On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems

Marcos K. Aguilera[1], Gérard Le Lann[2], and Sam Toueg[3]

[1] HP Systems Research Center, 1501 Page Mill Road, Palo Alto, CA, 94304, USA,
aguilera@hpl.hp.com
[2] INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France,
Gerard.Le_Lann@inria.fr
[3] Department of Computer Science, University of Toronto, Toronto, Canada,
sam@cs.toronto.edu

Abstract. We investigate whether fast failure detectors can be useful — and if so by how much — in the design of real-time fault-tolerant systems. Specifically, we show how fast failure detectors can speed up consensus and fault-tolerant broadcasts, by providing fast algorithms and deriving some matching lower bounds, for synchronous systems with crashes. These results show that a fast failure detector service (implemented using specialized hardware or expedited message delivery) can be an important tool in the design of real-time mission-critical systems.

1 Introduction

Consensus and various types of fault-tolerant broadcast — such as atomic and reliable broadcast — are central paradigms for fault-tolerant distributed computing. Unfortunately, the implementation of these paradigms can be expensive, particularly for real-time systems which are concerned about worst-case time behavior. For instance, consensus requires in the worst-case time $D(1 + f_{max})$ in a synchronous system, where D is the *maximum* message delay and f_{max} is the maximum number of process that may crash. This worst-case time complexity, which also applies to several types of fault-tolerant broadcasts, hinders the widespread use of these paradigms in many applications. This is especially problematic in real-time applications, which are particularly concerned about worst-case scenarios.

In this paper we explore a new approach, namely, the use of fast failure detection, to circumvent this lower bound and obtain faster algorithms for consensus and fault-tolerant broadcasts. Our results show that this approach is particularly suitable to the design of real-time mission-critical systems, where worst-case performance is crucial and where we can use special mechanisms to implement fast failure detection.

There are several ways one can achieve fast failure detection. One way is to use *specialized hardware*. For example, in some mission-critical systems, such as spaceborne ones [19], and in some Tandem systems, failure detection has been implemented in hardware and is very fast.

Another way to achieve fast failure detection is to use expedited message delivery. In fact, researchers in the real-time community have long considered ways to expedite the

D. Malkhi (Ed.): DISC 2002, LNCS 2508, pp. 354–369, 2002.

delivery of selected messages that are critical to the performance of applications, e.g., control or clock synchronization messages. One common way to do so is to tag the urgent messages so that they can be processed ahead of the others in the network and in the waiting queues of the sender and receiver (e.g., [16,17]). Another way to expedite urgent messages is to use a *physically separate medium* to carry them, as in [5] or in control networks of process control and transportation systems, or in signalling channels of telecommunications systems. Whatever mechanism is used, these approaches boil down to having two types of messaging services: a regular one for most application-level messages, and an expedited service for a small set of urgent, time-critical messages. In some of these systems, there can be a large difference between the *maximum* message delays of regular and urgent messages; in fact, they can be orders of magnitude apart. For example, in some local area networks, the highest priority messages are 8 to 30 times faster than the lowest priority ones, assuming only one stage of waiting queues [21]. Recent work [13] has shown how to use expedited message delivery to build a fast failure detector with an analytically-quantified maximum detection time d that is much smaller than the maximum delay D of regular messages.[1]

In this paper, we investigate whether fast failure detectors can speed up consensus and/or various types of fault-tolerant broadcasts, and if so, by how much. It is not entirely obvious that one can take advantage of a fast failure detector to speed up some or all of these problems. For example, a natural attempt is to use the fast failure detector to simulate synchronous rounds, and the hope is that the failure detector will shorten some of these rounds, namely, those that have failures. This in turn would speed up existing round-based algorithms in the worst-case runs, which are exactly those that have failures. But this naive idea does not work: every round could have one correct sender whose message to every process takes the maximum delay D to arrive; thus every round could take D.

We show that fast failure detectors can indeed be used to significantly speed up consensus and fault-tolerant broadcasts, and we also provide some tight lower bounds. Specifically, our results are as follows. We consider a synchronous system with process crashes, where the maximum message delay is some known constant D, and processes have access to a failure detector that detects crashes within some known constant d, where $d \leq D$. We first give an algorithm for consensus that reaches decision in time $D + f_{max}d$, where f_{max} is the maximum number of processes that may crash.[2] We then give an *early-deciding* [9] algorithm for consensus that reaches decision within time $D + fd$, where f is the number of crashes that actually occur (in many runs $f < f_{max}$). Note that in failure-free runs, decision occurs within time D — the fastest possible. In addition, our consensus algorithms are message efficient: they send at most $(f + 1)n$ point-to-point messages.

The consensus algorithms in this paper are time optimal: we prove that in a synchronous system with fast failure detection, consensus requires at least time $D + fd$.

[1] Here, we stress that D is the *maximum* and not the *average* message delay. In many systems, the maximum message delay of regular messages is *orders of magnitude* greater than their average delay.

[2] Actually, this algorithm solves *uniform* consensus. In fact all algorithms in this paper solve the uniform version of the problem in question.

This proof is novel in two respects: it is the first lower bound proof for synchronous systems equipped with failure detection, and moreover it uses a new technique to deal with continuous-time synchronous systems (as opposed to round-based ones), which we believe to be applicable in other contexts.

We then consider several forms of fault-tolerant broadcasts [12], and for each of them we give *early-delivering* algorithms, i.e., algorithms where message delivery time is proportional to f not f_{max}. Specifically, we first give an algorithm for terminating reliable broadcast that delivers messages within time $D + fd$. We next present a reliable broadcast algorithm that delivers within time $2D + (f - 1)d$. Finally, we describe an atomic broadcast algorithm that delivers within time $2D + (f - 1)d$. All these broadcast algorithms deliver within time D in the failure-free case — this is the best possible.[3] Moreover, the algorithms are message efficient: they use at most $(f + 2)n$ point-to-point messages. Note it is surprising that we can solve atomic broadcast in time $2D + (f - 1)d$. For instance, once the diffusion of a broadcast message has started, it is not clear that a process should be allowed to stop it, lest the message be delivered by some but not all correct processes. But continuing the diffusion could lead to a message relay chain of size f, and hence the broadcast time would be $(f + 1)D > 2D + (f - 1)d$.

All the results above assume that messages are not lost, but we show how to extend them to allow the following type of message losses. In most systems, a process sends a message by placing it in an outgoing waiting queue before it can be transmitted. If the machine hosting the process crashes, the waiting queue is wiped out and the message is lost. We model such losses by defining a parameter δ such that, if the sender crashes within δ time of sending a message, then this message may be lost (δ can be interpreted as the maximum time a message may spend in the outgoing queue; note that if $\delta = 0$ we fall back to the case without message losses). We then describe an early-deciding consensus algorithm that takes at most $D + f(d + \delta)$ time to decide. We next show that consensus requires at least time $D + f(d + \delta)$ — and thus our algorithm is time optimal. Finally, we give message-efficient early-delivering algorithms for the fault-tolerant broadcasts described above.

In summary, the contributions of this paper are the following:

- This is the first paper to study the impact of fast failure detectors on real-time fault-tolerant synchronous systems. In such systems, we focus on solving consensus and various types of fault-tolerant broadcasts.
- We give fast and simple algorithms for consensus, terminating reliable broadcast, reliable broadcast and atomic broadcast. Our algorithms are early-deciding or early-delivering. Their time complexity is $O(D + fd)$ and their message complexity is $O(fn)$ where the constants are all very small (1 or 2).
- We show that our consensus algorithms are time optimal, by proving a lower bound of time $D + fd$. (This bound carries over to all our broadcast problems.) The proof uses new techniques to handle failure detection in synchronous systems and to deal with continuous time.
- We extend our results to tolerate message losses that occur when the sender crashes within a certain period after sending.

[3] To achieve delivery in time D, two of our algorithms require some extra optimizations, as we describe in the paper.

We believe that the above results provide guidance on the design of future real-time fault-tolerant systems by showing that implementing a fast failure detection mechanism is indeed useful, and by quantifying the speed-up that can be obtained.

A remark is now in order. As we noted earlier, fast failure detection can be achieved by using specialized hardware or by expediting selected messages. In the latter case, one may wonder why not expedite directly the consensus or broadcast messages, instead of the failure detector's. The reason is that this method would not scale: the number of selected messages that can be expedited is limited, and as we increase the number of concurrent consensus and/or broadcasts this limit can be exceeded. In contrast, a failure detector can be implemented as a service that is *shared* among concurrent applications running on the same set of machines, and one can ensure that the number of messages that the failure detector sends is dependent on the number of machines and is relatively independent of the number of concurrent applications (this assumes that each failure detector message can hold a large number of process id's). Such a shared failure detector service has recently been designed and built [8,15].

Related work. Failure detectors have been used to solve a variety of problems in different environments (e.g., [6,4,11,1,14,2,8]). The idea to use priorities or deadlines to tag messages and to process them faster in the waiting queues has long been explored in scheduling theory and queueing theory (e.g., [10,22]). This idea is used in [13] to build a fault-tolerant distributed system for real-time applications. This is also the first paper to speed up consensus using expedited delivery; however, the work is for asynchronous systems and it assumes a failure detector that has been tailored to carry some consensus messages. [5] uses a separate network to ensure timely delivery of the messages in their timely computing base. One could use such a separate network for the failure detector messages.

Roadmap. The paper is organized as follows. In Section 2 we give our model with fast failure detectors. We consider the problem of consensus in Section 3: in Section 3.1 we present our basic consensus algorithm, and in Section 3.2 we present our early-deciding one. We show optimality of our consensus algorithms in Section 3.3. We then turn our attention to broadcast problems: first to terminating reliable broadcast in Section 4, then to spontaneous reliable broadcast in Section 5, and finally to atomic broadcast in Section 6. In Section 7 we consider message losses arising from process crashes: we first explain how to modify our algorithms to tolerate such losses in Section 7.1; we then show optimality of the modified consensus algorithm in Section 7.2. Due to space limitations all proofs are omitted, but they can be found in the full version of the paper.

2 Model

We consider a distributed system with n processes that can communicate with each other by sending messages over a fully-connected network. The system is synchronous in that there is a bound D on the time it takes to receive and process a message. Processes have access to a clock, which we assume to be perfectly synchronized for ease of presentation, but our results can be easily extended to clocks that are only nearly synchronized.

Processes can fail by crashing permanently, and there is an upper bound f_{max} on the number of processes that can crash. We let $f \leq f_{max}$ be the actual number of failures. Processes have access to a perfect failure detector that reports failed processes. The

detection time, i.e. the time to detect a crash [7], is bounded by some constant d. In useful realizations of our model (e.g., [13]), $d \leq D$ and in fact we make that assumption throughout the paper. In addition, for simplicity of presentation we assume that D is a multiple of d, and if it is not, we can simply increase D to $d\lceil D/d \rceil$, the next multiple of D.[4]

Links do not create or duplicate messages. Moreover, we assume there are no message losses in the first part of the paper, but we later extend our results to allow losses due to the sender's crash.

We now provide a more detailed description.

2.1 Processes

The system consists of a known totally-ordered set $\Pi = \{1, \ldots, n\}$ of processes. The computation proceeds by steps, and a step consists of several stages: (1) the process may first send a message to a subset of the processes, (2) the process then attempts to receive messages addressed to it, (3) the process may then query its failure detector, and (4) the process may change its state according to the messages it received and the information it got from the failure detector. We assume that steps are executed instantaneously. Up to f_{max} processes may fail by crashing; processes that do not crash are called *correct*. When a process crashes during a step, it may stop executing at any of the stages above. In particular, it may crash while attempting to send a message to some subset of processes. If that happens, the send may be partially successful in that it succeeds to send to only some of the targeted processes. This behavior is explained in detail in the next section.

2.2 Links

There is a link between every pair of processes, and messages sent through links are unique (uniqueness can be obtained through sequence numbers). Links guarantee the following properties:

- *(No Creation or Duplication)* A message m can be received at most once and only if it has been previously sent.
- *(D-Timeliness)* A message m sent at time t is not received after time $t + D$.

In the first part of the paper we assume that messages are not lost, that is:

- *(No Loss)* If p sends a message m to q at time t and p does not crash at time t then q eventually receives m from p.[5]

We later extend our results to a more general model that allows message losses if the sender crashes within a certain time after sending the message.

Note that D-Timeliness together with No Loss imply that if p sends m to q at time t and does not crash at time t then q receives m by time $t + D$. Moreover, if p crashes at time t then the messages that it sent may or may not be received. In particular, if p sent to both q and q' then it is possible that only one of them receives the message. To simplify presentation we assume that D is a multiple of d, but we remove this assumption in the full version of the paper.

[4] Doing so slightly increases the time complexities of our algorithms, but the complexity can be reduced as we explain in the full version of the paper.

[5] By convention, we assume that processes receive messages even if they are crashed. Of course, crashed process cannot do anything with these messages.

2.3 Failure Detectors

Processes may have access to a perfect failure detector that provides a list of processes deemed to have crashed [6]. If a process q belongs to the list of process p we say that p suspects q. The failure detector guarantees the following properties:

- *(Accuracy)* A process suspects a process q only if q has previously crashed.
- *(d-Timely Completeness)* If a process q crashes at time t then, by time $t + d$, every alive process permanently suspects q.

Note that if a process p crashes between times t and $t + d$ then some, but not necessarily all, processes may suspect p at time $t + d$.

3 Consensus

In the consensus problem, each process initially proposes a value, and processes must reach a unanimous decision on one of the proposed values. The following properties must be satisfied:

- *(Uniform Validity)* If a process decides v then some process previously proposed v.
- *(Agreement)* Correct processes do not decide different values.
- *(Termination)* Every correct process eventually decides.

Note that consensus allows processes that later crash to decide differently from correct processes. A stronger variant, called *uniform* consensus [20], disallows that by requiring a stronger property than Agreement:

- *(Uniform Agreement)* Processes do not decide different values.

To strengthen our results, we use consensus for our lower bounds and provide algorithms that solve uniform consensus.

A consensus algorithm is said to *reach decision* or *decide* when all alive processes have decided. It turns out that no consensus algorithm can always reach decision within time less than $(1 + f_{max})D$ in a synchronous system without failure detection. However, with a fast failure detector it is possible to do better, as shown in the next section.

3.1 Uniform Consensus Using Fast Failure Detection

Figure 1 shows a simple uniform consensus algorithm that terminates within time $f_{max}d + D$. Each process i keeps a variable e_i with its current estimate of the consensual decision value. Its initial value is the value that the process wishes to propose (line 2). In this algorithm, this variable never changes. We divide real time in consecutive rounds of duration d each, so that each round i corresponds to the time interval $[(i-1)d, id)$. Note that these "mini" rounds are *not* the same as the ones in a synchronous round-based system: here, if $D > d$ then messages sent in a round could be received in a higher round.

At the beginning of round i, process i checks if it suspects all processes with a smaller process id and, if so, it broadcasts (e_i, i) to all (line 4).[6] Then, at time $f_{max}d + D$, all processes decide on the estimate received from the largest process id (lines 5–7). It turns out that lines 1–4 need only be executed by processes $1, 2, \ldots, f_{max} + 1$; the other processes may simply wait and decide at time $f_{max}d + D$ in lines 5–7.

[6] For $i = 1$ note that process 1 vacuously suspects all processes with smaller id, because there are none. Thus, process 1 will send $(e_1, 1)$ to all if it does not crash.

```
   Code for each process i:
1    Initialization:
2      e_i ← value that process i wishes to propose
3    at time (i − 1)d do
4      if suspect j for all 1 ≤ j ≤ i − 1 then send (e_i, i) to all
5    at time f_max d + D do
6      max ← largest j such that received (e_j, j) from j
7      decide e_max
```

Fig. 1. Optimal uniform consensus algorithm with time complexity $f_{max}d + D$.

Theorem 1. *In a synchronous system with fast failure detection, uniform consensus can be solved with an algorithm that decides in at most time $D + f_{max}d$ using $(f_{max} + 1)n$ messages.*

3.2 Early-Deciding Uniform Consensus

The consensus algorithm in Figure 1 always decides at time $D + f_{max}d$, where f_{max} is the maximum number of process crashes. In practice, most of the time the number f of failures that actually occur is much smaller than f_{max}, and we would like algorithms to decide faster in these common cases. An algorithm is said to be *early-deciding* if its decision time is proportional to f, not f_{max}. We now present such an algorithm that decides within time $D + fd$. Since $f \leq f_{max}$, this algorithm always performs better than our previous one and, with few failures (small f), it performs much better. This early-deciding algorithm, which is shown in Figure 2, assumes that D is an integral multiple of d (we handle the general case in the full version of the paper).

Like in our previous algorithm, at each round i, process i sends its current estimate if it suspects all processes with a smaller id. However, now processes may change their estimate e_i when they receive the estimate of another process. More precisely, processes keep a variable max_i with the id of the largest process from which it has received an estimate (initially it is zero). When a process receives an estimate from a process whose id is larger than max_i it changes its estimate and updates max_i (line 4). At times $(j-1)d + D$ for $j = 1, \ldots, n$, processes check if they trust process j; if so, they decide on their current estimate (line 9).[7]

Theorem 2. *In a synchronous system with fast failure detection, uniform consensus can be solved with an algorithm that decides in at most time $D + fd$ using $(f+1)n$ messages.*

Note that in the failure-free case, delivery occurs within time D using only n messages.

An important remark is in order. A consensus box assumes that all processes have a priori knowledge that they wish to reach consensus, and it is not the role of consensus to convey that knowledge. Thus, to execute consensus, correct processes are expected to propose a value initially (or at some known common fixed time) and all properties

[7] At times $(j-1)d$ or $(i-1)d + D$, if the process receives a message, we assume it executes line 4 before lines 6 or 8.

```
Code for each process i:
1   Initialization:
2     e_i ← value that process i wishes to propose
3     max_i ← 0
4   upon receive (e_j, j) with j > max_i do
5         max_i ← j; e_i := e_j
6   at time (i − 1)d do
7     if suspect j for all 1 ≤ j ≤ i − 1 then send (e_i, i) to all
8   at time (j − 1)d + D for every 1 ≤ j ≤ n do
9     if trust j and not yet decided then decide e_i
```

Fig. 2. Optimal early-deciding uniform consensus algorithm with time complexity $D + fd$.

of consensus are contingent on that. It turns out, however, that even if some correct processes do not propose, the consensus algorithms of Figures 1 and 2 always guarantee that if some process decides a value, this value is one of the proposed values (Uniform Validity). This feature of our algorithms will be used in Section 5. However, it is possible for processes to decide differently when some of the correct processes do not propose. We do not believe this will not be a problem for most applications, but when it is, one could use the atomic broadcast algorithm of Section 6 to solve consensus in the obvious way: to propose a value, a process atomically broadcasts it and then processes decide on the first atomically delivered value. By doing so, we get a consensus algorithm that always satisfies Uniform Validity and Uniform Agreement, even if some correct processes do not propose.

3.3 Time Optimality

Our consensus algorithms are time optimal:

Theorem 3. *In a synchronous system with fast failure detection, every consensus algorithm has a run in which decision takes time at least $D + f_{max}d$.*

This theorem is a special case of a more general one stated in Section 7.2.

Observation 4 *No early-deciding algorithm can always ensure decision in less time than $D + fd$.*

This observation follows immediately from Theorem 3 (because if only f processes fail then we can take $f_{max} = f$). Thus, our early-deciding algorithm is also optimal.

4 Terminating Reliable Broadcast

In the terminating reliable broadcast problem, a designated process called the sender s wishes to broadcast a message. Processes have a priori knowledge of who the sender is and when it intends to broadcast (but they do not know what its message is). The sender, however, may crash and fail before or during the broadcast. The goal is for all alive

processes to either deliver the sender's message or to unanimously agree that the sender has crashed by delivering a special "sender faulty" message. More precisely, terminating reliable broadcast guarantees [12]:[8]

- *Validity:* If s is correct then it eventually delivers the message that it broadcasts.
- *Uniform Agreement:* If any process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity:* A process delivers a message m at most once[9] and if $m \neq$ "sender faulty" then m is the message of the sender.
- *Termination:* Every correct process eventually delivers a message.

To solve terminating reliable broadcast, note that the consensus algorithm in Figure 2 always decides on the value of the first process if this process is correct. Thus, we let the sender be that first process. So in line 2 the sender proposes its message, while other processes propose "sender faulty". By doing so, we get an early-delivering algorithm for terminating reliable broadcast that delivers within time $D + fd$. Its message complexity is $(f + 1)n$.

Theorem 5. *In a synchronous system with fast failure detection, terminating reliable broadcast can be solved with an algorithm that delivers in at most time $D + fd$ using $(f + 1)n$ messages.*

Note that in the failure-free case ($f = 0$), delivery occurs at time D using only n messages.

5 Reliable Broadcast

In terminating reliable broadcast (Section 4), all processes have a priori knowledge that there is a sender that wishes to broadcast a message at some known time. This is not the case with reliable broadcast: any process can broadcast at any time, and that time is unknown to other processes. Reliable broadcast guarantees the following properties [12]:

- *(Validity)* If some correct process p broadcasts a message m then p eventually delivers m.
- *(Uniform Agreement)* If any process q delivers a message m then eventually all correct processes deliver m.
- *(Uniform Integrity)* A process delivers a message m at most once and only if it has been previously broadcast.

We focus on *timely* reliable broadcast, which also satisfies the following:

- *(Δ-Timeliness)* If some process p broadcasts a message m at time t then no process can deliver m after time $t + \Delta$.

[8] This is actually *uniform* terminating reliable broadcast, and in fact all broadcasts we consider in this paper are uniform. For brevity we omit the word "uniform" from our broadcasts.

[9] Throughout this paper we assume that broadcast messages are unique (e.g., they contain a sequence number).

```
Code for each process p:
1   To broadcast a message m at time t:
2     send (m, p, t) to all
3   upon receive (m, q, t) do
4     schedule at time t + D:
5       if trusted q at time t + d then v ← m else v ← ⊥
6       propose_{q,t}(v)
7   upon decide_{q,t}(m) do if m ≠ ⊥ then deliver m
```

Fig. 3. Reliable broadcast algorithm.

Here Δ is a known value that specifies how fast processes must deliver messages: together with Uniform Agreement and Validity, Δ-Timeliness implies that if a correct process p broadcasts m then all correct processes deliver m within time Δ.

In Figure 3, we give a timely reliable broadcast algorithm that uses our early-deciding consensus algorithm in Figure 2. To broadcast m at time t, a process p sends (m, p, t) to all. If a process receives (m, q, t), it schedules the following action at time $t + D$: if it trusts q then it runs our consensus algorithm with m as the initial value; else it runs it with $\perp$ as the proposed value. If and when process p decides some value, p delivers that value if it is not $\perp$ else the process does nothing. If there are multiple concurrent broadcasts, the algorithm may start multiple instances of consensus — one for each broadcast. In order to differentiate between these instances, we have subscripted **propose** and **decide** with a unique identifier containing the identity q of the broadcaster and the time t of broadcast.

Notice that if the broadcaster fails in line 2 and only sends to a subset of the correct processes then some correct processes will not propose and start consensus. In that case, however, all processes that propose will propose $\perp$, because the broadcaster will be suspected at time $t + d$. This is the place in which we need the Uniform Validity property even if not all correct processes propose: it guarantees that the processes that decide will necessarily decide $\perp$ (the only proposed value), and thus they will all act in harmony by not delivering the message.

Theorem 6. *In a synchronous system with fast failure detection, timely reliable broadcast can be solved with an algorithm that delivers*[10] *in at most time* $\Delta = 2D + fd$ *using* $(f + 2)n$ *messages.*

Some simple optimizations can improve the delivery time of our algorithm. The first optimization requires that we order processes differently, so that the first round of the consensus algorithm is not executed by process 1, but by the process that wishes to broadcast — let us call this process q. When processes receive (m, q, t) and propose to the consensus box, they indicate that q should be the first process in the order (recall that there is a total order on the process id's); note that processes make a consistent choice on the first id before they start running consensus. Now the next processes in the

[10] If any delivery actually occurs. Note that if the sender crashes, it is possible that no process ever delivers any message.

order can be fixed or, for load balancing, it could be some hash function of q and the time t of broadcast[11]. Then, process q can start the consensus box by proposing m at time $t + D - d$ instead of $t + D$. It can do so because in our consensus algorithm, only the first process acts in the first d time units of execution. The other processes can join in at time $t + D$, after receiving the (m, q, t) message that q sent at time t. With this optimization we save d time units, i.e., processes deliver within time $2D + (f - 1)d$ and we save n messages, i.e., processes use $(f + 1)n$ messages. Note that if $D = d$ then in the failure-free case processes deliver within time D (it turns out that this is true even if there are failures but the *broadcaster* does not fail).

When $D \geq 2d$, a more important optimization allows processes to deliver within time D if the broadcaster does not fail. More precisely, suppose that p delivers the initial (m, q, t) message of q at some time u. Let us assume $u \geq t + 2d$ (if not, p waits until time $t + 2d$). If p trusted q at time $t + 2d$ then p can deliver m right away at time u, since p knows that (1) q's broadcast did not fail and will reach all processes by time $t + D$ and (2) all correct processes will trust q at time $t + d$ and will propose m to consensus. Note however that p still needs to run the consensus box, because other processes may have suspected q at time $t + 2d$ and they will need the consensus box to deliver m. Now if p still trusted q at time $t + 3d$ then p does not even have to bother starting consensus, since all alive processes must have trusted q at time $t + 2d$ and thus they delivered m at that time. With this optimization, if the broadcaster is correct then processes deliver within time D using only n messages[12]:

Theorem 7. *In a synchronous system with fast failure detection, timely reliable broadcast can be solved with an algorithm that delivers in at most time $\Delta = 2D + (f - 1)d$ using $(f + 1)n$ messages. If the broadcaster is correct then delivery takes at most time D using only n messages.*

6 Atomic Broadcast

In atomic broadcast, correct processes must deliver the same set of messages in exactly the same order. More precisely, (uniform) atomic broadcast is a (uniform) reliable broadcast that satisfies the following additional property:

– *(Uniform Total Order)* If any process delivers message m before m' then no process can deliver m' unless it has previously delivered m.

A traditional way to implement atomic broadcast is to order messages by increasing time of broadcast. To do so, the broadcaster timestamps its message, and a recipient can deliver a message as soon as it knows that there are no outstanding messages with a smaller timestamp. More precisely, if p wishes to atomically broadcast m at time t then p uses timely reliable broadcast to broadcast (m, t). If Δ is the maximum delay of the timely reliable broadcast box, a process that gets (m, t) from the box can atomically

[11] It is possible to modify the algorithm so that the broadcaster q selects the order of id's and includes it in its message, instead of just sending (m, q, t).

[12] It is worth noting that processes can deliver even earlier if they receive the message quickly, i.e., the actual message delay happens to be less than D.

```
Code for each process p:
1   To atomic-broadcast m at time t:
2     reliable-broadcast(m, t)
3   upon reliable-deliver(m, t) do
4     for each g ≤ f_max − 1, schedule at time t + Δ(g) + d:
5       if suspect at most g processes
6       then atomic-deliver in order all atomic-undelivered
7            messages with timestamp up to t
8     schedule at time t + Δ(f_max):
9       atomic-deliver in order all atomic-undelivered
10           messages with timestamp up to t
```

Fig. 4. Atomic broadcast algorithm with time complexity $\Delta(g) + d$.

deliver m at time $t + \Delta$, because it knows that the box will not later output any messages with a smaller timestamp than t. This idea can be made to work even if $\Delta = \Delta(f)$ is a function of the number f of failures — as long as processes can know f. It turns out there is a simple way to obtain a conservative bound on f: if at some time $u + d$ a process suspects exactly f_0 processes, then our failure detector guarantees that at time u we necessarily have $f \leq f_0$ (this is because our failure detector guarantees that a crashed process is suspected within d time of the crash). Using this idea, we get the algorithm shown in Figure 4.[13]

This algorithm works as follows: when a process p delivers (m, t) from the timely reliable broadcast box, it schedules for execution some subtasks that will atomic-deliver messages. For each $g \leq f_{max} - 1$, process p schedules at time $t + \Delta(g) + d$ a subtask that checks if p suspects at most g processes and, in that case, delivers in order all messages with timestamp up to t (including m). Process p also schedules at time $t + \Delta(f_{max})$ a subtask that will definitely atomically deliver m if it has not done so yet; here there is no need to test the number of crashed processes, because f_{max} is the maximum allowed.

It turns out that there is a way to shave off an extra d from the running time, as follows. Our early-deciding consensus algorithm of Figure 2 actually decides a little faster than within time $D + fd$: it decides by time $D + gd$ if there are only g suspicions at time $D + gd$.[14] Consequently, our timely reliable broadcast also delivers a little faster: a message broadcast at time t is delivered at time $t + 2D + (g - 1)d$ if there are only g suspicions at that time. That means that our atomic broadcast algorithm can deliver d time earlier: if m is broadcast at time t and at time $t + 2D + (g - 1)d$ there are only g suspicions then at this time it can atomically deliver all messages with timestamp up to t, because there are no messages with lower timestamp in transit. With this optimization, we get the following:

[13] Strictly speaking, this algorithm requires that $\Delta(f)$ be a monotonically nondecreasing function of f, which is always the case for our algorithms.

[14] To see why this can be faster, note that with our failure detector a process is only suspected if it has crashed, so at any time the number of suspicions is at most the number of crashes.

Theorem 8. *In a synchronous system with fast failure detection, atomic broadcast can be solved with an algorithm that delivers in at most time $2D + (f-1)d$ using $(f+1)n$ messages. In the failure-free case, it delivers in at most time D using only n messages.*

7 Losses in the Outgoing Buffers

In many systems, if a process crashes soon after sending a message, that message may be lost because the low-level outgoing message buffers of that process get wiped out. We now extend our model to allow this type of message losses. We assume that if the sender of a message crashes within δ time of the sending, this message may be lost. Here δ is a system-dependent constant (clearly $\delta < D$). More precisely, we weaken the No Loss property so that it guarantees no loss only if the sender remains alive for δ time:

– *(δ-Hold No Loss)* If p sends a message m to q at time t and p does not crash by time $t + \delta$ then q eventually receives m from p.

Note that D-Timeliness together with δ-Hold No Loss imply that if p sends m to q at time t and does not crash by time $t + \delta$ then q receives m by time $t + D$. Moreover, if p crashes between times t and $t + \delta$ then the messages that it sent may or may not be received. In particular, if p sent to both q and q' at time t, it is possible that only one of the messages is received. Note that if we set $\delta = 0$ then we get our previous model with no message losses.

Throughout Section 7, we assume that D is a multiple of $d + \delta$ (this can always be ensured by increasing D), but we handle the general case in detail in the full version of the paper.

7.1 Algorithms

In the full paper, we show how to modify our algorithms to work in the lossy model above. We have the following results:

Theorem 9. *Consider a synchronous system with fast failure detection, and suppose that messages sent δ time before the sender crashes may be lost.*

– *Uniform consensus can be solved with an algorithm that decides in at most time $D + f(d + \delta)$ using $(f + 1)n$ messages.*
– *Terminating reliable broadcast can be solved with an algorithm that delivers in at most time $D + f(d + \delta)$ using $(f + 1)n$ messages.*
– *Timely reliable broadcast can be solved with an algorithm that delivers in at most time $\Delta = 2D + (f - 1)(d + \delta)$ using $(f + 1)n$ messages. If the broadcaster is correct then delivery takes at most time $\max\{D, 2d + \delta\}$ using only n messages.*
– *Atomic broadcast algorithm can be solved with an algorithm that delivers in at most time $2D+(f-1)(d+\delta)$ using $(f+1)n$ messages. In the failure-free case, it delivers in at most time $\max\{D, 2d + \delta\}$ using only n messages.*

7.2 Lower Bound

In the full version of the paper we show that, if $D \geq d + \delta$, our consensus algorithm is time optimal, by proving that no algorithm can guarantee decision in less time than $D + f_{max}(d + \delta)$. This lower bound result automatically carries over to terminating reliable broadcast, timely reliable broadcast and atomic broadcast, since all these problems can be easily used to solve consensus without any extra time.

Theorem 10. *Consider a synchronous system with fast failure detection. Suppose that messages sent δ time before the sender crashes may be lost and $D \geq d + \delta$. Every consensus algorithm has a run in which decision takes time at least $D + f_{max}(d + \delta)$.*

Observation 11 *No early-deciding consensus algorithm can always ensure decision in less time than $D + f(d + \delta)$.*

Note that Theorem 10 is a generalization of Theorem 3 of Section 3.3: we obtain the latter by taking $\delta = 0$.

8 Discussion

A consensus lower bound in the continuous-time synchronous model. There is a well-known lower bound of $1 + f_{max}$ rounds to solve consensus in synchronous systems without failure detectors [18]. This result is for the round-based model, in which processes execute in lock-step rounds, and it does not immediately translate to the continuous-time synchronous model, in which there is no notion of round and processes take steps at any time they wish. In the latter model, one would expect a time lower bound of $(1 + f_{max})D$ to solve consensus. Indeed, such a bound is correct and it follows from Theorem 10: we simply take $d = D$ and $\delta = 0$. This seems to be the first proof of this fact.

Issues on achieving fast failure detection. Some fault-tolerant and real-time systems have fast built-in circuitry to detect failures, but often failure detection is not available in hardware and so it needs to be implemented in software by message passing. To achieve small detection time, one needs to use expedited delivery for the failure detector messages and send them frequently. In practice the bandwidth available for expedited messages is limited and so we should reduce the number of failure detection messages as much as possible. This can be achieved as follows. First, note that the straightforward way implement a perfect failure detector is for each process to periodically send I-am-alive messages to each other — a total of $O(n^2)$ periodic messages. It turns out, however, there are better failure detector implementations that only send linearly many messages while impacting the detection time by only a small factor (using similar techniques as in [3]). Second, note that our consensus algorithms do not require a failure detector among all n processes, but only among $f_{max} + 1$ of them: in many applications, f_{max} is significantly smaller than n. Thus, only $f_{max} + 1$ processes need to run the failure detector to solve consensus.[15] Third, if the failure detector is implemented as a shared service [8,15,13], the failure detector traffic does not significantly increase as the number

[15] This particular optimization does not apply to our broadcast algorithms because any process is allowed to broadcast and the broadcaster needs to be monitored by the failure detector.

of concurrent instances of consensus and broadcast increase: such an implementation sends machine-to-machine messages that combine the I-am-alive messages of different applications that share the same machine. Finally, note that [13] has shown that it is possible to implement a failure detector in a local area network with very fast detection time while maintaining a reasonably low total bandwidth and processor consumption: In an Ethernet network with (a) a total consumption chosen to be limited by 5%, (b) $f_{max} = 5$ and (c) n ranging from 16 to 1024, the optimal values of the maximum detection time d ranged from 51.9ms to 68.5ms (as n varied), while the maximum message delay D of regular messages ranged from 106.6ms to 6 828.7ms *plus* the maximum sojourn time of a regular message in the interprocess waiting queues of sender and receiver.

References

1. M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.
2. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, Apr. 2000.
3. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lecture Notes on Computer Science, Oct. 2001.
4. Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical Report UBLCS-95-18, Dept. of Computer Science, University of Bologna, Bologna, Italy, November 1995.
5. A. Casimiro, P. Martins, and P. Veríssimo. How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–134, Sept. 2000.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996. A preliminary version appeared in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, Aug., 1991, 325–340.
7. W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan. 2002.
8. B. Deianov and S. Toueg. Failure detector service for dependable computing (fast abstract). In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages B14–B15. IEEE Computer Society, June 2000.
9. D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1):191–204, Jan. 1985.
10. D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, Apr. 1990.
11. R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 41–50, Sept. 1995.
12. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

13. J.-F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, Aug. 2002. Special issue on Asynchronous Real-Time Distributed Systems.
14. M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.
15. D. Ivan, M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, November 2001. Prototype of a shared failure detector service with QoS guarantees.
16. J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *ACM Computing Surveys*, 16(1):43–70, Mar. 1984.
17. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.
18. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.
19. G. Le Lann, 2001. Private communication with Astrium, Axlog, European Space Agency.
20. G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.
21. K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(1):147–171, Sept. 1995.
22. H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1399, Oct. 1995.