Failure Detectors in Omission Failure Environments*

Danny Dolev[†] Roy Friedman[‡] Idit Keidar[†] Dahlia Malkhi[§]

September 19, 1996

Abstract

We study failure detectors in an asynchronous environment that admits message omission failures. In such environments, processes may fail by crashing, but may also *disconnect* from each other. We adapt Chandra and Toueg's definitions of failure detection completeness and accuracy to the omissions failure model, and define a weak failure detector $\diamond\mathcal{W}(om)$ that allows any majority of the processes that become connected to reach a Consensus decision, despite any number of transient communication failures in their past. We provide a protocol that solves the Consensus problem in this model whenever a majority of the processes become connected, regardless of past omissions. Moreover, in our protocol it is not necessary to save and repeatedly send all past messages, which makes it more efficient than previous protocols in this model.

*This work was supported by ARPA/ONR grants N00014-92-J-1866 and F30602-95-1-0047, and by grants from IBM, Siemens, and GTE Corporations and by the Ministry of Science of Israel, grant number 0327452.

[†]Computer Science Institute, The Hebrew University of Jerusalem. Email: `{idish,dolev}@cs.huji.ac.il`

[‡]Department of Computer Science, Cornell University. Email: `roy@cs.cornell.edu`

[§]AT&T Laboratories, Murray Hill, NJ 07974. Email: `dalia@research.att.com`

1 Introduction

Failure detectors are useful abstractions for specifying services and protocols in a distributed environment prone to failures. Failure detectors provide a clear analysis of the effects of failures on the solvability of certain problems in distributed environments. Examples of services that are defined using failure detectors include Consensus [12], atomic commit protocols [7, 9] and primary component membership [11] services.

In a seminal paper, Chandra and Toueg categorized failure detectors according to their degree of *accuracy* and *completeness* [4]. In particular, they defined an *eventual weak failure detector* $\diamond\mathcal{W}$ that can make an infinite number of mistakes but eventually some restrictions on its behavior hold. They present a protocol for solving Consensus using $\diamond\mathcal{W}$. The failure detectors definitions and protocols presented in [4] are described in a model that is limited to crash failures only and cannot be trivially extended to allow message omissions.

We extend the definitions of failure detectors to a more general asynchronous model with all benign types of failures: In our model processes may crash and recover, messages may be lost or arbitrarily delayed, the network may partition and previously disconnected network components may re-merge. We start by defining the *correct* and *faulty* processes of this environment in such a way that allows two correct processes to experience transient communication failures for any unbounded (but finite) period. In our model, the system eventually converges to a *stable* state in which all the correct processes reliably communicate with each other, and are disconnected from all faulty processes. Note that the stable state is observed only externally, as an analysis tool of the system, and the processes need not be aware of it. Furthermore a process need not know whether it is correct or not. Note also that, while stability has to formally hold forever, in any execution of our algorithms it needs to hold only for a finite time (though no prior bound on it exists).

We define classes of failure detectors in our environment. Our definitions follow closely the ones given in [4]. In particular, we define the *eventually weak failure detector* class, $\diamond\mathcal{W}(om)$. This class consists of weak, unreliable failure detectors that satisfy *Weak Completeness*, i.e., that every faulty process is eventually suspected by some correct process, and *Eventual Weak Accuracy*, i.e., that eventually there exists a correct process that is not suspected by any other correct process. We also define an *eventually strong failure detector* class, $\diamond\mathcal{S}(om)$, and prove the equivalence of the two classes, which is analogous to the equivalence of $\diamond\mathcal{W}$ and $\diamond\mathcal{S}$ in Chandra and Toueg's model.

We show that Consensus is solvable in our environment if no more than $\lfloor \frac{n}{2} \rfloor$ processes are faulty (where n is the number of processes in the system) and the environment is augmented with a failure detector of the class $\diamond\mathcal{W}(om)$. We begin by presenting a naive adaptation of the protocol in [4] to our environment. The naive protocol overcomes message loss by re-sending all past messages at each round. Unfortunately, this approach results in buffering requirements that grow linearly in the **total size** of exchanged data. We provide an alternative solution whose memory requirement is bounded logarithmically in the number of messages sent, and does not require storing nor re-sending of old messages, yet can overcome an unbounded (but finite) number of omission failures. The logarithmic space requirement stems from our lack of assumption on message ordering. Note that there can exist no constant memory protocols providing reliable multicast over an unreliable network that can re-order messages [1]. In our solution, the memory requirements increase logarithmically due to the use of monotonic counters. In any practical execution, 64 or 128 bits of memory will be more than enough. Therefore our solution is reasonable for practical use.

Finally, we prove minimality of $\diamond\mathcal{W}(om)$ for solving Consensus in our environment using similar arguments as the ones used in proving the minimality of $\diamond\mathcal{W}$ in the crash-failure environment model (see [3]).

1.1 Related Work

Guerraoui and Schiper introduced the notion of "Γ-Accurate" failure detectors in [8]. Their definitions are provided, again, in the crash failure model only. More precisely, they assume that messages between any pair of live processes are eventually received. Like the model in [4], over an unreliable network, this assumption requires unbounded message re-transmission capability. In a practical environment with message losses, in which live processes may omit an unbounded number of messages, the main reduction presented in [8] does not work.

In the *timed asynchronous* model, Cristian and Schmuck [6] explicitly use the notions of time and timeouts to state sufficient conditions for reaching agreement on the membership of a group of processes. As discussed in [13], timeouts are only one possible aspect of failure detection. Since our definitions and results use failure detectors as an abstract tool, our model is more general than the one presented in [6]. As in our model, their protocols are guaranteed to proceed only when the network *stabilizes*, with the important distinction that, in their work, a stable network delivers every message within a known delay.

Babaoglu, Davoli, and Montresor also define failure detectors for omission failure environments and use them to develop protocols for view-synchronous communication [2]. Their definitions of permanent *reachability* and *unreachability* require, like ours, eventual stability of the system, but unlike ours, capture pairwise connectivity only. Thus, in their model, only crashed processes are considered faulty, and consequently they do not provide a weakest failure detector for solving Consensus in omission failure environments.

2 The System Model and Definitions

The system contains a fixed set N of n processes which communicate via message passing. The set N is known to all processes. The underlying communication network provides *datagram message delivery*. There is no known bound on message transmission time, hence the system is *asynchronous*.

The datagram message delivery assumption captures the following types of possible failures: Messages may be lost or delivered out of order, failures may partition the network into several *components*, and previously disjoint network components may re-merge. Processes may crash and recover, but recovered processes come up with their stable storage intact and every state change in a process is logged on stable storage. Hence, our model does not distinguish processes that crash and later recover from "disconnected" ones, and from now on, when we say that a process crashes, we assume that it does not recover.

We assume also that each process is equipped with a failure detector, which provides hints regarding which processes may be faulty at any given time. The information provided by the failure detectors need not be accurate, although some restrictions on their behavior are imposed later in Section 3.

More formally, a process can be modeled as a (possibly infinite) automaton, which takes *steps* that consist of receiving `message-receive` events from the network and `suspect` lists from the failure detector, doing some local computation, and then generating zero or more `message-send` events. Events of type `message-receive` may be empty, i.e., containing null messages. (This allows processes to initiate operations spontaneously). A process can also receive a `crash` event, which is the last event it receives.

A *history* of a process is a sequence of events as they occur in that process, in which `crash` events are not followed by any other event. An *execution* is a collection of histories, one for each process, in which there is a mapping from each `message-receive` event to a corresponding `message-send` event. In this paper we consider only executions in which there are no causal cycles [5, 10].

An execution σ' is a *sub-execution* of another execution σ if they include histories of the same set of processes, and the history of each process p_i in σ' is a prefix of p_i's history in σ. Given an execution σ and a sub-execution σ' of σ, the collection of history suffixes resulted by eliminating the history of each process in σ' from its corresponding history in σ is an *extension* of σ'. We denote this extension by $\sigma \setminus \sigma'$.

An execution or a sub-execution is *infinite* if the history it contains for every process is either infinite or ends with a `crash` event.

Let σ be an infinite execution, σ' a sub-execution of σ, σ'' a sub-execution of σ', and let τ be the extension $\sigma' \setminus \sigma''$. Note that if σ' above is infinite, then $\sigma = \sigma'$. We use the following definitions:

alive Process p is *alive* in τ if p does not crash in σ'.

crashed Process p is (permanently) *crashed* in τ if p crashes in σ'.

connected Processes p and q are *connected* in τ if p and q are alive in τ, p receives in σ every message that was sent from q to p in τ, and vice versa. A set of processes P is *connected* in τ, if for every two processes $p, q \in P$, p and q are connected in τ.

If σ' is infinite, p and q are (P is) *permanently connected* in σ.

detached Processes p and q are *detached* in τ if p does not receive (in σ) any message that q sends in τ and vice versa. A set of processes P is *detached* from a set of processes Q in τ if for every process $p \in P$ and every process $q \in Q$, p and q are detached in τ. Note that the definitions of detached and connected do not complement each other.

If σ' is infinite, p and q (P and Q) are *permanently detached* in σ.

connected component A set of processes P is a *connected component* in τ, if P is *connected* in τ, and P is detached from $N \setminus P$ in τ.

If σ' is infinite, P is a *permanently connected component* in σ.

A permanently connected component defines a *stable* situation in which members of the permanently connected component can exchange messages among themselves, but cannot receive any message from other members. Note that, although messages are guaranteed to be delivered within a permanently connected component, there is no bound on the latency of these messages.

2.1 Correct Processes

In the crash-failure model, correct processes are identified with the live processes. In that model, two correct processes can communicate reliably. We make an analogous tagging in our model, such that any two processes we tag as *correct* will eventually be able to exchange messages reliably. However, we differ from previous models in that we allow two correct processes to lose an unbounded (but finite) number of messages through some transient period, and to then become permanently connected. The notion of correctness is viewed by an external observer that sees the infinite run. The processes are not aware of this tagging, nor can they know that they have reached the "stable" point in the execution. The tagging is only used for reasoning about the protocol.

Formally, the correct processes are tagged as follows:

Definition 2.1 *Let P be the largest permanently connected component in an execution σ (if there are two or more such components of equal size, choose one among them). For each $p \in P$, p is called* correct *in σ. For each $q \notin P$, q is called* faulty *in σ.*

Note that the above definition holds in the case that the largest permanently connected component contains only a minority of the processes (and in the worst situation, contains an empty set, if the system never stabilizes). However, this is of limited interest in the current context, since in the context of agreement protocols, a majority resilience threshold (on the number of correct processes) is required.

3 Failure Detectors

We have already identified the correct processes with a permanently connected component in the execution, thereby extending the model to include initial omissions. Using our definition, if a majority of the processes eventually becomes permanently connected, they are considered correct. This allows us to extend the definition of failure detectors of Chandra and Toueg [4] from a crash-failure asynchronous environment to a partitionable environment, in a natural way. It is important to note that in environments in which messages never get lost, the entire set of non crashed processes forms a connected component. Thus, in such environments, our definitions are compatible with those proposed by Chandra and Toueg.

Failure detectors are characterized using *completeness* and *accuracy* properties. Below, we provide the definitions of *completeness* and *eventual accuracy*[1] in our model.

Strong Completeness Eventually every faulty process is permanently suspected by *every* correct process.

Weak Completeness Eventually every faulty process is permanently suspected by *some* correct process.

Eventual Weak Accuracy There is a time after which some correct process is never suspected by any other correct process.

	Accuracy	
Completeness	Eventual Strong	Eventual Weak
Strong	*Eventually Perfect* $\Diamond\mathcal{P}(om)$	*Eventually Strong* $\Diamond\mathcal{S}(om)$
Weak	$\Diamond\mathcal{Q}(om)$	*Eventually Weak* $\Diamond\mathcal{W}(om)$

Figure 1: Classes of Failure Detectors

We define four classes of eventual failure detectors; each failure detector is characterized by the strength of its completeness and eventual accuracy properties. The failure detector classes are shown in Figure 1.

3.1 Reducing $\Diamond\mathcal{W}(om)$ to $\Diamond\mathcal{S}(om)$

In this section, we show equivalence between the two classes of failure detectors defined above. Reduction between failure detector classes is based on the notion of implementation, as follows:

[1] Chandra and Toueg also define *perpetual accuracy*, but this concept does not carry to our model because of the transient nature of link failures.

Every process p does the following:

1. Initially:
   $S_p \leftarrow \emptyset$

2. p periodically sends its suspects list to all the processes:
   `send` (p, W_p) to all.

3. **when** `receive` (q, W_q) for some q,
   $S_p \leftarrow S_p \cup W_q \setminus \{q\}$

Figure 2: From $W \in \diamond\mathcal{W}(om)$ to $S \in \diamond\mathcal{S}(om)$

Definition 3.1 *We say that a failure detector class $\mathcal{R}$ can be* reduced *to a failure detector class $\mathcal{S}$ if, for every $R \in \mathcal{R}$, there exists a protocol that uses R (as input) and whose output is a failure detector $S \in \mathcal{S}$.*

We reduce $\diamond\mathcal{W}(om)$ to $\diamond\mathcal{S}(om)$ using the algorithm suggested by Chandra and Toueg for reducing $\diamond\mathcal{W}$ to $\diamond\mathcal{S}$ [4], described in Figure 2. We prove below that this algorithm reduces $\diamond\mathcal{W}(om)$ to $\diamond\mathcal{S}(om)$. We denote the given (input) failure detector by W, and the resulting (output) failure detector by S. We denote by W_p the suspects list of p with the failure detector W, and S_p the suspects list with S.

The algorithm works as follows: p periodically sends its input suspects list to all the processes (including itself). When p receives a suspects list from a process q, it removes q from its suspects list, and merges q's suspects list into its output suspects list.

Lemma 3.1 *If W satisfies* weak completeness *then S satisfies* strong completeness.

Proof: Let p be a correct process. If q is faulty, then eventually $q \in W_r$ for some correct r. Furthermore, since q is faulty, there is a time after which p does not hear from q. Since r periodically sends W_r to p, and since p and r are permanently connected, eventually p receives r's message and merges W_r into S_p, and thus, eventually, $q \in S_p$. Subsequently, q is never removed from S_p. ■

Lemma 3.2 *If W satisfies* eventual weak accuracy *then S also satisfies* eventual weak accuracy.

Proof: Let p be a correct process such that there is a time t_0 after which p never appears in W_r, for any correct process r. Then after this time, no correct process ever sends a suspects list containing p. Furthermore, there is a time $t_1 \geq t_0$ after which the correct processes do not receive messages sent by faulty processes and no message that is received by a correct process (and was also sent by a correct process) includes p in the suspects list. If at t_1 $p \in S_q$ for some correct q, then since p and q are connected, there exists a time $t_2 > t_1$ in which q gets p's suspects list and removes p from S_q. p is never again added to S_q. ■

From these two lemmata, the theorem immediately follows:

Theorem 3.3 *The algorithm in Figure 2 reduces $W \in \diamond\mathcal{W}(om)$ to $S \in \diamond\mathcal{S}(om)$.*

We note that the reduction from $\diamond\mathcal{W}(om)$ to $\diamond\mathcal{S}(om)$ takes constant space (w.r.t. the number of iterations) and works correctly even in the absence of FIFO links. The importance of this fact will become apparent below, when we construct a logarithmic-space solution for the Consensus problem using a failure detector in $\diamond\mathcal{S}(om)$, and thus obtain a logarithmic-space solution for Consensus using a failure detector in $\diamond\mathcal{W}(om)$.

4 Consensus with Failure Detectors

There are many (equivalent) formulations of the Consensus problem, originally introduced in [12]. We say that a protocol solves the Consensus problem if:

Agreement All the processes that decide decide on the same value.

Non-Triviality The protocol has different executions that decide on at least two different values.

Termination All the correct processes eventually decide.

Note that although the definition of Consensus in the omission failure model looks the same as in the definition for the crash failure model, it is actually harder to solve Consensus in the former than it is in the latter. This stems from the differences between the definitions of correct processes in the two models. For example, consider a failure detector that accurately detects all processes that crash, but never detects faulty processes that do not crash. Such a failure detector is strong enough to solve Consensus in the crash failure model, but not in the omission failure model.

In the rest of this section, we show protocols for solving Consensus in the omission failure model with a majority of correct processes, using an eventually strong failure detector $S \in \diamond\mathcal{S}(om)$. However, since we have shown how to implement $\diamond\mathcal{S}(om)$ from $\diamond\mathcal{W}(om)$, then the protocols we describe work with $\diamond\mathcal{W}(om)$ by superimposing the reduction from $\diamond\mathcal{W}(om)$ to $\diamond\mathcal{S}(om)$ onto them.

4.1 Consensus with $\diamond\mathcal{S}(om)$ – Naive Solution

Chandra and Toueg [4] suggested an algorithm that solves Consensus with a failure detector $S \in \diamond\mathcal{S}$ if there are no more than $\lfloor\frac{n}{2}\rfloor$ crash-failures. If there are more than $\lfloor\frac{n}{2}\rfloor$ failures, the algorithm blocks, but in no case does it allow two processes to decide on different values. Their algorithm was not designed to overcome messages losses, and consequently, if more than $\lfloor\frac{n}{2}\rfloor$ of the processes incur send or receive omission faults, the algorithm may block.

The algorithm can be changed to overcome message losses as follows:

1. Each process records all the messages it sent in the past and piggybacks on every message all the previous messages.
2. Every undecided process periodically re-sends the latest message to every other process, at every stage of the protocol. In this way, processes retransmit past messages even when they are otherwise idle.
3. A process that receives a message that it has already received, or a message from a previous round, discards this message.
4. When a process *decides* it stops periodic sending of messages. When a decided process receives a message other than *decide*, it responds by sending a *decide* message with its decision value. This way, undecided members that reconnect with decided processes receive the *decide* message, and the protocol stops sending messages once all the correct processes have decided.

Note that the memory requirements of the suggested protocol grow **linearly** in the **total size** of exchanged data, and if the exchanged data is constant, the memory requirement is $O(t \log t)$ where t is the number of rounds. Therefore, this protocol is not reasonable for use in practice.

Theorem 4.1 *The protocol given in [4] with the modifications above solves Consensus in the omission failure environment extended with a failure detector $S \in \diamond\mathcal{S}(om)$, given that a majority of the processes are correct.*

Proof: (Sketch) The protocol in [4] uses a majority to guarantee against inconsistent decisions. Therefore, it is only left to show that the protocol with the above modifications guarantees termination. Assume to the contrary that there is an infinite run of the protocol above.

With the above modifications to the protocol, for any pair (p, q) of correct processes, q eventually receives all of the messages p sends it. Furthermore, p cannot distinguish the case that it is permanently detached from q, from the case that q has crashed. Therefore, to the correct processes in the crash failure model the execution seems the same as to a group of $\lceil\frac{n+1}{2}\rceil$

correct processes in the omission failure model. Therefore, we can construct an infinite run in the crash failure model by stopping all the incorrect processes at the point of detachments. The runs are indistinguishable to the correct processes, and therefore it does not terminate in the crash failure model as well. This contradicts the results of [4]. ■

4.2 Consensus with $\diamond\mathcal{S}(om)$ – Practical Solution

In this section, we offer an alternative Consensus algorithm to the one above, but whose memory requirement is bounded logarithmically in the number of messages sent, and does not require storing nor re-sending of old messages. The algorithm uses a failure detector $S \in \diamond\mathcal{S}(om)$.

The algorithm uses a *rotating coordinator* paradigm. The coordinator of round r is process $(r \bmod n) + 1$, while the other processes are slaves. Every coordinator tries to determine a consistent decision value.

Process p maintains the following variables: the round number, r_p, the coordinator of this round, C_p, the current estimate, est_p, and a timestamp, ts_p, which contains the number of the round when this estimate was last changed. Initially, the estimate contains the process' initial value, and $ts_p = 0$.

The algorithm is easily described as a state machine with the following states: an **Init** state, two coordinator states, **C1** and **C2**, two slave states, **S1** and **S2**, and a final **Decide** state. Periodic sending is modeled as a response to a null event.

The state diagram of round r_p of the algorithm at process p is depicted in Figure 3. Circles represent states, arrows represent state transitions. Each circle is labeled with the name of the state, and with the type of message that p periodically sends while it is in this state. Each arrow is labeled with the event that caused the state transition. The event $r_q > r_p$ is a shorthand for: p receives a message from round r_q where $r_q > r_p$. The event **suspect** q denotes that q is in the suspects list.

The state chart of the protocol is described in Figure 4. Each round of the algorithm is conducted in phases. First, the coordinator sends a *new* message informing all the processes that a new round has started, and the slaves send timestamped estimates to the coordinator. The coordinator gathers *estimates* from $\lceil\frac{n+1}{2}\rceil$ processes (including its own estimate which it initially holds), adopts one of the estimates with the latest timestamp, sets its ts to the current round number, and multicasts the chosen estimate to all the slaves.

When a slave receives the new estimate it adopts it as its new estimate, sets its ts to the current round number, and sends an ACK to the coordinator. Once a majority (including the coordinator) adopts this estimate, this value becomes fixed, and it is the only possible decision value in the system.

When the coordinator gathers $\lceil\frac{n+1}{2}\rceil$ ACKs, (counting an ACK from itself), it sends a *decide* message. Every process that receives a *decide* message, decides on the suggested value.

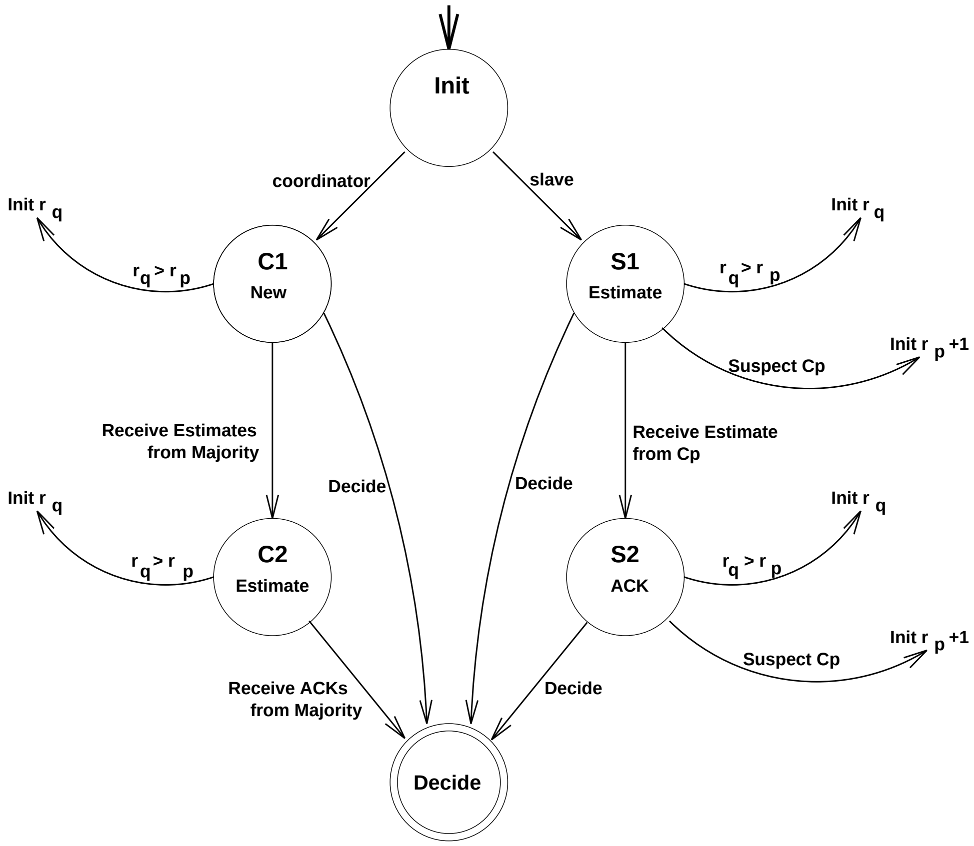

Figure 3: State Diagram for Round r_p of the Protocol

The following two rules prevent the processes from blocking in case of failures:

1. If a slave **suspects** the current coordinator, it aborts the current round and starts a new round of the protocol.

2. When a process receives a message with a higher round number than its own, it aborts the current round and joins the new round.

With these rules, if the coordinator crashes or detaches, a new round is started, with a different coordinator. If the coordinator, c, is suspected by some slave p, then p "crowns" a new coordinator, q. Once q receives the estimate from p it realizes that it is the new coordinator, and multicasts a *new* message to all. When c receives this message, it joins the round lead by q. This may be repeated several times, but with a failure detector $S \in \diamond\mathcal{S}(om)$, eventually a correct process u, that is not suspected by any other correct process becomes coordinator. The round lead by u is not aborted by any correct member, and eventually leads to a decision. Thus, the algorithm solves Consensus using $S \in \diamond\mathcal{S}(om)$.

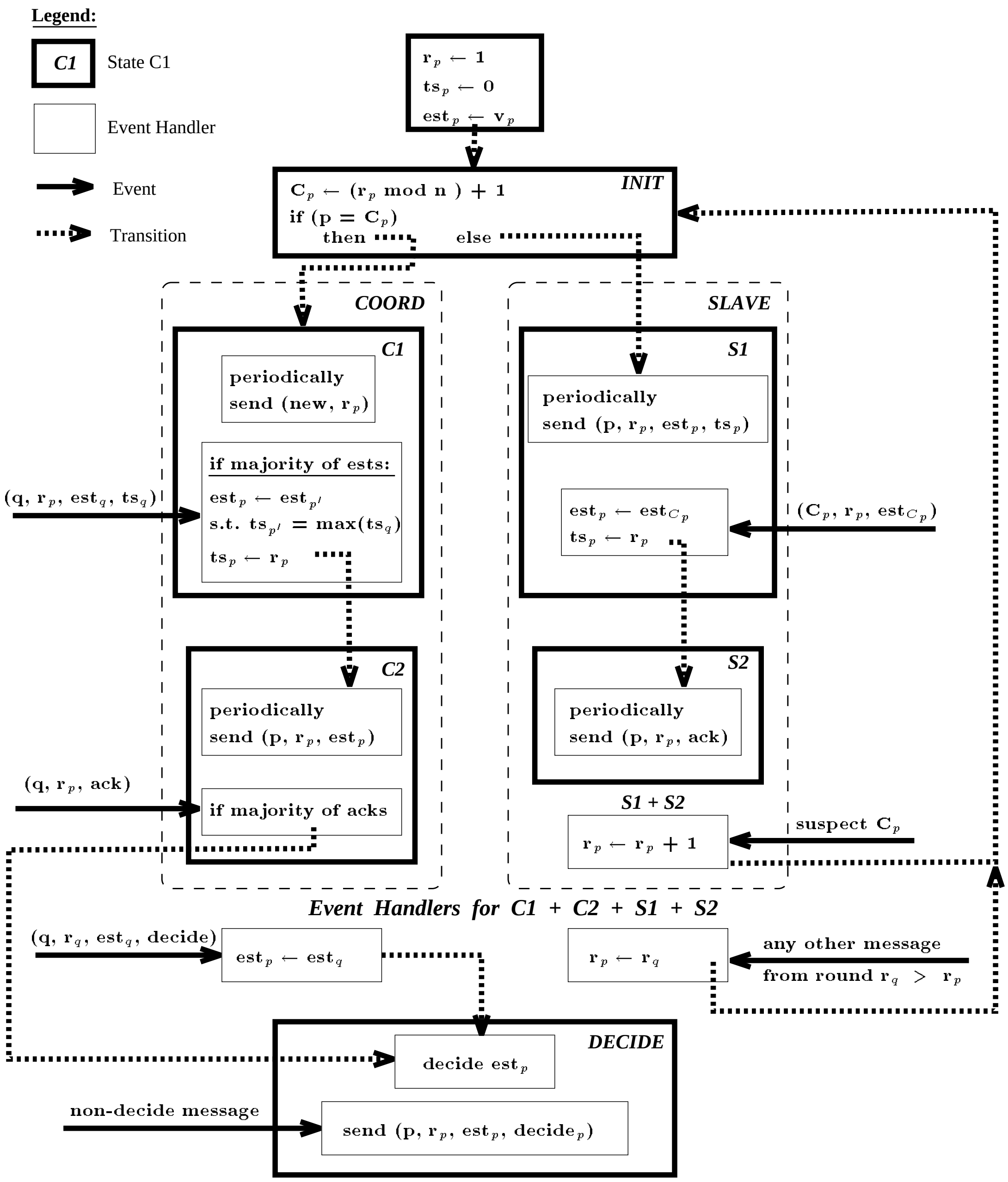

Figure 4: Deciding with Bounded Buffers and $S \in \diamond\mathcal{S}(om)$
State Chart

To allow for recovery from message omissions, each process periodically re-sends its last message, which contains the round number of the current round. Thus, even if there are message losses that cause different processes to be "stuck" in different rounds, when the system stabilizes and a group of processes becomes permanently connected, they all shift to the latest round. Note that due to Strong Completeness , no process may be "stuck" as a slave of a disconnected coordinator.

A process stops periodic sending of its latest message when it *decides*. When a decided process receives a message other than *decide*, it responds by sending a *decide* message with its decision value. This way, undecided members that reconnect with decided processes receive the *decide* message, and the protocol stops sending messages once all the live and connected processes have decided.

The number of bits required by this protocol increases logarithmically with the number of rounds, as the counters used by it increase. In practice, this is not a real problem, since some number between 64 bits to 128 bits would be sufficient for any practical application.

4.3 Proof of Correctness

We now prove that the protocol above solves Consensus.

Lemma 4.2 *(Agreement) Let r be the first round in which a coordinator collects $\lceil\frac{n+1}{2}\rceil$ ACKs. Assume that in round r, the coordinator chooses estimate v. Then for any round $r' > r$ in which a coordinator collects $\lceil\frac{n+1}{2}\rceil$ estimates, the coordinator picks v as its new estimate (in state $C1$).*

Proof: First, note that in any round in which the coordinator ejects before collecting $\lceil\frac{n+1}{2}\rceil$ estimates, no process changes its estimate. Also, since the coordinator of round r collects $\lceil\frac{n+1}{2}\rceil$ ACKs, there are at least $\lceil\frac{n+1}{2}\rceil$ processes in round r that have v as their estimate with timestamp at least r. Let $r' > r$ be the first round in which the coordinator chooses its estimate to some value other than v (denoted $\bar{v}$). This means that at least one process proposed $\bar{v}$ with a timestamp greater than or equal r. However, a process can change its timestamp only in rounds in which the coordinator collects $\lceil\frac{n+1}{2}\rceil$ estimates, and in these cases the process changes its estimate to whatever the coordinator has chosen. Therefore, there is a round between r and r' in which a coordinator chooses $\bar{v}$, contradicting the minimality of r'. Thus, there does not exist such r', and therefore in any round higher than r, in which the coordinator changes its estimate, it does so to value v. ■

Lemma 4.3 *(Termination) In every execution of the protocol described in this section in which $\lceil\frac{n+1}{2}\rceil$ processes or more are correct, every correct member decides.*

Proof: Intuitively, we need to prove that the algorithm does not run into a deadlock, i.e., a situation where all the correct processes wait for each other somehow, nor does it run into a livelock, i.e., a race where the current round number increases endlessly without decision.

Consider a time after which a majority of correct processes become permanently connected, no messages from faulty processes are delivered to any correct process, and some correct process p is never suspected by any correct process from this point on. If some correct process p has decided already, and another correct process q has not decided yet, then p will eventually receive a message from q and will respond by re-sending a decide message which will be received by q, hence q will decide.

Otherwise, let r be the highest round number held by any correct process at this point, and let q be the coordinator corresponding to round r. Since all the correct processes belong to a permanently connected component, eventually they will receive a message from round r or higher, and move to that round. If q is faulty, then eventually all of the correct processes will suspect it and move to round $r+1$ or higher. If q is correct and does not decide in round r, then this could only happen if some correct process suspects q, moves to round $r+1$, and causes enough processes to move to round $r+1$ before they send ACKs to q. Again, this will eventually cause all the correct processes to move to round $r+1$ or higher. This can be repeated however only until some round r' is reached in which the coordinator is p.

Once some correct process reaches round r', it will cause all of the correct processes to receive a message from round r', and since none of them suspects p, none will eject from round r'. Thus, it is guaranteed that p will eventually receive at least $\lceil \frac{n+1}{2} \rceil$ estimates in $C1$ and $\lceil \frac{n+1}{2} \rceil$ ACKs in $C2$, and therefore will eventually decide. ∎

Lemma 4.4 *(Non-Triviality) In every execution of the protocol in which some process decides, it decides on a value that was the initial value of some process.*

Proof: By the code of the protocol, a process can only decide on a value that was the estimate of some process in the round in which the decision was taken. In every round but the first one, the estimated value of every process is an estimated value of some process in a previous round. In the first round, the estimated value of each process is its initial value. ∎

The following theorem follows immediately from Lemmas 4.2, 4.3, and 4.4.

Theorem 4.5 *The protocol described in this section solves the Consensus problem in an omission failure environment extended with a failure detector $S \in \Diamond\mathcal{S}(om)$, provided that a majority of the processes are correct.*

5 The Weakest Failure Detector for Solving Consensus

We now show that $\Diamond\mathcal{W}(om)$ is the weakest failure detector for solving Consensus in an asynchronous model with message omissions with a majority of correct processes.

Theorem 5.1 *$\Diamond\mathcal{W}(om)$ is the weakest failure detector allowing Consensus to be solved in an asynchronous model with message omissions with up to $\lfloor \frac{n}{2} \rfloor$ faulty process.*

Proof: (Sketch) In [3], Chandra, Hadzilacos and Toueg prove that $\diamond\mathcal{W}$ is the weakest failure detector for solving Consensus in an asynchronous environment with crash failures. Their proof reduces a failure detector $\mathcal{D}$ that solves Consensus to $\diamond\mathcal{W}$; the reduction sends the entire history of each process with each message, and therefore works even if message omissions occur. Therefore, the same reduction may be applied to reduce any failure detector $\mathcal{D}(om)$ that solves Consensus in our model, reducing it to $\diamond\mathcal{W}(om)$. ■

Note that the notion of comparing between failure detectors using the *implements* relation, as defined by Chandra et al. [3], does not take into account the space-complexity of the reduction. Therefore, a failure detector may exist, that requires a linear space (or more) implementation to implement $\diamond\mathcal{W}(om)$, but is nevertheless considered no weaker than $\diamond\mathcal{W}(om)$. An interesting open direction is to consider an alternative definition of the *implements* relation, by constraining the space complexity of the implementation. It is an open question whether there exists a failure detector weaker than, in this sense, $\diamond\mathcal{W}(om)$, that can solve Consensus in our environment.

6 Conclusions

In this paper, we have shown how to analyze the weakest failure detector in an environment where message reliability matches the properties of real networks, and have shown that in such a setting, the intuitive findings of Chandra and Toueg remain valid. In particular, we have defined $\diamond\mathcal{W}(om)$ and $\diamond\mathcal{S}(om)$ analogous to $\diamond\mathcal{W}$ and $\diamond\mathcal{S}$ and showed that $\diamond\mathcal{W}(om)$ and $\diamond\mathcal{S}(om)$ are equivalent in the omission failure model too.

We have presented protocols that solve Consensus using $\diamond\mathcal{S}(om)$ in executions in which there are at least $\lceil\frac{n+1}{2}\rceil$ correct processes. We suggested a practical algorithm that can be used, as a building block, to make various distributed services more fault tolerant. For example, the primary component membership protocol in [11], and the atomic commitment protocol in [7] use Chandra and Toueg's Consensus protocol as a building block. Replacing that module with our Consensus protocol will make these services tolerant to omission faults.

Acknowledgments

We are grateful to Sam Toueg and Vassos Hadzilacos for pointing out weaknesses in our previous version. Special thanks to Gregory Chockler and Esti Yeger Lotem for their help in making the presentation of the algorithm clearer. Thanks to Ken Birman for many helpful discussions on the area.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, W. Dai-Wei, and L. Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, November 1994.

[2] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, November 1995.

[3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*. To appear.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[6] F. Cristian and F. Schmuck. Agreeing on process group membership in asynchronous distributed systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, 1995.

[7] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *IEEE International Symp. on Parallel and Distributed Processing (SPDP)*, October 1995.

[8] R. Guerraoui and A. Schiper. "γ-accurate" failure detectors. In *International Workshop on Distributed Algorithms (WDAG)*, 1996. To appear.

[9] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *ACM Symp. on Prin. of Database Systems (PODS)*, pages 245–254, May 1995.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[11] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *Proceedings 2nd Open Workshop of the ESPRIT project BROADCAST (number 6360)*, July 1995.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[13] W. Vogels. World wide failures. In *ACM SIGOPS European Workshop*, 1996. To appear.