Synchronous Atomic Broadcast for Redundant Broadcast Channels

Flaviu Cristian
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093−0114*

Abstract

We propose a synchronous atomic broadcast protocol for distributed real-time systems based on redundant broadcast channels. The protocol can tolerate a finite number f of concurrent processor crash failures, channel adapter performance failures and channel omission failures. Its message cost is optimal: when no failures occur only $f + 1$ messages are sent per broadcast. The cost implications of providing tolerance to other failure classes are also investigated.

1 Introduction

To achieve high-availability of a computing service despite failures, a key idea is to implement the service by a group of processes running on distinct processors. Replication of service state information among group members enables the group to provide the service even when some of its members fail since the remaining members know enough about the service state to be able to continue to provide it. Replication creates a need for a *communication service* which can be used by processes to disseminate state updates so that replica consistency is maintained despite random message delays and component failures. This communication service is variously termed reliable [CM84] or atomic [CASD85] broadcast. We will refer to it as *atomic broadcast*. Two classses of protocols for atomic broadcast have been proposed to date: *synchronous* protocols, such as [BD85] and [CASD85], which rely on synchronized clocks and the passage of clock time to achieve replica consistency, and

*This work was done while the author was with the IBM Almaden Research Center, San Jose; it was published in *The Journal of Real-Time Systems*, 2, pp. 195-212, 1990, Kluwer Academic Publishers.

asynchronous protocols, such as [BJ87, Car85, CM84, Lam89, MSMA90, VRB89], which use message acknowledgments.

A synchronous atomic broadcast protocol ensures the existence of a time constant Δ such that, even if up to f failures occur during a broadcast, the following properties are satisfied. *Atomicity*: if any correct processor delivers an update at time U on its clock, then that update was initiated by some processor and is delivered by all correct processors at time U on their clocks; *Order*: all updates delivered by correct processors are delivered in the same order by each correct processor, and *Termination*: every update whose broadcast is initiated by a correct processor at time T on its clock is delivered by all correct processors at time $T + \Delta$ on their clocks. The atomicity and order properties ensure that the same updates are applied to all correct replicas in the same order. Therefore if replicas are initially consistent, they stay consistent. The termination property ensures that updates broadcast by correct processors are applied to each correct replica Δ time units later. In this way, all correct replicas display identical contents at identical clock times.

Synchronous atomic broadcast protocols are needed for critical real-time applications which must enforce bounds on response times even when component failures occur. This bounds are achieved by assuming that message delays among correct processors are bounded and there exist enough redundant communication paths between processors so that, if all paths are used in parallel for a broadcast, the probability that all fail during the broadcast is negligible. The bounded message delays assumption requires that the processors which implement a synchronous protocol be controlled by real-time operating systems capable of guaranteeing bounds on task scheduling delays. When message delays are not bounded, that is, communication performance failures causing partitions can occur, broadcast termination within a bounded time cannot be guaranteed. Asynchronous atomic broadcast protocols sacrifice termination for the sake of providing tolerance to communication performance failures, including partition failures. Because they do not guarantee a bound on the time it takes to broadcast an information in the presence of failures, asynchronous atomic broadcast protocols cannot be used in critical applications which must ensure that deadlines are always met (even in the presence of component failure occurrences). However, since asynchronous protocols do not require that communication delays be bounded, they can be implemented by processors controlled by conventional time-sharing (non real-time) operating systems, for applications where the cost of missing some deadlines is not too high.

This paper proposes a synchronous atomic broadcast protocol for distributed real-time systems based on redundant broadcast channels. Examples of popular broadcast channels are Ethernet, Token Bus, Token Ring, and fiber optic FDDI. To broadcast an update in the absence of failures (that is, the vast majority of times), the protocol sends $f + 1$ messages, independently of the number n of processors participating in broadcast. We show that this is the minimum failure-free message overhead that must be spent to achieve both atomicity and termination despite up to f failures, so our protocol is optimal in this respect. The very low message overhead makes the protocol scalable to systems with large numbers of

processors.

The failure classes considered are processor crashes, channel adapter performance failures, and channel omission failures. The paper uses the nested failure classification of [CASD85] which we briefly recall. An *omission* failure occurs when a component omits to respond to an input event; we talk of a *crash* failure when after a first omisssion the component systematically omits to respond to subsequent input events until restart. A *timing* failure occurs when a component either omits to respond or responds too early or too late. Most of the timing failures observed in practice are late timing failures, or *performance* failures. Crash failures are a subclass of omission failures, omission failures are a subclass of timing failures, and timing failures are a subclass of the class of all (or *arbitrary*) failures.

We begin by stating our assumptions. We then describe the basic idea behind our protocol. A detailed protocol description follows. The cost implications of providing tolerance to other failure classes are then briefly investigated. A comparison with previous work concludes the paper.

2 System model and asssumptions

We consider a system of *processes* which maintain replicated state information. Updates to replicas are disseminated by using the atomic broadcast service implemented by n distributed *processors* with different, totally ordered, names. Each processor is connected via $f + 1$ independent channel *adapters* to $f + 1$ independent broadcast *channels* (see Figure 1).

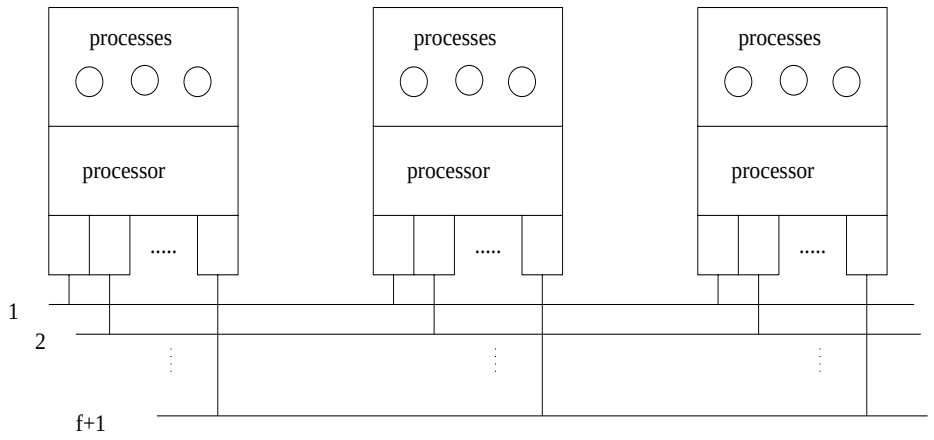

Figure 1: System Model

A process can ask a processor to broadcast an update σ by invoking a *SEND* command. To receive broadcasts from its underlying processor a process invokes a *RECEIVE* command.

Each adapter contains buffer memory, logic to control access to its channel, as well as logic for sending messages to and receiving messages from the channel. It is convenient to understand each adapter as being composed of two unidirectional halves: an *out-adapter* which accepts messages from its attached processor and transmits them on the channel, and an *in-adapter* which accepts messages from the channel and delivers them to its attached processor. To transmit a message on a channel c, a processor invokes a *send* command on the out-adapter for c. To receive a broadcast message from its attached in-adapters, a processor invokes a *receive* command. After a message is entrusted to an out-adapter for transmission, it waits in the out-adapter buffer memory until its transmission on the channel becomes possible. Whenever an out-adapter gets control of its attached channel c, it inserts a message on c. The broadcast channel c then delivers the message to all in-adapters connected to it. As in-adapters receive messages they deliver them to their attached processors, which in turn deliver updates to their processes.

We make the following assumptions.

1. *Bounded broadcast rate.* The rate at which updates are generated is bounded. This bound is smaller than the rate at which processors can *receive* the messages which carry the updates as well as the rate at which processes *RECEIVE* these updates.

2. *Broadcast channels suffer only omission failures.* The error detecting codes used by the physical channel transmission protocol detect any message corruption due to transmission errors, so that corrupted messages can be discarded by in-adapters. Thus, a channel c can behave in only the following two ways. When no channel failure occurs a message m transmitted on c is delivered to all correct in-adapters attached to c within a short constant time C, as measured on any correct processor clock. When a channel failure occurs only a (possibly empty) subset of the in-adapters attached to c get m within C clock time units from its insertion on c. In all cases, the message m disappears from the channel C clock time units after its transmission began.

3. *Out-adapters suffer only performance failures.* Under normal circumstances, the delay between the moment a processor enqueues a message m on an out-adapter by invoking a *send* command and the moment the out-adapter begins to successfully transmit m on its attached channel is bounded in time by a constant O, as measured on any correct processor clock. Temporarily, this delay can become greater than O. For example, in a Token Ring, the loss of a token can delay all messages enqueued in out-adapters until a new token is regenerated. Similarly, the occurrence of too many collisions on an Ethernet can result in excessive delays between the moment an out-adapter attempts to transmit a message and the moment the adapter successfully transmits the message without detecting further collisions. Thus, in response to a *send(m)* event, an out-adapter o can behave in only the following two ways: either o successfully sends m on its attached channel c within 0 clock time units, or o suffers a performance failure. When o fails to insert m on c within 0 clock time units, it might either send m on c

later or it might never send m on c.

4. *In-adapters suffer only omission failures.* The dealy between the moment a channel delivers a message to an in-adapter and the moment the message is *received* by the attached processor is bounded in time by a constant I, as measured on any correct processor clock. Thus, an in-adapter can only behave in one of the following two ways: when it receives a message m from its attached channel, it either delivers m to its processor within I clock time units from the receipt of m or it never delivers m.

5. *Processors suffer only crash failures.* The delay between the moment a processor initiates processing a $SEND(\sigma)$ command invoked by a process and the moment the processor finishes enqueueing messages containing σ on all its $f + 1$ attached out-adapters is bounded by a time constant P, as measured on any correct processor clock. We also assume that P is a bound on the delay which can elapse between the moment an adapter is ready to deliver a message m to a processor p and the moment p *receives* and processes m. To ensure that this assumption holds at run-time, it is necessary that the operating system(s) controlling the processors be real-time executive(s), capable of enforcing bounded delays for processing atomic broadcast messages under worst case load conditions. Under this assumption, a processor p which interprets a $SEND(\sigma)$ command issued by a process can behave only in the following two ways. Either p correctly enqueues messages m containing σ on all its attached out-adapters within P time units, or p crashes after enqueueing m only on a (possibly empty) subset of adapters within P time units. In the latter case, m is never enqueued on the remaining out-adapters.

6. *Processors have access to correct clocks that are approximately synchronized.* A correct clock drifts from real time at a rate whose absolute value is bound by a small, manufacturer specified, constant p. A correct clock also yields different, monotonically increasing time values each time it is read. We assume that processor clocks are approximately synchronized within a known, constant, maximum deviation ϵ. (Clock synchronization algorithms that work under the above assumptions on processors, adapters and channels can be found in [Cri89, CF94].)

7. *Tasks can be scheduled for certain deadlines.* The real-time executive controlling the execution of processors provides a *schedule A(B) at T* command that allows a task (or process) A to be scheduled for execution at local time T with input parameters B. An invocation of *schedule A(B) at T* at a local time $U > T$ has no effect, and multiple invocations of *schedule A(B) at T* have the same effect as a single invocation.

8. *At most $f \leq n$ - 2 components can be faulty during a broadcast.* By a component, we mean a processor, an adapter, or a channel. This assumption ensures that if a correct processor s successfully completes executing *send(m)* commands on all its $f + 1$ out-adapters, then m will be received by each correct processor r. Indeed, there exist $f + 1$ independents *paths*, which we denote 1, 2, ..., $f + 1$, between s and r,

where path c consists of the out-adapter interfacing s to channel c, the channel c, and the in-adapter interfacing c to r. At most f faulty components leave all components on at least one path between s and r correct, thus, r receives a copy of m on at least this path. (Algorithms that detect at run-time a violation of this assumption and recover from such events are described in [SSCA87].)

3 Basic idea: lazy forwarding

Assumptions 2-6, ensure that, in the absence of failures, any update σ accepted for broadcast by a processor s at time T on its clock, is received and processed by any processor q by time $T + \delta + \epsilon$ on q's clock, were $\delta = P + O + C + I + P$ denotes the processor-to-processor message delay bound. The term ϵ has to be added because q's clock can be as far as ϵ time units ahead of the clock of s.

In the presence of failures, the situation is more complicated. If at least two failures can occur during a broadcast ($f > 1$), then a correct processor q may not receive by time $T + \delta + \epsilon$ on its clock a message m sent by a processor s at time T on its clock, while another correct processor r may receive m by $T + \delta + \epsilon$ on its clock. To see how this is possible, consider the following scenario: the sender s crashes after enqueuing m only on the out-adapter for channel 1, and a transmission error on channel 1 corrupts m after it is delivered to the in-adapter for r and before it arrives at the in-adapter for q (Figure 2). The first adapter will deliver m to r while the second adapter will discard the corrupted message it receives. Thus, when $f > 1$, it is sometimes necessary that a processor which receives a new message m *forward it*, to ensure m is received by all correct processors.

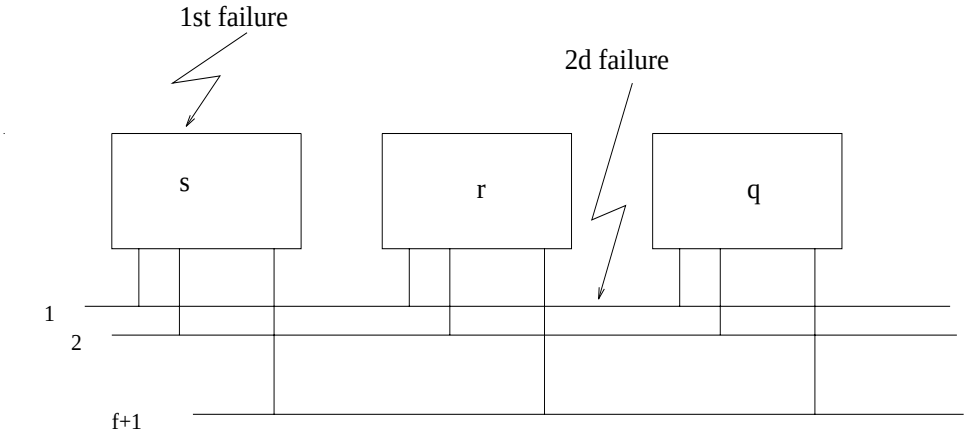

Figure 2: Worst two failures scenario

One possible rule for message forwarding is that described in [CASD85]: a processor forwards any new message it receives from a channel on all other channels as soon as it receives

the message. With this *prompt forwarding* rule, $(f + 1) + (n - 1)f = nf + 1$ messages are sent per broadcast. The main drawback is that message forwarding always takes place, even when no failures occur (the vast majority of cases). The protocol we present is based on a new, lazy forwarding rule. The goal is to not forward when there is no need to, for example when no failures occur. A simple version of the rule follows. The complete rule is given later.

Simple lazy forwarding rule. A sending processor s enqueues any message m to be broadcast on the out-adapters to channels 1, 2, ..., $f + 1$ in this (increasing) order. Let p be an arbitrary processor different from s that receives m and let c be the highest channel number on which p receives a copy of m. If $c \geq f$ then p does not need to forward m, else, p forwards m on channels $c + 1, \ldots, f$.

The above forwarding rule ensures the following *unanimity* property: if a processor s initiates the broadcast of a message m and some correct processor p receives m, then any correct processor q receives m. (When m does not reach any correct processor -this can only happen if s is faulty-there is no need to worry about forwarding m, since the atomic broadcast specification allows for the case that no correct processor delivers a message when its initiator fails.) We prove the above property as follows. Consider the two possible cases (a) c $\geq f$ and (b) c $< f$. If c $= f + 1$ then s enqueued m on all its $f + 1$ out-adapters, so the at most f components which can fail during the broadcast of m cannot prevent an arbitrary correct processors q from receiving at least a copy of m (remember there are $f + 1$ independent communication paths between s and q). If $c = f$ then (a1) either s correctly sent m on all its out-adapters and a failure on the path $f + 1$ between s and r prevented r from receiving m from its $f + 1$ in-adapter or, (a2) m was only sent by s on channels 1, 2, ..., f (because s crashed). We have already mentioned that when s sends m correctly on all channels, all correct processors get at least one copy of it, thus, if (a1) is true, any correct processor q gets m. If (a2) is true, the crash of s counts as one failure. Since by assumption 8 we assume that at most f components can fail during the broadcast of m, it follows that besides the faulty sender s there exist at most f - 1 other faulty components. These cannot prevent a correct processor q from getting at least a copy of the message m broadcast on the f independent paths numbered 1 ... f. Consider now case (b):$c < f$. This can only happen when at least one failure has occurred before the correct processor r received m: either s crashed or a channel with number higher than c failed, or an adapter attached to such a channel failed. Since processor r forwards m on channels $c + 1$, ..., f, and the at most f - 1 additional faulty components which, by assumption 8, may exist cannot affect all of the f disjoint communication paths 1, 2, ... , f on which copies of m were sent (by s and r), it follows that any correct processor q must receive at least one copy of m on one of these paths.

The key advantage of lazy forwarding is that, in the absence of failures, only $f + 1$ messages are sent per broadcast. Since in a general system, no processor p knows in advance the time at which another processor s chooses to broadcast, $f + 1$ is the *minimum* number of

messages that must be sent if both atomicity and termination must be provided despite up to f component failures. The minimality of the $f + 1$ message overhead can be proven by contradiction as follows. Assume there exists a synchronous atomic broadcast protocol P, tolerant of up to f component failures and having termination time Δ, which, in the absence of failures, uses only $k \leq f$ messages to atomically broadcast an update. Consider now a broadcast scenario S in which a sender s broadcasts an update σ at time T on its clock and no failures occur during the broadcast of σ, so that all processors deliver σ at time $T + \Delta$ on their clocks. Since by assumption 8: $k \leq n$ - 2, it follows that at least two correct processors p and q do not send messages in S. Consider now another broadcast scenario S' in which s broadcasts the update σ as in S and all k in-adapters interfacing the correct processor q to the k channels on which σ was transmitted fail. Scenarios S and S' are indistinguishable to correct processor p by construction. Since p delivers σ at $T + \Delta$ in S, p also delivers σ in S'. On the other hand, scenario S' was constructed so that q will not know by $T + \Delta$ that s has initiated a broadcast at time T, so q will not deliver σ at $T + \Delta$ in S', thereby violating the atomicity property that P is supposed to achieve. Thus, the hypothesized protocol P does not exist.

4 Detailed protocol description

We begin our detailed description by first considering the simpler case of providing tolerance to a single failure ($f = 1$). In this particular case no forwarding is necessary, since a single failure cannot cause both independent communication paths between two correct processors to fail. By first studying this simple -but practically important- case, we hope to make it easier to follow the description of the more complex protocol for $f > 1$.

4.1 The single-fault tolerant protocol

Each message broadcast by a processor carries its initiation time (or *timestamp*) T, the name of the source processor s and an update σ. Since all processor names are distinct, and by assumption 6 each clock reading yields a different value, the T and s values uniquely identify each broadcast. As messages are received by a processor p, these are processed and stored in a *history* log H, local to p, until p delivers them to its local processes. The order property required of atomic broadcasts is achieved by letting each processor deliver the updates it receives in timestamp order, by ordering the delivery of updates with identical timestamps in increasing order of their initiator's name, and by ensuring that no correct processor p begins the delivery of updates timestamped T before time $T + \delta + \epsilon$ on its clock, at which point p is certain that it has received all updates with timestamp at most T that it may ever have to deliver. We call the time $\Delta = \delta + \epsilon$ the protocol *termination time*, and $T + \Delta$ the *delivery time* for updates with timestamp T.

```
1       task Start;
2         const Δ = δ + ε;
3         var T: Time; σ: Update; s: Processor;
4         cycle SEND(σ); T ← clock;
5             for c = 1 to 2 do send(T,myid,σ) on c;
6             H ← H⊕(T,myid,σ);
7             schedule Deliver (T) at T + Δ;
8       endcycle;
```

Figure 3: Start Task of the single-fault tolerant protocol.

```
1       task Receive;
2         const Δ = δ + ε;
3         var U, T: Time, σ: Update; s: Processor;
4         cycle receive(T, s, σ) from c; U ← clock;
5            if U ≥ T + Δ then "late message" iterate fi;
6            if T∈dom(H)&s∈dom(H(T)) then "deja vu" iterate fi;
7            H ← H⊕ (T, s, σ);
8            schedule Deliver (T) at T + Δ;
9       endcycle;
```

Figure 4: Receive Task of the single-fault tolerant protocol.

To keep the local history H *finite*, messages are purged from H when the updates contained in them are delivered. However, a simple-minded application of the above garbage-collection rule would not be sufficient for ensuring that local histories remain bounded, since it is possible that copies of a message (T, s, σ) continue to be received by a correct processor p after the delivery time $T + \Delta$ has passed on p's clock (for example because an out-adapter attached to the sending processor suffers a performance failure). To prevent such late residual messages from accumulating in local histories, we use a *late message* acceptance test. This test discards a message (T, s, σ) if it arrives at a local time U past the delivery time $T + \Delta$, that is, if $U \geq T + \Delta$. The late message acceptance test and assumption 1 ensure that local histories stay bounded.

A detailed description of the single-fault tolerant atomic broadcast protocol is given in Figures 3, 4, and 5. Each processor runs three concurrent tasks: a Start task (Figure 3) that initiates atomic broadcasts, a Receive task (Figure 4) that receives atomic broadcast messages and a Deliver task (Figure 5) that delivers broadcast updates to processes which invoke *RECEIVE* commands. In what follows we refer to line j of figure i as *(i.j)*.

```
1       task Deliver (T:Time);
2         var p: Processor; val: Processor → Update;
3         val ← H(T);
4         while dom(val) ≠ {}
5         do p ← min (dom(val));
6           RECEIVE(val(p));
7           val ← val\ p;
8       od;
9       H ← H\T;
```

Figure 5: Deliver Task of the single-fault tolerant protocol.

A process triggers the broadcast of an update σ by invoking the $SEND(\sigma)$ command exported by its local processor. This will activate the Start task at the matching $\overline{SEND}$ entry point with σ as input (3.4). The broadcast of σ is identified by the local time T at which σ is received (3.4) and the identity of the sending processor, obtained by invoking the function *myid* (3.5). This function returns different processor identifiers when invoked by distinct processors. The broadcast of σ then proceeds by invoking the *send* command exported by the out-adapters to channels 1, 2 (3.5). We do not assume that the FOR loop command is atomic with respect to crashes: a processor crash can prevent messages from being sent on some out-adapters. The fact that the broadcast of σ has been initiated is then recorded in the history variable H shared by all broadcast tasks:

$$var H : Time \rightarrow (Processor \rightarrow Update).$$

We assume H is initialized to the empty function at processor start. The variable H keeps track of ongoing broadcasts by associating with instants T in Time a function *H(T)* (of type Processor → Update). The domain of *H(T)*, denoted dom(*H(T)*), consists of names of processors that have initiated atomic broadcasts at time T on their clock. For each such processor p, *H(T)(p)* is the update broadcast by p at T. We use the following operators on histories. The update "$\oplus$" of a history H by a message (T, s, σ) yields a (longer) history, denoted $H \oplus (T, s, \sigma)$, that contains all the facts in H, plus the fact that s has broadcast σ at local time T. The deletion "\" of some instant T from a history H yields a (shorter) history, denoted H\T, which does not contain T in its domain, that is, everything about the broadcasts that were initiated at time T is deleted. Once the history H is updated (3.6), the Deliver task is scheduled to deliver σ at local clock time $T + \Delta$ (3.7).

The Receive task uses the *receive* command to receive messages formatted as (T, s, σ) from in-adapters. The identity c of the channel on which a message is received is a return parameter of this command (4.4). In describing this task, we use double quotes to delimit comments and the command *iterate* to mean *terminate the current iteration and begin the next cycle*, (4.5, 4.6). If a received message is a duplicate of a message that was already

received (4.6) or delivered (4.5) it is discarded. A message is inserted in the history variable (4.7) only if it passes both the *late message* and *deja vu* acceptance tests (4.5, 4.6). When a message (T, s, σ) is inserted in the history variable, we say that it is *accepted* (for delivery). Once a message originated at clock time T is accepted, the Deliver task is scheduled to start at local time $T + \Delta$ (4.8). The Deliver task (Figure 5) starts at clock time $T + \Delta$ to deliver to processes which have invoked *RECEIVE* commands (5.6) updates timestamped T in increasing order of their sender's identifier((5.5)-(5.8)) and to delete from the local history H everything about broadcasts initiated at time T (5.9).

4.2 The multiple-fault tolerant protocol

To guarantee order, the broadcast termination time Δ must be chosen to be at least equal to the worst case delay which can elapse between the time a broadcast is initiated and the time by which all correct processors have received the broadcast. This ensures that, when any correct processors p and q begin to deliver a broadcast initiated at time T, p and q know the set B of *all* broadcasts with timestamp at most T that a correct processsor will ever have to deliver. Processors p and q can then order the delivery of the updates in B identically to achieve order. The above worst case delay depends on how many times a message can be forwarded before it reaches a correct processor for the first time. Below we simultaneously derive Δ and generalize the simple lazy forwarding rule introduced earlier. We also make the rule sufficiently precise so that it becomes implementable, for example, we give phrases such as: let c be the highest channel on which a processor r receives m the more precise meaning: let c be the highest channel on which processor r receives m *by a certain local time*. (Without a point in time by which r must decide, r might wait forever in the hope of receiving m on a higher channel.)

To enable processors to determine how many times a message was forwarded, we add to each message a hop count h. A sender s initiating the broadcast of a message m sets h to 1. Each time a processor forwards m, h is incremented by 1. Since a combination of processor crash and out-adapter performance failures can cause a message originated by a processor to be delayed more than δ time units before it is received by another processor, we use the timeliness test of the second protocol of [CASD85] to detect and discard late messages. A message timestamped T with hop count h received at local time U will be called *timely* if

$$U < T + h(\delta + \epsilon).$$

A message is *accepted*, that is, is inserted in the local history variable, only if it passes the *late message* and *deja vu* tests mentioned earlier and *is* timely. The above timeliness test ensures that if a correct processor p accepts a broadcast (T, s, σ, h) that needs to be forwarded, then the messages (T, s, σ, $h + 1$) that p will forward will also be timely for all correct processors.

In a system based on redundant broadcast channels governed by the failure assumptions 2-6, the failure scenarios which can delay most the reception of a message by a first correct processor are of the type examined in Figure 2, where a processor crash - after a message is output just on one out-adapter- is followed by a performance failure of the out-adapter or an omission failure of the attached channel. For example if f is even, that is, $f = 2k$ for some integer $k \geq 1$, the following scenario containing $2k$ failures can lengthen the route which a message $(T, s, \sigma, 1)$ must travel before being accepted by a first correct processor to k hops: the sender $s = p_0$ crashes after sending $(T, s, \sigma, 1)$ on channel 1 and channel 1 suffers an omission failure after delivering $(T, s, \sigma, 1)$ to a single correct in-adapter attached to processor p_1 which accepts the message, p_1 crashes after forwarding $(T, s, \sigma, 2)$ on channel 2 and channel 2 suffers an omission failure after delivering $(T, s, \sigma, 2)$ to a single correct in-adapter attached to processor p_2 which accepts the message, ... p_{k-1} crashes after sending (T, s, σ, k) on channel k and channel k suffers an omission failure after having delivered T, s, σ, k to a single correct in-adapter attached to a first processor p_k which accepts the message. If f is odd, that is, $f = 2k + 1$ for some integer $k \geq 1$, then the first correct processor to receive a message (T, s, σ, h) sent by a faulty processor s can be as far as $k + 1$ hops away from s. Indeed, an initial scenario containing $f - 1 = 2k$ failures indentical to the one above can lead to the acceptance of (T, s, σ, k) by a processor p_k which is k hops away from $s = p_0$. Another failure (the last possible according to assumption 8) can crash p_k after forwarding $(T, s, \sigma, k + 1)$ on channel $k + 1$ so the first correct processor to receive a message $(T, s, \sigma, *)$ is $k + 1$ hops away from s_0.

The above worst-case-delay-to-first-correct-processor scenarios suggest that, if $f = 2k$, $k \geq 1$, then $\Delta = (k + 1)(\delta + \epsilon)$ is an acceptable broadcast termination time, since if p is a correct processor that accepts as timely a message (T, s, σ, h) arriving on highest channel $c < f + 1 - h$ which needs forwarding, $h \leq k$, then p has enough time (i.e., *time units*) to forward $(T, s, \sigma, h + 1)$ to all correct processors before the delivery time $T + \Delta$ occurs on their clock. If $f = 2k + 1$, $k \geq 1$ the time $(k + 2)(\delta + \epsilon)$ would be an acceptable termination time since we have seen that the latest clock time by which a correct processor can accept a broadcast is $(k + 1)(\delta + \epsilon)$. This termination time can be improved by observing that, if ever a correct processor p accepts by local time $T + k(\delta + \epsilon)$ a message (T, s, σ, k) which, because of a failure scenario equivalent to the $2k$ failures scenario described previously could be missed by some other correct processor by local time $T + k(\delta + \epsilon)$, then if p forwards $(T, s, \sigma, k + 1)$ on the channels $k + 1$, $k + 2 = f + 1 - k$, the last $(2k + 1)$th failure allowed by assumption 8 will not prevent the other correct processors from receiving at least a copy of $(T, s, \sigma, k + 1)$ by time $T + (k + 1)(\delta + \epsilon)$. Thus, it is safe to reduce the termination time to $\Delta = (k + 1)(\delta + \epsilon)$ when $f = 2k + 1$, $k \geq 1$.

Knowledge of the hop count h allows us not only to detect late messages, but also to minimize the set of channels on which a processor has to forward messages. To this end, we now give the complete rule for lazy forwarding.

Lazy Forwarding Rule. To initiate a broadcast, a sender s enqueues messages $(T, s, \sigma, 1)$ on

its out-adapters to channels 1, 2, ..., $f+1$ in this order. Let (T,s,σ,h), $h \leq k$ be a message accepted by a processor $p \neq s$ and let c be the highest channel on which p receives a copy of the message by local time $T + h(\delta + \epsilon)$. If at $T + h(\delta + \epsilon)$ on p's clock, $c < f + 1 - h$, then p forwards $(T,s,\sigma,h+1)$ on channels $c+1, \ldots, f+1-h$, else, p does not forward.

This forwarding rule, like the original simple lazy forwarding rule, ensures the *unanimity* property mentioned earlier: if a correct processor p accepts a broadcast by time $T + \Delta$ on q's clock, then each correct processor q accepts the broadcast by time $T + \Delta$ on q's clock. A proof that a broadcast protocol satisfying the unanimity property satisfies the atomicity, order, and termination properties required for atomic broadcast is given in [CASD85]. The proof of the unanimity property is based on the following lemma (proven in the Appendix):

LEMMA. Let (T,s,σ,h) $h \leq k$ be a message accepted by a processor p and let c be the highest channel on which p receives a copy of the message by local time $T + h(\delta + \epsilon)$. If at time $T + h(\delta + \epsilon)$ on p's clock $c \leq f + 1 - h$, then at least h component failures have occurred since the broadcast was initiated, else, at least $h - 1$ failures have occurred.

The proof of the unanimity property is by case analysis. If the highest channel c on which p receives a timely message (T, s, σ, h) by time $T + h(\delta + \epsilon)$ on its clock is such that $c < f + 1 - h$, $h \leq k$, then, by the lazy forwarding rule, p forwards $(T, s, \sigma, h+1)$ at local time $T + h(\delta + \epsilon)$ on channels $c+1, \ldots, f+1-h$. Thus, the at most $f - h$ components which, by assumption 8 and the above lemma, can fail after p forwards $(T,s,\sigma,h+1)$ at local time $T + h(\delta + \epsilon)$ on channels $c+1, \ldots f+1-h$. Thus, the at most $f-h$ components, which by assumption 8 and the above lemma, can fail after p forwards $(T,s,\sigma,h+1)$ will not prevent at least one of the messages $(T,s,\sigma,*)$ sent on channels 1,2,..., $f+1-h$ by s, the processors which have forwarded $(T,s,\delta,*)$ before p, and p, to reach all correct processors. If $c = f + 1 - h$, $f + 1 - h$ messages have already been sent on channels 1, 2, ...,$f + 1 - h$, the at most $f - h$ component failures which can occur after p's clock displays time $T + h(\delta + \epsilon)$ will not prevent at least one of these messages from reaching all correct processors. If at local time $T + h(\delta + \epsilon)$ $c > f + 1 - h$, at least $f + 2$ - h messages have already been sent, the at most $f - (h - 1) = f + 1 - h$ components which, by the above lemma, can still fail cannot prevent at least one of these messages from reaching all correct processors.

The lazy forwarding rule, like the simple rule given previously, ensures that in the absence of failures only $f + 1$ messages are sent for each broadcast. The worst case message cost in the presence of failures is $(f + 1) + (f - 1)(n - 1) = (f - 1)n + 2$.

The Start, Receive, and Forward tasks of the multiple-fault tolerant protocol are given in Figures 6, 7, and 8. In expressing the protocol termination time (6.2), we denote "/" the integer division operator and $[x]$ the floor function, that is, if x is a rational number, $[x]$ denotes the greatest integer i such that $i \leq x$. In addition to the history variable H, the tasks of the multiple-fault tolerant protocol also share a variable C of type

$$varC : Time \rightarrow (Processor \rightarrow \{1, \ldots, f+1\}).$$

```
1       task Start;
2         const Δ = [f/2] (δ + ε) + (δ + ε);
3         var T: Time; σ: Update; s: Processor;
4         cycle SEND (σ); T ← clock;
5             for c = 1 to f + 1 do send(T,myid,σ,1) on c;
6             H ← H⊕(T,myid,σ);
7             schedule Deliver(T) at T + Δ;
8       endcycle;
```

Figure 6: Start Task of the multiple-fault tolerant protocol.

For each broadcast (T,s) in the history, C records the highest channel on which a message (T,s, *, *) was received. As for H, we assume that the C shared variable is initialized to the empty function. A broadcast initiation time T is purged from the domain of C at the same time it is purged from the domain of H (5.9). Because the Deliver task for the multiple-fault tolerant protocol is so similar to that described in Figure 5, we do not describe it in detail for brevity reasons.

We conclude our description by mentioning an optimiziation to the lazy forwarding rule which lowers the average number of messages forwarded when failures occur. When ϵ is large compared to δ, it is possible that a processor p among the processors which have to forward a message (T,s, σ, $h + 1$), $h < k$, forwards the message so early that other processors which had to forward the same message receive it before the forwarding deadline $T + h(\delta + \epsilon)$ occurs on their clocks. If q is such a processor, then q can omit to forward (T,s, σ, $h + 1$) when the deadline for forwarding occurs on its clock. What q needs to do, however, is to check at local time $T + (h + 1)(\delta + \epsilon)$ whether a crash has prevented p from sucessfully completing the forwarding. Processor q has to forward (T,s, σ , $h + 2$) at local time $T + (h + 1)(\delta + \epsilon)$ if the highest channel c' on which q has received a copy of (T,s, σ, $h + 1$) is smaller than f - h and, of course, q has not received other (early) copies of (T,s, σ, $h + 2$) forwarded by some other fast processor.

5 What does it cost to tolerate other failure classes?

Because of the timeliness tests (4.5, 7.6, 7.7) in-adapter or channel performance failures can be tolerated at no additional cost. An increase in cost is observed when processors (instead of adapters) can suffer performance failures, for example because standard time-sharing (not real-time) operating systems are used to control them. Indeed, if the actual delay between the time T at which a processor s accepts to broadcast an update σ and the moment the processor s finishes to enqueue messages (T,s, σ, 1) on all its $f + 1$ out-adapters can be

```
1       task Receive;
2         const Δ = [f/2] (δ + ε) + (δ + ε);
3         var U, T: Time; σ: Update; s: Processor; h: Integer;
4         cycle receive (T, s, σ, h) from c; U ← clock;
5           if U ≥ T + Δ then "late message" iterate fi;
6           if U ≥ T + h (δ + ε) then "too late to forward" iterate fi;
7           if T ∈ dom (H) & s ∈ dom (H(T))
8           then "deja vu" C(T)(s) ← max c,C(T)(s);
9           else H ← H⊕ (T, s, σ);
10                if h ≤ [f/2] & c < f + 1 - h
11                then C ← C⊕(T,s,c);
12                  schedule Forward (T,s,h) at T + h (δ + ε)
13                fi;
14                schedule Deliver (T) at T + Δ;
15          fi;
16      endcycle;
```

Figure 7: Receive Task of the multiple-fault tolerant protocol.

```
1       task Forward (T: Time; s: Processor; h: Integer);
2         if C(T)(s) < f + 1 - h
3         then for i = C(T)(s) + 1 to f + 1 - h do send(T,s,H(T)(s), h + 1) on i fi;
```

Figure 8: Forward Task of the multiple-fault tolerant protocol.

greater than P, the previous protocols no longer achieve atomicity despite the timeliness tests mentioned earlier. This can be demonstrated by the following counter-example for the two-fault tolerant protocol.

Let s, p, e, and g be four processors with correct clocks running at the speed of real-time, such that s and p are faulty and e and g are correct, but e's clock is earlier than (or in advance of) g's clock. To fix ideas, let $\delta = 8$ and $\epsilon = 6$, and assume that at real-time 0, the clocks were showing the times: $C_s(0) = 0$, $C_p(0) = 1$, $C_g(0) = 3$, $C_e(0) = 5$. Assume that, at real time 0, s accepts to broadcast an update σ but, because of a performance failure, s invokes the *send* (0, s σ, 1) command on channels 1, 2, and 3 at real-times 5, 6, and 7, respectively. Assume also that all messages take exactly 7 real-time units (note: $7 < \delta$) to be delivered to processors p, g, and e. Processor p will accept the message (0, s, σ, 1) coming on channel 1 at real-time 12 (because it arrives at local time $C_p(12) = 13$ before the deadline 14 for accepting messages with timestamp 0 and hop count 1 occurs at p) but will reject the copies coming on 2 and 3. Processors g and e will reject all copies of (0, s, σ, 1) because they will receive them at local times past the deadline for accepting messages with timestamp 0 and hop count 1. Assume now that, because of a performance failure, p forwards (0, s, σ, 2) on channels 1, 2, and 3 at real-time 17 instead of 14, and that all messages are delivered to g and e after exactly 7 real-time units, at real-time 24. Processor g will accept all copies of (0, s, σ, 2) coming on channels 1, 2, 3 because the delivery time 28 has not yet passed on its clock $C_g(24) = 27$, but processor e will reject all copies of the message on all channels as being late, because the delivery time 28 has passed on its clock $C_e(24)$=29. This violates atomicity.

The reasons why the previous protocol based on lazy forwarding does not work if processors can suffer performance failures, perhaps because they are controlled by non real-time operating systems, are as follows:

1. because of clock differences between correct processors, a message sent by a slow processor s on all channels can be accepted only by a subset of correct processors even if no other failures than the failure of s occur,
2. the hop count of a message received for the first time by a first correct processor can now be as high as f (as opposed to a maximum of $\lceil f/2 \rceil$ for the previous protocol).

Since a correct processor g which accepts a message m on a high channel c can no longer infer that any correct processor e also accepts m from channel c when no failures affect the components on the c path between g and e, to tolerate processor performance failures one has to use *prompt* fowarding. Moreover, because the termination time $(\lceil f/2 \rceil + 1)\ (\delta + \epsilon)$ does no longer ensure that if a correct processor p accepts a timely message m that needs forwarding, then all correct processors which receive the messages forwarded by p accept them as timely, we have to increase the termination time to $(f + 1)\ (\delta + \epsilon)$. This will ensure that if a correct processor p ever accepts a message m timestamped T as time by $T + h\ (\delta$

$+ \epsilon$) on its clock, where $h \leq f$, then p has enough time (i.e. at least $\delta + \epsilon$ time units) to forward m message as timely, too.

The second protocol of [CASD85] is based on prompt forwarding and has a termination time of $(f + 1)\ (\delta + \epsilon)$. If appropriately adapted to forward messages on all channels other than the incoming channel instead of on all links other than the incoming link, this protocol can be used to achieve atomic broadcast in systems based on redundant broadcast channels in the presence of processor timing failures and *locally consistent* [DSC89] clock failures. Roughly speaking, a clock failure is locally consistent if it cannot be detected by detection mechanisms local to a processor only; knowledge of the values of other clocks in the system or of external time might be necessary for that. A stopped clock or a clock that runs backward are examples of locally detectable faulty clocks (to detect such failures it is sufficient to compare successive clock readings). A monotonically increasing clock that drifts from real time at an actual rate inferior to -ρ or greater than ρ is an example of a locally consistent faulty clock. The cost increase for tolerating locally consistent clock failures and timing processor failures is $nf + 1$ messages instead of $f + 1$ messages for processor crash failures, and a termination time of $(f + 1)\ (\delta + \epsilon)$, instead of $([f/2] + 1)(\delta + \epsilon)$. If tolerance of arbitrary clock failures and arbitrary processor and communication failures detectable by using message authentication methods is desired, then the third protocol of [CASD85] can be used at the following cost: $nf + 1$ messages per broadcast and a termination time of $(f + 1)\ (\delta' + \epsilon')$, where δ' is the upper bound on message transmission and processing delays that includes the time needed for message authentication, and ϵ' is the worst case deviation among clocks where components can suffer authentication detectable failures.

While tolerance of timing, clock or authentication detectable arbitrary failures costs more, one might ask what cost savings are possible by strengthening our failure assumption 3, for example by assuming that out-adapters can suffer only omission - instead of performance-failures (this can be achieved by using out-adapters based on Time Division Multiple Access or Tree Collision Resolution methods [Gal85] which ensure upper bounds on the time needed to insert atomic broadcast messages on non-faulty channels). If adapters can only suffer omission failures, then it is possible to achieve a termination time of $([f/2] + 1)\delta + \epsilon$ by eliminating the (now) unnecessary timeliness checks (7.6, 7.7),and by using the following (slightly changed) lazy forwarding rule. Let c be the highest channel on which a processor p receives a message (T, s, σ, h) with hop count $h \leq k$. If at time $T + h\delta + \epsilon$ on p's clock condition $c < f + 1 - h$ holds, then p must forward $(T, s, \sigma, h + 1)$ on channels $c + 1, \ldots, f + 1 - h$ else, p does not have to forward. Since the resulting protocol is so similar to the one described in Figures 5-8, we do not give its detailed description for brevity reasons.

6 Comparison with previous work

The synchronous atomic broadcast protocol proposed in this paper achieves a significant message overhead reduction over the synchronous atomic broadcast protocols proposed for point-to-point networks earlier [CASD85]: when no failures occur only $f + 1$ messages are sent per broadcast, instead of between n and n^2 messages. The termination time is at least halved: $([f/2] + 1)\delta + \epsilon$ instead of $(f + d)\delta + \epsilon$ when only omission component failures can occur, and $[f/2]\ (\delta + \epsilon)$ instead of $f(\delta + \epsilon) + d\delta + \epsilon$ when adapter performance failures but no timing processor failures can occur, where $d \geq 1$. When processors can suffer timing or authentication detectable arbitrary failures and processor clocks can be faulty, the use of broadcast channels does not help much: the termination time stays the same as in [CASD85] and the number of messages becomes $nf + 1$ instead of between n and n^2.

The comparison between our protocols and the synchronous atomic broadcast protocols proposed by Babaoglu and Drumond [BD85] is made difficult by the use of different system models and assumptions. For example, while Babaoglu and Drumond assume a model based on message rounds and exactly synchronized clocks in which all receiving processors know the time at which a sending processor broadcasts, we do not assume exact synchronization or the existence of any (pre)agreement on the times when messages can be broadcast. Because of the exact clock synchronization assumption of (Babaoglu and Drummond 1985) for example, some of the issues that we had to address explicitly, like the possibility that a message broadcast by a faulty slow processor is accepted by only a subset of correct processors even if no adapters or channels fail, just because of differences between correct processor clocks, did not have to be considered in [BD85].

Another important difference is that we do not assume that channels are *atomic*[1]. The atomic channel assumption of [BD85] states that: for any message m inserted on a channel c either all (in case the channel is non-faulty) or none (in case the channel is faulty) of the processors connected to c through non-faulty adapters receive m. Instead of this assumption on faulty channel behavior we use a strictly weaker assumption: if a channel c fails while it carries a message m any subset (not only the total subset) of correct processors attached through correct adapters to c can miss m. This weaker assumption was needed to model run-time situations observed in common local area networks, such as Ethernet and Token Ring, in which the bits of a message are corrupted while the message transits on the channel. For example, undetected collisions or an improper functioning at run-time of a repeater between

[1] Although Babaoglu and Drumond examine later in their paper the consequences of relaxing the *atomic channel* failure assumption and derive n $\geq \lambda' + \pi$ as being a sufficient condition for correctness in the presence of omission failures, where λ' is the worst case number of in-adapters attached to all channels that might miss a message broadcast on all channels because of channel failures and π is the worst case number of processor failures, their condition is generally false under failure assumptions 2 and 8, which allow λ' and π to be as high as $f(n - 1)$ and f, respectively.

two Ethernet transceivers can cause a degradation of the signal on the Ether so that only a subset of the correct adapters receive a message while the remaining ones discard it because of check-sum errors [MB76]. Similarly, in a Token Ring context, the malfunctioning of any (active) adapter p when it re-transmits a sequence of received bits to a down-stream adapter q can cause q and all the following down-stream adapters to receive (and discard)a corrupted sequence of bits [TMZ88].

These differing system models and assumptions lead to differing termination times. While the stronger assumptions of [BD85] enable the protocols tolerant of omission and authentication detectable arbitrary failures proposed by Babaoglu and Drumond to terminate in a constant number of two rounds (in our model this would probably correspond to a termination time of $2(\delta + \epsilon)$ with $\epsilon = 0$, our weaker assumptions cause our protocols to only achieve termination times proportional to the maximum number of failures to be tolerated. The number of messages per broadcast is also different: while the protocols tolerant of adapter omission and performance failures need $f + 1$ messages in the absence of failures, the omission tolerant protocol of [BD85] sends $n(f + 1)$ messages per broadcast. For timing, authentication detectable arbitrary and clock failures the message overheads are similar: while our protocols send $nf + 1$ messages, the protocol of [BD85] sends $n(f + 1)$ messages.

Acknowledgments

The research presented was partially sponsored by IBM's Systems Integration Division group located in Rockville,Maryland, as part of a FAA sponsored project to build the Advanced Automation System, a new, highly available air traffic control system. A prototype of the system, including one of the protocols presented in this paper, was successfully demonstrated in June 1988. We would like to thank Paul Ezhilchelvan, Ray Strong, and the referees for their criticisms and suggestions on earlier versions of this paper.

References

[BD85] O. Babaoglu and R. Drummond. Streets of Byzantine: Network architectures for fast reliable broadcast. *IEEE Transactions on Software Engineering*, SE-11(6):546–554, Jun 1985.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[Car85] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, Jun 1985.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CF94] F. Cristian and C. Fetzer. Fault-tolerant internal clock synchronization. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, Dana Point, Ca., Oct 1994.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[DSC89] D. Dolev, R. Strong, and F. Cristian. Distinguishing timing failures from clock failures. Technical Report RJ 7150, IBM Research, San Jose, CA, 1989.

[Gal85] R. Gallager. A perspective on multiaccess channels. *IEEE Tr. on Information Theory*, IT-31(2), March 1985.

[Lam89] L. Lamport. The part time parliament. Technical Report TR-49, DEC-Systems Research Center, Palo Alto, September 1989.

[MB76] R. Metcalfe and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, Jul 1976.

[MSMA90] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, Jan 1990.

[SSCA87] R. Strong, D. Skeen, F. Cristian, and H. Aghili. Handshake protocols. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 521–528, Berlin, Sep 1987.

[TMZ88] J. Tusch, H. Meyr, and E. Zurfluh. Error handling performance of a token ring local area network. In *13th IEEE Int. Conf. on Local Area Networks*, Minneapolis, Minessota, 1988.

[VRB89] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

A Appendix

We prove below a stronger version of the lemma used to justify the correctness of the general lazy forwarding rule.

LEMMA. Let c_h be the highest channel number on which a correct processor p_h receives a message T, s, σ, h with hop count $h \leq k$. Let p_i and c_i, $i = 1, \ldots, h$ - 1, be sequences of processors and channels defined recursively as follows: p_{h-1} forwarded the message (T,s,σ,h) received by p_h after receiving (T,s, σ, h - 1) on highest channel c_{h-1}, ..., processor p_1 forwarded the message (T, s, σ, 2) received by p_2 after receiving from $s = p_0$ the message (T, s, σ, 1) on highest channel c_1. Let A_1 be the set of components $\{p_{i-1}$, channel c_{i+1}, p_{i-1}'s out-adapter to $c_i + 1$, p_i's in-adapter to $c_i + 1$ $\}$, and denote C_h the union of the sets A_i, $i = 1, \ldots, h$ - 1. If at time $T_h = T + h(\delta + \epsilon)$ on the clock of processor p_h, $c_h \leq f + 1$ - h, then at least h components in the set C_h are faulty.

The proof of the lemma is by induction on h.

Consider the reception of a timely message (T, s, σ, h) with $h = 1$ by p_1 on highest channel c_1. If at time $T_1 = T + (\delta + \epsilon)$ on p_1's clock $c_1 \leq f + 1$ - $h = f$, then the sender $s = p_0$ did not enqueue a message (T, s, σ, 1) on the out-adapter to channel $c_1 + 1 = f + 1$ because of a crash, or this out-adapter suffered a performance failure, or the channel $c_1 + 1$ suffered an omission failure, or the in-adapter between channel $c_1 + 1$ and p_1 suffered an omission failure. Thus, at least one component in the set C_1 is faulty.

Assume now that the lemma is true for $h = i$, $i \geq 1$, and consider the case $h = i + 1$. By induction hypothesis, some processor p_i forwarded the message (T, s, σ, $i + 1$) after it received (T, s, σ, i) on highest channel c_i. By the lazy forwarding rule, processor p_i had to forward the message (T,s, σ, $i + 1$) on channels $c_i + 1$, ..., $f + 1$ - i. If at time $T_{i+1} = T + (i + 1)(\delta + \epsilon)$ on p_{i+1}'s clock, the highest channel c_{i+1} on which p_{i+1} received a message (T, s, σ, $i + 1$) is such that $c_{i+1} \leq f + 1$ -($i + 1$) = f - i, it follows that p_{i+1} did not receive in time the message (T, s, σ, $i + 1$) that p_i should have sent on channel $f + 1$ - i. This could only happen if at least a component in the set $A_{i+1} = \{$ p_i, p_i's out-adapter to channel $f + 1$ - i, channel $f + 1$ - i, p_{i+1}'s in-adapter from channel $f + 1$ - i $\}$ is faulty. Since $c_i < c_{i+1}$, the intersection between the component sets C_i and A_{i+1} is empty. Since by induction hypothesis, the set C_i contains at least i faulty components, it follows that the set $C_{i+1} = C_i \cup A_{i+1}$ contains at least $i + 1$ faulty components. The lemma is thus true for any integer h.