Communication Predicates: A High-Level Abstraction for Coping with Transient and Dynamic Faults *

Martin Hutle
martin.hutle@epfl.ch

André Schiper
andre.schiper@epfl.ch

École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland

Abstract

Consensus is one of the key problems in fault tolerant distributed computing. A very popular model for solving consensus is the failure detector model defined by Chandra and Toueg. However, the failure detector model has limitations. The paper points out these limitations, and suggests instead a model based on communication predicates, called HO model. *The advantage of the HO model over failure detectors is shown, and the implementation of the HO model is discussed in the context of a system that alternates between good periods and bad periods. Two definitions of a good period are considered. For both definitions, the HO model allows us to compute the duration of a good period for solving consensus. Specifically, the model allows us to quantify the difference between the required length of an* initial *good period and the length of a* non initial *good period.*

1. Introduction

Consensus is one of the key problems in fault tolerant distributed computing. Consensus is related to replication and appears when implementing atomic broadcast, group membership, etc. The problem is defined over a set of processes Π, where each process $p_i \in \Pi$ has an initial value v_i: All processes must agree on a common value that is the initial value of one of the processes.

Consensus can be impossible to solve, as established by the FLP impossibility result [13]. Later it has been shown that consensus can be solved in a partially synchronous system with a majority of correct processes [12]. Roughly speaking, a partially synchronous system is a system that is initially asynchronous, but eventually becomes synchronous.[1] Moreover, in a partially synchronous system links are initially lossy, but eventually become reliable.

The notion of failure detectors has been suggested a few years later [5]. The failure detector model is defined as an asynchronous system "augmented" with failure detectors, which are defined by some completeness and some accuracy property (see [5] for details). Over the years failure detectors have become very popular. The model is today widely accepted and has become the model mostly used for expressing consensus algorithms. However, the failure detector model has limitations.

First, failure detectors are not an abstraction of the partially synchronous model (even though this claim has sometimes been made). The reason is that in the partially synchronous model links are initially lossy, while the use of failure detector to solve a problem requires perpetual reliable links.[2] When using failure detectors, either the system must provide reliable links, or reliable links need to be implemented on top of the unreliable system links. As a consequence, the capability of algorithms of tolerating message loss — as it is the case for the Paxos algorithm [19] — cannot be expressed naturally in the failure detector model. Only a variant of Paxos that assumes reliable links can be expressed using failure detectors, as done, e.g., in [4].

Second, failure detectors are not well suited to solve consensus in the crash-recovery model, with or without stable storage [1]. In the crash-recovery model, a process can crash and later recover. This is in contrast to the crash-stop model, in which process crashes are permanent. Intuitively, one would think that solving consensus in the crash-stop model or in a crash-recovery model should not lead to major algorithmic differences. However, the comparison of (i) the $\diamond\mathcal{S}$ consensus algorithm in the crash-stop model [5] with (ii) the corresponding algorithm in the crash-recovery model with stable storage [1] shows that the crash-recovery algorithm is a much more complicated protocol than the corresponding crash-stop algorithm. Moreover, the complexity of the crash-recovery consensus algorithm makes it hard to see that the crash-recovery algorithm is based on the

*Research funded by the Swiss National Science Foundation under grant number 200021-111701.

[1] This is not the only definition of a partially synchronous system.

[2] Failure detectors lead to the following programming pattern: Process p (i) waits for a message from process q or (ii) suspects q. If q is not suspected while the message is lost, p is blocked.

37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)
0-7695-2855-4/07 $20.00 © 2007 IEEE

same basic ideas as the crash-stop algorithm. This leads to the following question: Is there an inherent gap between the crash-stop and the crash-recovery model that would explain the higher complexity of the crash-recovery consensus algorithm?

Third, failure detectors cannot handle Byzantine failures. The reason is that the definition of a Byzantine behavior is related to an algorithm: It is impossible to achieve a complete separation of failure detectors from the algorithm using them. To overcome this problem, the notion of muteness detectors has been suggested [9, 10, 18]. However, it is not clear what system model could allow the implementation of muteness detectors, which is an inherent limitation of the approach.

These arguments suggest that failure detectors might not be the ultimate answer to the consensus problem. As an alternative to failure detectors, one could program directly at the level of the partially synchronous system model. However, this model provides too low level abstractions. It is indeed useful to provide higher level abstractions for expressing consensus algorithms. The goal of this paper is to show that another abstraction, namely *communication predicates*, provides a better abstraction than failure detectors for solving consensus. Specifically, the paper brings an answer to the question raised in [17], about quantifying the time it takes the environment to reach round synchronization after the system has stabilized.

The paper is structured as follows. Section 2 serves as a motivation to the introduction of communication predicates. Communication predicates are defined in Section 3. The implementation of communication predicates is presented in Section 4. Related work is discussed in Section 5, and Section 6 concludes the paper. Note that the paper is restricted to benign faults; Byzantine faults will be addressed in another paper.

2. Fault taxonomy

In this section we discuss the taxonomy of faults, with the goal to understand the limitation of failure detectors. The discussion will serve as the basis for the introduction of the notion of *communication predicates*.

2.1. Failure detectors and the paradox of the classical fault taxonomy

Let us come back to the second limitation of failure detectors (see Section 1), namely the gap between solving consensus with failure detectors in the crash-stop model and in the crash-recovery model. Our goal is to explain this gap, and so to understand the limited context in which failure detectors provide a good abstraction.

When looking at process failures, the classical fault taxonomy distinguishes, from the most benign to the most severe, (i) crash faults, (ii) send-omission faults, (iii) general-omission faults (which includes receive-omission faults), and (iv) malicious faults [22]. It can be observed that this taxonomy does not distinguish crash faults without recovery (the crash-stop model) and crash faults with recovery (the crash-recovery model). So, one would expect little difference when solving consensus in either of these two models. However, as already mentioned, this is not the case with failure detectors:

- In the crash-stop model, a standard solution to consensus is the rotating coordinator algorithm that requires the failure detector $\diamond\mathcal{S}$ and a majority of processes [5].
- Extending this solution to the crash-recovery model is not easy. It requires the definition of new failure detectors, and the algorithm becomes more complex [1]. This can be observed by comparing the two algorithms that are given in the appendix of [16].

This observation leads to the following question: What is the key issue, not captured by the classical fault taxonomy, that explains the gap between the crash-stop and crash-recovery consensus algorithm? The key issue is in the distinction between *permanent* faults and *transient* faults. Crash-stop is a model with permanent (crash) faults, while crash-recovery is a model with transient (crash) faults. A fault taxonomy that does not distinguish between permanent and transient fault is not able to explain the limitation of the failure detector model. In the next section we suggest another new fault taxonomy that makes the distinction between permanent and transient fault explicit.

2.2. Alternative fault taxonomy (for benign faults)

An alternative process fault taxonomy can be organized along two dimensions. The first dimension distinguishes between the already discussed *permanent* (P) and *transient* faults (T). The second dimension distinguishes faults that can hit any process in the system from faults that hit only a subset of the processes. We use the term *static* (S) for faults that can hit only a fixed subset of processes and *dynamic*[3] (D) for all other cases, i.e., faults that can hit all processes.

Combining this two dimensions leads to four classes of process faults:

- *SP*: at most f processes out of n are faulty ($f<n$); a faulty process is permanently faulty.
- *ST*: at most f processes out of n are faulty ($f<n$); faults are transient.

[3] This notion of static/dynamic faults was also used by [21].

- *DP*: all processes can be faulty; faults are permanent.
- *DT*: all processes can be faulty; faults are transient.

Among this classes, SP is clearly the most restrictive, whereas DT is the most general one. The crash-stop fault in the classical taxonomy corresponds the SP class. The send-omission and general-omission faults are transient faults. If we assume that only a subset of processes suffer from send-omission or general-omission faults, then send-omission and general-omission faults are classified as ST. Otherwise, these faults are classified as DT.

This alternative taxonomy is able to capture the distinction between the crash-stop model and the crash-recovery model: The crash-stop model corresponds to the class SP, whereas the crash-recovery model can be classified either as ST (if some processes never crash) or as DT. Failure detectors are well-suited to handle the SP fault class, but not to handle dynamic faults. Communication predicates will allow us to handle SP and DT faults in the same way.

2.3. Transmission faults

It is usual to distinguish between process faults and link faults. However, the distinction becomes irrelevant with DT faults. To see this, consider process p sending message m to process q. Process q might not receive m if (i) p suffers from a send-omission fault, (ii) the link loses m, or (iii) q suffers from a receive-omission fault. In case (i) p is the faulty component, in case (ii) the link l_{pq} is the faulty component, in case (iii) q is faulty. However, if the fault is transient, it may not occur later, for another message m' sent by p to q. For this reason, it makes no sense to put the responsibility of the fault on one of the components (process p, process q, or link l_{pq}). This observation leads to consider only *transmission faults*:[4] a transmission fault is a fault that results in the non reception of some message m.

As we will see in Section 3, communication predicates are based on the notion of transmission faults. As such, communication predicates — contrary to failure detectors — are able to handle SP and DT fault classes uniformly.

3. Communication predicates and algorithms

3.1. Communication predicates

Communication predicates are defined in the context of a communication-closed round model. An algorithm for this model comprises, for each round r and process $p \in \Pi$, a sending function S_p^r and a transition function T_p^r. At beginning of a round r, every process sends a message to all according to $S_p^r(s_p)$, where s_p is p's state at the beginning of the round. At the end of a round r, p makes a state transition according to $T_p^r(\vec{\mu}, s_p)$, where $\vec{\mu}$ is the partial vector of all messages that have been received by p in round r.

We denote by $HO(p,r)$ the support of $\vec{\mu}$, i.e., the set of processes (including itself) from which p receives a message at round r: $HO(p,r)$ is the *heard of* set of p in round r. If $q \notin HO(p,r)$, then the message sent by q to p in round r was subject to a transmission failure. Communication predicates are expressed over the sets $(HO(p,r))_{p\in\Pi, r>0}$. For example,

$$\exists r_0 > 0,\ \forall p, q \in \Pi\ :\ HO(p,r_0) = HO(q,r_0)$$

ensures the existence of some round r_0 in which all processes hear of the same set of processes. Another example is a communication predicate that ensures that in every round r all processes hear of a majority of processes (n is the number of processes):

$$\forall r > 0,\ \forall p \in \Pi\ :\ |HO(p,r)| > n/2.$$

Let $\mathcal{A} = \langle S_p^r, T_p^r \rangle$ be an HO algorithm. A problem is solved by a pair $\langle \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{P}$ is a communication predicate. The consensus problem is specified by the following conditions:

- *Integrity:* Any decision value is the initial value of some process.
- *Agreement:* No two processes decide differently.
- *Termination:* All processes eventually decide.

The termination condition requires all processes to decide; a weaker condition is considered later. An example of a consensus algorithm is given by Algorithm 1.[5] The sending function is specified in lines 4–5. When the transition

Algorithm 1 The *OneThirdRule* algorithm [6].

```
1:  Initialization:
2:    x_p ← v_p
3:  Round r:
4:    S_p^r :
5:      send ⟨x_p⟩ to all processes
6:    T_p^r :
7:      if |HO(p,r)| > 2n/3 then
8:        if the values received, except at most ⌊n/3⌋, are equal
          to x̄ then
9:          x_p ← x̄
10:       else
11:         x_p ← smallest x_q received
12:       if more than 2n/3 values received are equal to x̄ then
13:         DECIDE(x̄)
```

[4] The term is taken from [21], in which transmission faults are considered in the context of synchronous systems.

[5] We have chosen this algorithm, rather than Paxos or another algorithm, for its simplicity. It allows us to keep the algorithmic part as simple as possible.

function (lines 6–13) is called, messages are available such that the predicate on the HO sets is guaranteed to hold. The consensus problem is solved by Algorithm 1 and the communication predicate $\mathcal{P}_{otr}$, given in Table 1 (next page).

Theorem 1. *The pair $\langle$ Algorithm 1, $\mathcal{P}_{otr}\rangle$ solves consensus.*

Proof. Algorithm 1 never violates the safety properties of consensus, namely integrity and agreement. For agreement, if some process decides v at line 13 of round r, then in any round $r' \geq r$, only v can be assigned to any x_p, and hence only v can be decided. Predicate $\mathcal{P}_{otr}$ ensures the liveness property of consensus (termination). The first part of $\mathcal{P}_{otr}$, namely the existence of some round r_0 in which all processes in Π have the set HO equal to some (large enough) set Π_0, ensures that at the end of round r_0 all processes in Π adopt the same value for x_p. The second part of $\mathcal{P}_{otr}$ forces every process $p \in \Pi$ to make a decision at the end of round r_p. □

Note that $\mathcal{P}_{otr}$ allows rounds in which no messages are received.

3.2. Restricted scope communication predicates

Section 2.3 has introduced the "transmission fault" abstraction, which covers various types of faults. One instantiation is to assume that transmission faults abstract link faults, send-omission faults and receive-omission faults, but not process crashes (i.e., processes do not crash). In this case the predicate $\mathcal{P}_{otr}$, which expresses a condition that must hold for all processes $p \in \Pi$, is perfectly adapted. This interpretation of transmission faults is also consistent with the termination condition for consensus that requires all processes to decide.

Let us now assume that transmission faults include in addition process crashes (without recovery). As already mentioned in [6], from the viewpoint of an HO algorithm this is still not a problem, since a crashed process does not send any messages and is thus indistinguishable from one that receives all messages but sends no messages. This holds no more if we implement the HO machine in a system where processes may exhibit any sort of benign faults. The problem can be addressed by restricting the scope of $\mathcal{P}_{otr}$ to the subset Π_0, as defined by $\mathcal{P}^{restr}_{otr}$, see Table 1 (next page).

Predicate $\mathcal{P}^{restr}_{otr}$ sets a requirement only for processes in Π_0, and so ensures termination only for processes in Π_0. If processes in Π_0 do not crash, while processes in $\Pi \setminus \Pi_0$ crash, then $\langle$Algorithm 1, $\mathcal{P}^{restr}_{otr}\rangle$ allow all processes that do not crash to decide. So we have:

Theorem 2. *The pair $\langle$Algorithm 1, $\mathcal{P}^{restr}_{otr}\rangle$ ensures the validity and agreement property of consensus. Moreover, all processes in Π_0 eventually decide.*

Proof. Proof of Theorem 1, by replacing Π with Π_0. □

3.3. Crash-recovery model

Algorithm 1 with predicate $\mathcal{P}^{restr}_{otr}$ solves consensus with process crashes (crash-stop), link faults, send-omission, and receive-omission faults. In Section 2.1 we pointed out the gap between solving consensus with failure detectors in the crash-stop *vs.* the crash-recovery model. The gap disappears with the transmission fault abstraction and communication predicates.

Without any changes, Algorithm 1 can be used in the crash-recovery model. Handling of recoveries is done at a lower layer (cf. Section 4).

4. Achieving predicate $\mathcal{P}^{restr}_{otr}$ in good periods

We discuss now the implementation of the communication predicate $\mathcal{P}^{restr}_{otr}$ introduced in Section 3. Figure 1 shows the algorithmic HO layer, the predicate implementation layer that we discuss now, and the interface between these two layers defined by communication predicates. This illustration shows also that the implementation of the predicates relies on assumptions about the underlying system (these assumptions define the fault and synchrony hypothesis). Note that "transmission faults" is an abstraction relevant to the upper layer: This abstraction does not appear at the lower layer.

In our implementation model, the system alternates between *good* and *bad* periods. In a good period the synchrony and fault assumptions hold; in a bad period the behavior of the system is arbitrary (but malicious behavior is excluded). The idea is here to compute the minimal duration of a good period that allows us to implement the communication predicates, i.e., the minimal duration of a good period that allow Algorithm 1 to solve consensus.

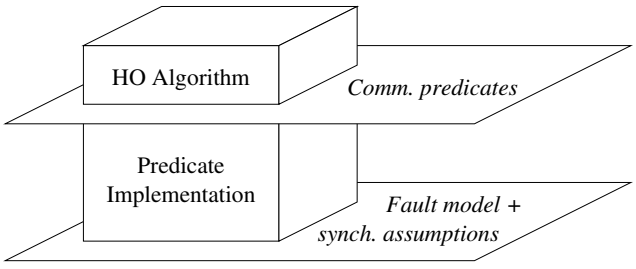

Figure 1. The two layers.

0-7695-2855-4/07 $20.00

$$\mathcal{P}_{otr}:: \exists r_0 > 0, \exists \Pi_0, |\Pi_0| > 2n/3 : (\forall p \in \Pi : HO(p, r_0) = \Pi_0) \wedge (\forall p \in \Pi, \exists r_p > r_0 : |HO(p, r_p)| > 2n/3) \quad (1)$$

$$\mathcal{P}^{restr}_{otr}:: \exists r_0 > 0, \exists \Pi_0, |\Pi_0| > 2n/3 : (\forall p \in \Pi_0 : HO(p, r_0) = \Pi_0) \wedge (\forall p \in \Pi_0, \exists r_p > r_0 : HO(p, r_p) \supseteq \Pi_0) \quad (2)$$

Table 1. Communication predicates

4.1. System model

Our system model is inspired by [12]; the differences are pointed out at the end of the section. We consider a message-passing system, and assume the existence of a fictitious global real-time clock that measures time with values from $\mathbb{R}$ (see the remark on the next page for the reason for considering values from $\mathbb{R}$ rather than integers). The clock is used only for analysis and is not accessible to the processes. Processes execute a sequence of atomic steps, which are either *send* steps or *receive* steps. As in [12], steps take no time (atomic steps), but time elapses between steps.[6] The network can take a *make-ready* step that is introduced to distinguish a message ready for reception from a message in transit: (i) Every process has two sets of messages called $network_p$ and $buffer_p$; (ii) a make-ready step transfers a message from the first to the second set. Send steps, receive steps, and make-ready steps are defined to adequately model a real system:

- In a *send* step, a process p sends a message to either a single process or to all other processes and makes some local computation. More precisely, if p executes $send_p(m)$ to all, then m is put into $network_s$, for all $s \in \Pi$.
- In a *make-ready* step, the network transfers some messages from $network_p$ into $buffer_p$. More precisely, if the network executes *make-ready*$_p(M)$ for some subset $M \subseteq network_p$, all messages $m \in M$ are removed from $network_p$ and put into $buffer_p$. Messages in $buffer_p$ are *ready for reception* by process p.
- In a *receive* step executed at time t, a process p may receive a *single* message that was in $buffer_p$ at time t and makes some local computation. *So n receive steps are needed to receive n messages.* If $buffer_p = \emptyset$ at the time of a receive step, the empty message λ is received. A process p may specify any policy, according to which the message $buffer_p$ is selected for reception (e.g., *"message with the largest round number first"*).

We consider that the system alternates between good and bad periods. In a bad period, processes can crash and recover and suffer from send and receive omission; furthermore links can loose messages. We distinguish three types of good periods, from the strongest to the weakest. All these definitions refer to a subset π_0 of Π. In all the three definitions, the following property π_0-*sync* holds in a good period for processes in π_0:

π_0-sync: The subsystem π_0 is synchronous, i.e., there is a known upper and lower bound on the process speed and a known upper bound on the communication delays among processes in π_0. Formally:

Let I be an open contiguous time interval and R a run. Processes and links are synchronous during I if there exist $\Phi^+, \Phi^-, \Delta \in \mathbb{R}$ such that:

- In any contiguous sub-interval of I of length Φ^+, every process in π_0 takes at least one step.
- In any open contiguous sub-interval of I of length Φ^-, every process in π_0 takes at most one step.
- Consider two processes $p, q \in \pi_0$. If process p executes $send_p(m)$ at time $t \in I$, then $m \in buffer_q$ at time $t + \Delta$, provided that $t + \Delta \in I$.

The length of the good period is $|I|$. If I starts at time 0, we say I is an *initial good period*. We denote $\Pi \setminus \pi_0$ by $\overline{\pi_0}$. We can now define the three types of good periods:

1. **Π-good period:** The property π_0-sync holds for $\pi_0 = \Pi$. All processes are up, none of these processes crashes (during the good period).

2. **"$\overline{\pi_0}$-down" good period:** The property π_0-sync holds for $\pi_0 \subseteq \Pi$. Processes in π_0 do not crash. Processes in $\overline{\pi_0}$ are down and do not recover (during the good period). Moreover, no messages from processes in $\overline{\pi_0}$ are in transit during the good period.

3. **"$\overline{\pi_0}$-arbitrary" good period:** The property π_0-sync holds for $\pi_0 \subseteq \Pi$. There are no restrictions on the processes in $\overline{\pi_0}$ and on the links to and from processes processes in $\overline{\pi_0}$ (during the good period processes in $\overline{\pi_0}$ can crash, recover, be asynchronous; links to and from processes in $\overline{\pi_0}$ can lose messages, be asynchronous).

Case 2 includes case 1, and case 1 leads to the same implementation as case 2. Thus we distinguish below only between case 2 and case 3. For simplicity, we will use the following notation: We scale all values Φ^+, Φ^-, Δ, and t

[6] We model a step that "terminates" at time t as an atomic step that "occurs" at time t.

with $1/\Phi^-$ and use $\phi = \Phi^+/\Phi^-$ as the normalized upper bound of the process speed, $\delta = \Delta/\Phi^-$ as the normalized transmission delay, and $\tau = t/\Phi^-$ as normalized time. Remember that ϕ and δ are "known" values, and note that *these values are unit-less*.[7]

Remark: For our modeling, we have chosen real-values clocks to represent time. Consider case 3 above, assuming integer clock values instead. By the definition of Φ^+, the slowest process in π_0 takes at least one step in any interval Φ^+. However, with integer clock values, any process can take at most Φ^+ steps in an interval Φ^+, independent how small Φ^- is chosen. So, in case 3, processes in $\overline{\pi_0}$ cannot be arbitrarily fast with respect to processes in π_0. In other words, with integer clock values, processes in $\overline{\pi_0}$ have some synchrony relation with respect to processes in π_0, which we wanted to exclude under case 3.

Differences between our system model and DLS [12]: In [12] the clocks take integer values. We have explained the reason to consider clocks with real-time values. In [12] a send step allows a process to send a message only to a single destination. Our send primitive allows messages to be broadcast, a facility provided, e.g., by UDP-multicast. In [12], a receive step allows a process to receive several messages. Our receive primitive allows reception of a message from one single process only, which reflects the feature, e.g., of UDP. The reception of messages one by one led us to introduce the make-ready step. Two different synchrony assumptions are considered in [12]: (i) The synchrony bounds are known but hold only eventually; (ii) the synchrony bounds are not known, but hold from the beginning. We considered option (i), which is needed to compute the minimal length of a good period (in the context of the implementation of the communication predicates). In the context of option (i), [12] assumes that the good period holds eventually forever and that the synchrony assumption holds on the whole system. We consider the system alternating between good and bad periods, and synchrony assumptions that hold only on a subset π_0. We also assume the more general crash-recovery model, while [12] considers the crash-stop model. On the other hand, contrary to our fault model, [12] considers also Byzantine faults.

4.2. Implementation of $\mathcal{P}^{restr}_{otr}$

We give now algorithms for implementing the predicate $\mathcal{P}^{restr}_{otr}$ in $\overline{\pi_0}$-down and $\overline{\pi_0}$-arbitrary good periods. It turns out that both definitions of a good period lead naturally to the implementation of a predicate that is stronger than $\mathcal{P}^{restr}_{otr}$. We define:

$$
\begin{aligned}
\mathcal{P}_{su}(\Pi_0, r_1, r_2) &:: \forall p \in \Pi_0, \forall r \in [r_1, r_2] : HO(p, r) = \Pi_0 \\
\mathcal{P}_{k}(\Pi_0, r_1, r_2) &:: \forall p \in \Pi_0, \forall r \in [r_1, r_2] : HO(p, r) \supseteq \Pi_0 \\
\mathcal{P}^{2}_{otr}(\Pi_0) &:: \exists r_0 > 0 : \mathcal{P}_{su}(\Pi_0, r_0, r_0) \\
&\qquad \wedge \mathcal{P}_{k}(\Pi_0, r_0{+}1, r_0{+}1) \\
\mathcal{P}^{1/1}_{otr}(\Pi_0) &:: \exists r_0 > 0, \exists r_1 > r_0 : \mathcal{P}_{su}(\Pi_0, r_0, r_0) \\
&\qquad \wedge \mathcal{P}_{k}(\Pi_0, r_1, r_1)
\end{aligned}
$$

Predicate $\mathcal{P}_{su}(\Pi_0, r_1, r_2)$ ensures that rounds from r_1 to r_2 are so called "space uniform" for the processes in Π_0. Predicate $\mathcal{P}_{k}(\Pi_0, r_1, r_2)$ ensures a weaker property (k stands for *kernel*). Predicate $\mathcal{P}^{2}_{otr}(\Pi_0)$ ensures two consecutive rounds such that the first satisfies $\mathcal{P}_{su}(\Pi_0, -, -)$ and the second $\mathcal{P}_{k}(\Pi_0, -, -)$. Predicate $\mathcal{P}^{1/1}_{otr}(\Pi_0)$ ensures the same property for two rounds that do not need to be consecutive. We clearly have:

$$
\begin{aligned}
(\exists \Pi_0,\ s.t.\ |\Pi_0| > 2n/3 : \mathcal{P}^{2}_{otr}(\Pi_0)) &\Rightarrow \mathcal{P}^{restr}_{otr} \\
(\exists \Pi_0,\ s.t.\ |\Pi_0| > 2n/3 : \mathcal{P}^{1/1}_{otr}(\Pi_0)) &\Rightarrow \mathcal{P}^{restr}_{otr}.
\end{aligned}
$$

We give below algorithms for $\mathcal{P}_{su}(-,-,-)$ and $\mathcal{P}_{k}(-,-,-)$, for both definitions of good periods. We also analyze the timing property of the algorithms under the following two scenarios:

1. Assume that a good period starts at an arbitrary time t_G resp. $\tau_G = t_G/\Phi^-$. We compute, in the worst case, the minimal length of the good period needed to implement the communication predicates. We call this value *minimal length of a good period*.
2. We do the same, assuming that a good period starts from the beginning, i.e., $\tau_G = 0$. We call this value *minimal length of an initial good period*.

Intuitively, scenario 2 allows us to compute the time to solve consensus in the fault-free case, which is often called a "nice" run. Scenario 1 allows us a timing analysis of consensus in "not nice" runs.

4.2.1. Ensuring $\mathcal{P}^{restr}_{otr}$ in a "$\overline{\pi_0}$-down" good period

Let us consider a "$\overline{\pi_0}$-down" good period that is "long enough", with π_0 arbitrary. Algorithm 2 implements $\mathcal{P}_{su}(\pi_0, -, -)$. The function $S^{r_p}_p$ at line 7 returns the message to be sent; the send occurs at line 8. Variable i_p (line 9, 11) counts the number of receive steps. If p executes x steps, at least x and at most $x\phi$ (normalized) time has elapsed (see Section 4.1). Process p executes at most $\lceil 2\delta + n + 2\phi \rceil$ receive steps, see line 12 (message reception takes place at line 14; non-empty messages are added to the set $msgsRcv_p$, see line 16). Process p executes receive steps (1) until $\lceil 2\delta + n + 2\phi \rceil$ receive steps have been executed, or (2) if p receives a message from a round r' larger

[7] For obtaining real-time values, the results in this section have thus to be multiplied by Φ^-.

0-7695-2855-4/07 $20.00

Algorithm 2 Ensuring $\mathcal{P}_{su}(\pi_0, -, -)$ with a "$\overline{\pi_0}$-down" good period

```
1:  Reception policy: Highest round number first
2:  msgsRcv_p ← ∅                                   {set of messages received}
3:  r_p ← 1                                         {round number}
4:  next_r_p ← 1                                    {next round number}
5:  s_p ← init_p                                    {state of the consensus algorithm}
6:  while true do
7:     msg ← S_p^{r_p}(s_p)
8:     send ⟨msg, r_p⟩ to all
9:     i_p ← 0
10:    while next_r_p = r_p do
11:       i_p ← i_p + 1
12:       if i_p ≥ 2δ + n + 2φ then
13:          next_r_p ← r_p + 1;
14:       receive a message
15:       if message is ⟨msg, r'⟩ from q then
16:          msgsRcv_p ← msgsRcv_p ∪ {⟨msg, r', q⟩}
17:          if r' > r_p then
18:             next_r_p ← r'
19:    R ← {⟨msg', q'⟩ | ⟨msg', r_p, q'⟩ ∈ msgsRcv_p}
20:    s_p ← T_p^{r_p}(R, s_p)
21:    forall r' in [r_p+1, next_r_p−1] do s_p ← T_p^{r'}(∅, s_p)
22:    r_p ← next_r_p
```

than r_p. In both cases the state transition function $T_p^{r_p}$ is executed with the set R of messages received in round r_p (line 20). Then the state transition function $T_p^{r_p}$ is executed for all rounds $r_p + 1$ to $next\_r_p - 1$ with an empty set of messages.[8]

In order to cope with recoveries after crashes, variables r_p and s_p are stored on stable storage. In case of a recovery, the algorithm starts on line 6, with $msgsRcv_p$ and $next\_r_p$ reinitialized. Reading variables on stable storage is inefficient. The implementation can be made more efficient by keeping a copy of the variables in main memory: a read operation reads the *in memory* copy, a write operation updates the *in memory* and the *stable storage* copies. Upon recovery, the in memory copy is reset with the value of the stable copy.[9]

Algorithm 2 is not optimized regarding space, i.e., the set $msgsRcv_p$ grows forever. Obviously, messages for round smaller than r_p can safely be discarded. To keep the presentation short, we did not include this simple optimization.

It should be noted that Algorithm 2 relies exclusively on messages sent by the upper algorithmic layer: Algorithm 2 does not send any additional message.

We prove Algorithm 2 in two steps. First we prove that there exists $r > 0$ such that, for any $x > 0$, Algorithm 2 ensures $\mathcal{P}_{su}(\pi_0, r, r+x-1)$, assuming a "long enough" good period. Then we compute the minimal duration of a good period to ensure $\mathcal{P}^2_{otr}(\pi_0)$, and the minimal duration of two good periods to ensure $\mathcal{P}^{1/1}_{otr}(\pi_0)$. Note that by the definition of a $\overline{\pi_0}$-down good period, all processes in $\overline{\pi_0}$ are down in a good period, and no messages from these processes are in transit in the good period. In other words, processes in $\overline{\pi_0}$ can simply be ignored.

Theorem 3. *With Algorithm 2, the minimal length of a good period to achieve $\mathcal{P}_{su}(\pi_0, \rho_0, \rho_0+x-1)$ is:*

$$(x+1)(2\delta + n + 2\phi + 1)\phi + \delta + \phi.$$

The proof, also for all other theorems of this paper, can be found in [16]. The following Corollary follows directly from Theorem 3 with x=1 and x=2, and the fact that $\mathcal{P}_{su}(-,-,-) \Rightarrow \mathcal{P}_k(-,-,-)$:

Corollary 4. *For implementing $\mathcal{P}^2_{otr}(\pi_0)$ with Algorithm 2, we need one "$\overline{\pi_0}$-down" good period of length*

$$(6\delta + 3n + 3 + 6\phi)\phi + \delta + \phi.$$

For implementing $\mathcal{P}^{1/1}_{otr}(\pi_0)$ with Algorithm 2, we need two "$\overline{\pi_0}$-down" good periods of length

$$(4\delta + 2n + 2 + 4\phi)\phi + \delta + \phi.$$

Corollary 4 shows an interesting trade-off in terms of the length of a good period. The next theorem gives us the minimal length of an *initial* good period:

Theorem 5. *With Algorithm 2, the minimal length of an initial good period to achieve $\mathcal{P}_{su}(\pi_0, 1, x)$ is:*

$$x(2\delta + n + 2\phi + 1)\phi.$$

As already pointed out, Theorem 5 is related to so-called "nice" runs, while Theorem 3 is related to "not nice" runs. This second case has not been addressed in the literature with a time analysis as done here (see Section 5). The results show a factor of approximately $3/2$ between the two cases for the relevant value $x = 2$.

4.2.2. Ensuring $\mathcal{P}^{restr}_{otr}$ in a "$\overline{\pi_0}$-arbitrary" good period

In this section we consider a $\overline{\pi_0}$-arbitrary good period. Compared with the previous section, the problem is more complex. We proceed in two steps. First we show how to implement the predicate $\mathcal{P}_k(\pi_0, -, -)$. Second, we show how to obtain the predicate $\mathcal{P}_{su}(\pi_0, -, -)$ from $\mathcal{P}_k(\pi_0, -, -)$. Note that we introduce here a parameter f defined such that $|\pi_0| = n - f$. The implementation of $\mathcal{P}_k(\pi_0, -, -)$ requires $f < n/2$.

[8] This is required only if $T_p^{r_p}(\emptyset, s_p) \neq s_p$. Calling the sending function $S_p^{r_p}$ is not needed, since the function does not change the state s_p.

[9] We could express this formally as a variant of Algorithm 2, but the space constraints prevent us from doing this.

0-7695-2855-4/07 $20.00

a) Implementing $\mathcal{P}_k(\pi_0, -, -)$

The algorithm for implementing $\mathcal{P}_k(\pi_0, -, -)$ is given as Algorithm 3. It uses two different types of messages, INIT messages and ROUND messages. Processes express the intention to enter a new round ρ with an $\langle \text{INIT}, \rho, - \rangle$ message. If a process receives at least $f + 1$ INIT messages for some round ρ, it starts round ρ and sends a $\langle \text{ROUND}, \rho, - \rangle$ message. A process in round ρ that receives a ROUND message for a higher round ρ' enters immediately round ρ'. This ensures fast synchronization at the beginning of a good period, and is one of the major differences of this algorithm compared to Byzantine clock synchronization algorithms.

The reception policy for Algorithm 3 (line 1) is a little bit more complicated than for Algorithm 2. Algorithm 3 has to ensure that a fast process with a large round number r' is not able to prevent messages from other processes with lower round numbers $r < r'$ from being received. The reception policy is as follows: At the ith receive step, the message with the highest round number from process $p_{i \bmod n}$ is selected for reception. If no such message exists, an arbitrary message is selected.

As for the previous algorithm, variables r_p and s_p are assumed to be on stable storage (possibly with a copy in volatile memory) and the algorithm starts after a recovery in line 6, with $msgsRcv_p$ and $next\_r_p$ reinitialized.

We prove the following results:

Theorem 6. *With Algorithm 3 and $f < n/2$, the minimal length of a good period to achieve $\mathcal{P}_k(\pi_0, \rho_0, \rho_0 + x - 1)$ is*

$$(x + 2)[\tau_0\phi + \delta + n\phi + 2\phi] + \tau_0\phi =$$

$$= (x+2)[(2\delta+n\phi+\phi)\phi+\delta+2n\phi+2\phi]+(2\delta+n+n\phi+\phi)\phi$$

Theorem 7. *With Algorithm 3 and $f < n/2$, the minimal length of an initial good period to implement $\mathcal{P}_k(\pi_0, 1, x)$ is:*

$$(x - 1)[\tau_0\phi + \delta + n\phi + 2\phi] + \tau_0\phi + \phi.$$

b) Implementing $\mathcal{P}_{su}(\pi_0, -, -)$ from $\mathcal{P}_k(\pi_0, -, -)$

We show now that $f+1$ rounds that satisfy $\mathcal{P}_k(\pi_0, -, -)$, with $|\pi_0| = n - f$, allow us to construct one *macro-round* that satisfies $\mathcal{P}_{su}(\pi_0, -, -)$. The "translation" is given by Algorithm 4, which is derived from a similar translation in [6]. Let $r_1, \ldots, r_{f+1}$ denote the sequence of the $f + 1$ rounds that form a macro-round $\mathcal{R}$. In round r_1, every process p sends its message for macro-round $\mathcal{R}$ (line 7). In all subsequent rounds $r_2, \ldots, r_{f+1}$ messages previously received are relayed (line 7). In round r_{f+1} (i.e., $r \equiv 0 \pmod{f+1}$, see line 9), the set of messages of macro-round $\mathcal{R}$ to be received by p are computed (lines 13 and 14).

Algorithm 3 Ensuring $\mathcal{P}_k(\pi_0, ,)$ with a "$\overline{\pi_0}$-arbitrary" good period

```
1:  Reception policy: The highest round message from each process in a round robin fashion
2:  msgsRcv_p ← ∅
3:  r_p ← 1
4:  next_r_p ← 1
5:  s_p ← init_p
6:  while true do
7:      msg ← S_p^{r_p}(s_p)
8:      send ⟨ROUND, r_p, msg⟩ to all
9:      i ← 0
10:     while next_r_p = r_p do
11:         receive a message
12:         if message is ⟨ROUND, msg, r'⟩ or ⟨INIT, msg, r' + 1⟩ from q then
13:             msgsRcv_p ← msgsRcv_p ∪ {⟨msg, r', q⟩}
14:             if r' > r_p then
15:                 next_r_p ← r'
16:         if received f+1 messages ⟨INIT, r_p+1, −⟩ from distinct processes then
17:             next_r_p ← max{r_p + 1, next_r_p}
18:         i ← i + 1
19:         if i ≥ 2δ + n + nφ + φ then
20:             send ⟨INIT, r_p + 1, msg⟩ to all
21:     R ← {⟨msg', q'⟩ | ⟨msg', r_p, q'⟩ ∈ msgsRcv_p}
22:     s_p ← T_p^{r_p}(R, s_p)
23:     forall r' in [r_p+1, next_r_p−1] do s_p ← T_p^{r'}(∅, s_p)
24:     r_p ← next_r_p
```

c) Putting it all together

When combining Algorithm 3 and Algorithm 4, the function $S_p^{r_p}(\,)$ called in Algorithm 3 refers to line 7 of Algorithm 4. Similarly, the function $T_p^{r_p}(\,)$ called in Algorithm 3 refers to the lines 9 to 17 of Algorithm 4. The functions S_p^r and T_p^r in Algorithm 4 refer to the sending phase and state transition phase of Algorithm 1.

We compute now the minimal duration of a good period to ensure $\mathcal{P}^2_{otr}(\pi_0)$ (considering instead $\mathcal{P}^{1/1}_{otr}(\pi_0)$ is not a valuable alternative here):

1. We need first $f + 1$ rounds that satisfy $\mathcal{P}_k(-, -, -)$ to implement one macro-round that satisfies $\mathcal{P}_{su}(-, -, -)$ (Algorithm 4).
2. Then we need one round that satisfies $\mathcal{P}_k(-, -, -)$.

For 1, the worst case happens when the good period starts immediately after the beginning of a macro-round. In this worst case, $\mathcal{P}_{su}(-, -, -)$ requires two macro-rounds. Since one macro-round consists of $f+1$ rounds, in the worst case we need $2(f+1)$ rounds. Item 2 adds one round. So we end up with a minimal duration of $2f + 3$ rounds. Applying Theorem 6, we get the minimal length of a good period:

$$(2f+5)[(2\delta+n\phi+\phi)\phi+\delta+2n\phi+2\phi]+(2\delta+n+n\phi+\phi)\phi.$$

Algorithm 4 Ensuring $\mathcal{P}_{su}(\pi_0, -, -)$ with $\mathcal{P}_k(\pi_0, -, -)$ (adapted from [6]).

1: **Variables:**
2: $Listen_p$, initially Π {*set of processes*}
3: $NewHO_p$ {*set of processes*}
4: $Known_p$, initially $\{\langle S_p^{\mathcal{R}_p}(s_p), p\rangle\}$ {*set of* $\langle message, process\rangle$ }
5: **Round** r:
6: S_p^r :
7: send $\langle Known_p\rangle$ to all processes
8: T_p^r :
9: $Listen_p \leftarrow Listen_p \cap \{q \mid \langle Known_q\rangle \text{ received}\}$
10: **if** $r \not\equiv 0 \pmod{f+1}$ **then**
11: $Known_p \leftarrow Known_p \cup \bigcup_{q \in Listen_p} Known_q$
12: **else**
13: $NewHO_p \leftarrow \{s \mid \langle -, s\rangle \in Known_q \text{ for } n-f \text{ processes } q \in Listen_p\}$
14: $R \leftarrow \{\langle msg, s\rangle \mid s \in NewHO_p\}$
15: $s_p \leftarrow T_p^{\mathcal{R}_p}(R, s_p)$
16: $Listen_p \leftarrow \Pi$
17: $Known_p \leftarrow \{\langle S_p^{\mathcal{R}_p}(s_p), p\rangle\}$

5 Related work

The paper addresses several issues that appear in the literature. We now point out the key differences.

The HO model was proposed in [6]. The paper establishes relationship among several communication predicates and identifies the weakest predicate, among the class of predicates with non-empty kernel rounds, for solving consensus. The paper also expresses well-known consensus algorithms (or variants) and new ones in the HO model, with the goal of showing the expressiveness of the model. The implementation of communication predicates is not addressed in [6], nor is the ability of the model to handle uniformly crash-stop and crash-recovery models, and the reason for that. In [7] the HO model is used to express a new consensus algorithm.

The HO model generalizes the round model of [12], but does not reintroduce failure detectors as done in [14] and in [17]. The implementation in [12], contrary to ours, explicitly refers to some "common notion of time" and relies on a distributed clock synchronization algorithm.

It has been sometimes claimed that the partial synchrony model has been superseded by the failure detector model [5]. In out opinion this claim is only partially correct. The models that extend the failure detector model, e.g., [14, 17], all inherit from the limitations of failure detectors pointed out in Section 1.

The issue of performance of consensus following asynchronous periods is considered in [11, 17]. In [17] the focus is on number of rounds rather than time; [11] considers time. Moreover, in [17] the authors write that being able to quantify the time it takes the environment to reach round synchronization after the system has stabilized is an interesting subject for further studies. This question is answered here. In [11] and [17] the synchronous period is defined only by properties of links: Processes are always considered to be synchronous. This is in contrast to our definition of $\overline{\pi_0}$-arbitrary good period, where only a subset of processes are assumed to be synchronous. This definition opens the door to the analysis of the duration of good periods with Byzantine processes. Our algorithm shares some similarities with the Byzantine clock synchronization of [23]. However, the algorithm in [23] assumes reliable links; adapting the algorithm to message loss, we end up with the algorithm of [12].

The notion of good and bad period appears in [8], but the issue of the length of a good period for solving consensus is not addressed. Restricting the scope of synchrony, as we do in good periods, has been considered in other settings, e.g., [15] and [2, 3]. However, in all these papers the issue of synchrony is implicitly restricted to links (i.e., process synchrony is not addressed). This is not the case in our definition of $\overline{\pi_0}$-arbitrary good period.

The Paxos algorithm [19] does not assume reliable links and, because of this, works under the crash-recovery model with stable storage. However, the condition for liveness is not expressed by a clean abstraction as done by communication predicates in the HO model (a consensus algorithm *à la Paxos* in the HO model can be found in [6]). The same comment applies to [11], where the system must stabilize before consensus is reached. System stabilization is not required with $\overline{\pi_0}$-arbitrary good periods: the HO model provides a clean separation of concerns between the HO algorithmic layer and the predicate implementation layer, which allows a finer definition of good periods, and so a finer timing analysis. As pointed out in Section 3, we have chosen here an algorithm that is simpler than Paxos to illustrate as simply as possible the approach based on communication predicates.

The notion of transmission faults was suggested in [21], however only in the context of synchronous systems. Varying the quorums for "init" and round messages — in the context of $\overline{\pi_0}$-arbitrary good periods — was to our knowledge done first in [20, 24], but for other fault scenarios.

6 Conclusion

Abstractions are essential when solving difficult problems. Failure detectors provide a nice abstraction for solving the difficult consensus problem; this explains why they have been widely adopted. However, transient and dynamic faults show the limitations of the failure detector approach: For example, solving consensus in the crash-stop model and

in the crash-recovery model leads to significantly different solutions. The HO model provides a different abstraction, namely the "communication predicates", which allow us to handle uniformly static, dynamic, transient, and permanent faults and so overcome the limitations of failure detectors. Moreover, the HO model allows a nice and concise expression of consensus algorithms.

Similarly to failure detectors, solving consensus in the HO model leads to distinguish two layers: The "algorithmic" layer and the "abstraction" layer (the layer at which the abstraction is implemented). In the case of failure detectors, the abstraction layer must ensure the properties of the failure detectors, based on assumptions of the underlying system. The same holds for communication predicates. However, while communication predicates are based on the very general notion of transmission faults, failure detector assume the limited notion of process crash faults. The communication predicate layer defines a larger "playground" than the failure detector playground, in which more issues can be addressed. Specifically, the communication predicate approach has allowed us to bring an answer the question raised in [17], about quantifying the time it takes to reach round synchronization after the system has stabilized.

Acknowledgments We like to thank Bernadette Charron-Bost, Nicolas Schiper, Martin Biely, Josef Widder, Nuno Santos, Sergio Mena, and the anonymous reviewers for their valuable comments that helped us to improve the paper.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proc. PODC'03*. ACM Press, 2003.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proc. PODC'04*, pages 328–337. ACM Press, 2004.

[4] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Reconstructing Paxos. *ACM SIGACT News*, 34(1):47–67, 2003.

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

[6] B. Charron-Bost and A. Schiper. The "Heard-Of" model: Unifying all benign faults. Technical Report TR, EPFL, June 2006.

[7] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. In *Pacific Rim Dependable Computing, Proceedings*, 2006.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[9] A. Doudou and A. Schiper. Muteness Failure Detectors for Consensus with Byzantine Processes. TR 97/230, EPFL, Dept d'Informatique, October 1997.

[10] A. Doudou and A. Schiper. Muteness detectors for consensus with byzantine processes (Brief Announcement). In *Proc. PODC'98*, Puerto Vallarta, Mexico, July 1998.

[11] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *Proc. DSN'05*, pages 22–27, Los Alamitos, CA, USA, 2005.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[14] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proc of the 17th ACM Symp. Principles of Distributed Computing (PODC)*, pages 143–152, Puerto Vallarta, Mexico, June-July 1998.

[15] R. Guerraoui and A. Schiper. "gamma-accurate" failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, pages 269–286, London, UK, 1996. Springer-Verlag.

[16] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. Technical Report LSR-REPORT-2006-006, EPFL, 2006. http://infoscience.epfl.ch/search.py?recid=97290.

[17] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *Proc. PODC'06*, pages 169–178, New York, NY, USA, 2006. ACM Press.

[18] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, Chantilly, France, Dec. 1997.

[19] L. Lamport. The Part-Time Parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

[20] G. Le Lann and U. Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Technische Universität Wien, Department of Automation, Jan. 2003.

[21] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings of the 6th Symposium on Theor. Aspects of Computer Science*, pages 304–313, Paderborn, Germany, 1989.

[22] F. B. Schneider. What Good are Models and What Models are Good. In S. Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

[23] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[24] J. Widder. Booting clock synchronization in partially synchronous systems. In *Proceedings of the 17th International Conference on Distributed Computing (DISC'03)*, pages 121–135, 2003.