony: the waiting period of \( \delta \) is extended by \( 2\epsilon \), giving a total of \( n - 1 + \lceil \delta + 2\epsilon \rceil \) steps.

To verify that the waiting period is indeed long enough, assume that a correct processor \( p_i \) sends a message at local time \( k \). In global time, this takes place at or before time \( k + \epsilon \). The message will arrive at most \( \delta \) later, at global time \( k + \epsilon + \delta \). At that instant, no correct local clock will be past \( k + \epsilon + \delta + \epsilon \). Thus, a delay of \( \lceil \delta + 2\epsilon \rceil \) local clock ticks suffices.

\[1\text{The amount of processing required in a single step depends on the algorithm, and so does } \phi. \text{ Since processors are expected to take steps at the same speed as global time, we assume } \Phi < 1.\]
Bounded-drift precision

This case is considerably more complicated than the previous two. We inductively define $T_r$, the local time at which a processor starts round $r$.

The first $n - 1$ steps are used for sending messages; any correct processor has finished this at local time $T_r + n - 1$, which is at or before global time $(T_r + n - 1)(\rho + 1)$. At the latest, the messages arrive at global time $(T_r + n - 1)(\rho + 1) + \delta$; since the fastest correct local time is at most $1/(1 - \rho)$ times the global time, we have

$$T_{r+1} = \left\lceil \frac{(T_r + n - 1)(\rho + 1) + \delta}{1 - \rho} \right\rceil$$

To obtain a general formula for $T_r$, we approximate the above equation by

$$T_{r+1} \leq (T_r + n + \delta)\frac{1 + \rho}{1 - \rho}$$

The left side is of the form $cT_r + d$ where $c = \frac{1 + c}{1 - \rho}$ and $d = c(n + \delta)$. The series starts with $T_1 = 0$, so in general we have

$$T_r \leq c^{r-2}d + c^{r-3}d + \ldots + cd + d = d \cdot \frac{1 - c^{r-1}}{1 - c}$$

Perfect accuracy

The difficulty for this case lies in the fact that the clocks have only a weak relation with global time, through the $\phi$ and $\Phi$ bounds. It is possible to simulate a lock-step synchronous system, but we must change the period spent waiting for incoming messages. To ascertain that this period lasts at least $\delta$ time, it must take at least $\delta/\Phi$ steps. Thus, a single round takes $(n - 1 + \lceil \delta/\Phi \rceil)$ steps.

Bounded accuracy

Compared with the case for perfectly accurate clocks, the waiting period must be extended to $\delta + \epsilon$, so the number of steps per round is $n - 1 + \lceil (\delta + \epsilon)/\Phi \rceil$.

Bounded-drift accuracy

We derive $T_r$ in a way similar to the one for Bounded-drift precision. The last message in a round is sent at local time $T_r + n - 1$. When the slowest correct processor has reached this point, the fastest one is at local time $(T_r + n - 1)(1 + \rho)$. The processor must then wait for $\delta$ global time units, hence

$$T_{r+1} = \left\lceil (T_r + n - 1)(1 + \rho) + \delta/\Phi \right\rceil$$

An approximation is

$$T_r = h \frac{1 - g^{r-1}}{1 - g}$$

where $g = 1 + \rho$ and $h = ng + \delta/\Phi$. 
4.2 Optimizing execution time

Using the simulations of a lock-step synchronous system, Consensus can be solved, and with maximal resiliency. The number of messages is $O(n^2)$, which also leaves little room for improvement. The worst-case execution time, however, is not optimal. For each timing model, an upper bound for the execution times for $f + 1$ rounds is given in Table 4.2.

The theoretical bound of $f + 1$ rounds for lock-step synchrony translates into a bound of $(f + 1)\delta$ units of global time. When we compare this with the figures in the table, we find that the simulation is not time-optimal for the last four cases: there is a factor $1/\phi$ in the execution time, which makes the simulation very inefficient if $\phi$ is small. Furthermore, in the cases where drift is involved and in the last 'clock-less' model, the execution time is polynomial in the drift, with degree approximately $f$.

To start with the first problem: we can make the execution time independent of $\phi$ by simulating an eventually synchronous system, and applying Algorithm 3.4. The idea behind the simulation is that the number of steps in a round is increased with every phase. As in the previous simulations, incoming messages that were sent in an earlier round are discarded.
Eventually, a round will last long enough for correct processors to successfully exchange messages.

For the cases of Perfect accuracy and Bounded accuracy, we get an execution time that is \(O(f \delta)\), as was shown in [DS91]. In the remaining case, Bounded-drift accuracy, we cannot use this technique. The time difference between the slowest correct processor and the fastest one increases with every step. We must therefore ensure that the time per phase increases faster than this difference. This can be done, but it re-introduces a factor \(1/\phi\) into the execution time.

An added advantage of this simulation is that the value of \(\delta\) is not used. In an execution where the message delays are smaller than \(\delta\), the time taken by processors to decide will be correspondingly shorter. The downside is of course that the resiliency is reduced by half: the algorithm only works for \(2f < n\).

The second problem, of execution time being polynomial in \(\rho\), can be treated by modifying Algorithm 3.1, as was done for [DDS87] on page 19. The resulting algorithm is shown as Algorithm 4.3. Again, the timeout \(T\) is chosen such that no correct processor times out before a certain instant in global time. This global time is equal to \((f + 1)(n\Phi + \delta)\), the time it takes to receive and retransmit a value \(f + 1\) times, where the delays are maximal.

**Theorem 4.4** If all correct processors time out after global time \((f + 1)(n\Phi + \delta)\), then Algorithm 4.3 is an \(f\)-omission-resilient Strong Consensus algorithm.

**Proof.** Validity and Termination are easy to verify. To prove Agreement, observe that when the timeout expires, all messages in transit have been relayed \(f + 2\) times or more. By the same argument as in the proof of Algorithm 3.1, we can show that the values carried by those messages can not be lower than the value of \(M_i\) of any correct processor. Thus, it makes no difference whether such a message will be received by a correct processor before the timeout or not. All correct processors will take the same decision. \(\square\)

For the No accuracy model, the timeout \(T\) must be at least \((f + 1)(n\Phi + \delta)/\phi\) clock ticks. The
worst-case execution time is therefore linear in the timing uncertainty \( \Phi/\phi \):

\[
(f + 1)(n\Phi + \delta) \frac{\Phi}{\phi}
\]
Bibliography


**Computing Science Reports**

**Department of Mathematics and Computing Science**
**Eindhoven University of Technology**

**In this series appeared:**

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...,then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of $P$-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
<tr>
<td>91/16</td>
<td>A.J.J.M. Marcelis</td>
<td>An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.</td>
</tr>
</tbody>
</table>
| 91/17 | A.T.M. Aerts  
         P.M.E. de Bra  
| 91/18 | Rik van Geldrop | Transformational Query Solving, p. 35. |
| 91/19 | Erik Poll | Some categorical properties for a model for second order lambda calculus with subtyping, p. 21. |
| 91/20 | A.E. Eiben  
| 91/21 | J. Coenen  
         W.-P. de Roever  
         J. Zwiers | Assertional Data Reification Proofs: Survey and Perspective, p. 18. |
| 91/23 | K.M. van Hee  
         L.J. Somers  
         M. Voorhoeve | Z and high level Petri nets, p. 16. |
| 91/24 | A.T.M. Aerts  
         D. de Reus | Formal semantics for BRM with examples, p. 25. |
| 91/25 | P. Zhou  
         J. Hooman  
         R. Kuiper | A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52. |
| 91/26 | P. de Bra  
         G.J. Houben  
| 91/27 | F. de Boer  
         C. Palamidessi | Embedding as a tool for language comparison: On the CSP hierarchy, p. 17. |
| 91/28 | F. de Boer | A compositional proof system for dynamic process creation, p. 24. |
| 91/29 | H. Ten Eikelder  
         R. van Geldrop | Correctness of Acceptor Schemes for Regular Languages, p. 31. |
| 91/30 | J.C.M. Bacten  
         F.W. Vaandrager | An Algebra for Process Creation, p. 29. |
<p>| 91/31 | H. ten Eikelder | Some algorithms to decide the equivalence of recursive types, p. 26. |
| 91/33 | W. v.d. Aalst | The modelling and analysis of queueing systems with QNM-ExSpect, p. 23. |
| 91/34 | J. Coenen | Specifying fault tolerant programs in deontic logic, p. 15. |</p>
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p.45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p.22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p.36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases; an exposition, p.32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt, F. Kamareddine</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? P. 32.</td>
</tr>
<tr>
<td>92/21</td>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
</tr>
<tr>
<td>92/22</td>
<td>R. Nederpelt F.Kamareddine</td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td>92/24</td>
<td>M.Codish D.Dams Eyal Yardeni</td>
<td>Bottum-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td>92/25</td>
<td>E.Poll</td>
<td>A Programming Logic for Foa, p. 15.</td>
</tr>
<tr>
<td>93/01</td>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02</td>
<td>T. Verhoeff</td>
<td>A continuous version of the Prisoner’s Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03</td>
<td>T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04</td>
<td>E.H.L. Aarts J.H.M. Korst P.J. Zwietering</td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
<tr>
<td>93/05</td>
<td>J.C.M. Baeten C. Verhoeef</td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/06</td>
<td>J.P. Veltkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
</tr>
<tr>
<td>93/07</td>
<td>P.D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
</tr>
<tr>
<td>93/08</td>
<td>J. Verhoosel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/10</td>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
</tr>
<tr>
<td>93/13</td>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
</tbody>
</table>


A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27

Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.

A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.

The Design of an Online Help Facility for ExSpect, p.21.


A Typechecker for Bijective Pure Type Systems, p. 28.

Relational Algebra and Equational Proofs, p. 23.

Pure Type Systems with Definitions, p. 38.


Multi-dimensional Petri nets, p. 25.

Finding all minimal separators of a graph, p. 11.

A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.

GOLD, a Graph Oriented Language for Databases, p. 42.

On Vertex Ranking for Permutation and Other Graphs, p. 11.

Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.

<table>
<thead>
<tr>
<th>#</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/33</td>
<td>L. Loyens and J. Moonen</td>
<td>ILIAS, a sequential language for parallel matrix computations, p. 20.</td>
</tr>
<tr>
<td>93/34</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Real Time Process Algebra with Infinitesimals, p.39.</td>
</tr>
<tr>
<td>93/36</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Non Interleaving Process Algebra, p. 17.</td>
</tr>
<tr>
<td>93/38</td>
<td>C. Verhoef</td>
<td>A general conservative extension theorem in process algebra, p. 17.</td>
</tr>
<tr>
<td>93/41</td>
<td>A. Bijlsma</td>
<td>Temporal operators viewed as predicate transformers, p. 11.</td>
</tr>
<tr>
<td>93/42</td>
<td>P.M.P. Rambags</td>
<td>Automatic Verification of Regular Protocols in P/T Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>Code</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America, M. van der Kammen, R.P. Nederpelt, O.S. van Roosmalen, H.C.M. de Swart</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddine, R.P. Nederpelt</td>
<td>Canonical typing and Π-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Basten, T. Kunz, J. Black, M. Coffin, D. Taylor</td>
<td>Time and the Order of Abstract Events in Distributed Computations, p. 29.</td>
</tr>
<tr>
<td>94/08</td>
<td>O.S. van Roosmalen</td>
<td>A Hierarchical Diagrammatic Representation of Class Structure, p. 22.</td>
</tr>
<tr>
<td>94/09</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
<tr>
<td>94/10</td>
<td>T. verhoeff</td>
<td>The testing Paradigm Applied to Network Structure, p. 31.</td>
</tr>
<tr>
<td>94/13</td>
<td>R. Seljée</td>
<td>A New Method for Integrity Constraint checking in Deductive Databases, p. 34.</td>
</tr>
<tr>
<td>94/14</td>
<td>W. Peremans</td>
<td>Ups and Downs of Type Theory, p. 9.</td>
</tr>
<tr>
<td>94/16</td>
<td>R.C. Backhouse, H. Doombos</td>
<td>Mathematical Induction Made Calculational, p. 36.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>94/18</td>
<td>F. Kamareddine, R. Nederpelt</td>
<td>Refining Reduction in the Lambda Calculus, p. 15.</td>
</tr>
<tr>
<td>94/19</td>
<td>B.W. Watson</td>
<td>The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.</td>
</tr>
<tr>
<td>94/20</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>Beyond ( \beta )-Reduction in Church's ( \lambda \rightarrow ), p. 22.</td>
</tr>
<tr>
<td>94/22</td>
<td>B.W. Watson</td>
<td>The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.</td>
</tr>
<tr>
<td>94/23</td>
<td>S. Mauw and M.A. Reniers</td>
<td>An algebraic semantics of Message Sequence Charts, p. 43.</td>
</tr>
<tr>
<td>94/24</td>
<td>D. Dams, O. Grumberg, R. Gerth</td>
<td>Abstract Interpretation of Reactive Systems; Abstractions Preserving ( \forall \text{CTL}^* ), ( \exists \text{CTL}^* ) and ( \text{CTL}^* ), p. 28.</td>
</tr>
<tr>
<td>94/25</td>
<td>T. Kloks</td>
<td>( K_4 )-free and ( W_5 )-free graphs, p. 10.</td>
</tr>
<tr>
<td>94/29</td>
<td>J. Hooman</td>
<td>Correctness of Real Time Systems by Construction, p. 22.</td>
</tr>
<tr>
<td>94/30</td>
<td>J.C.M. Baeten, J.A. Bergstra, Gh. Ştefanescu</td>
<td>Process Algebra with Feedback, p. 22.</td>
</tr>
<tr>
<td>94/31</td>
<td>B.W. Watson, R.E. Watson</td>
<td>A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.</td>
</tr>
<tr>
<td>94/33</td>
<td>T. Laan</td>
<td>A formalization of the Ramified Type Theory, p.40.</td>
</tr>
<tr>
<td>94/34</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>The Barendregt Cube with Definitions and Generalised Reduction, p. 37.</td>
</tr>
<tr>
<td>94/35</td>
<td>J.C.M. Baeten, S. Mauw</td>
<td>Delayed choice: an operator for joining Message Sequence Charts, p. 15.</td>
</tr>
<tr>
<td>94/36</td>
<td>F. Kamareddine, R. Nederpelt</td>
<td>Canonical typing and ( \Pi )-conversion in the Barendregt Cube, p. 19.</td>
</tr>
<tr>
<td>Number</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>94/38</td>
<td>A. Bijlsma, C.S. Scholten</td>
<td>Point-free substitution, p. 10.</td>
</tr>
<tr>
<td>94/39</td>
<td>A. Blokhuis, T. Kloks</td>
<td>On the equivalence covering number of splitgraphs, p. 4.</td>
</tr>
</tbody>
</table>