Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement

Flaviu Cristian, Houtan Aghili, Ray Strong, Danny Dolev
IBM Alamaden Research Center*
San Jose, CA 95120-6099

March 29, 1994

Abstract

In distributed systems subject to random communication delays and component failures, atomic broadcast can be used to implement the abstraction of synchronous replicated storage, a distributed storage that displays the same contents at every correct processor as of any clock time. This paper presents a systematic derivation of a family of atomic broadcast protocols that are tolerant of increasingly general failure classes: omission failures, timing failures, and authentication-detectable Byzantine failures. The protocols work for arbitrary point-to-point network topologies, and can tolerate any number of link and process failures up to network partitioning. After proving their correctness, we also prove two lower bounds that show that the protocols provide in many cases the best possible termination times.

Keywords and phrases: Atomic Broadcast, Byzantine Agreement, Computer Network, Correctnesss, Distributed System, Failure Classification, Fault-Tolerance, Lower Bound, Real-Time System, Reliability, Replicated Data.

*Flaviu Cristian is now with the University of California, San Diego, Houtan Aghili is with the IBM TJ Watson Research Center and Danny Dolev is with the Hebrew University

1 Introduction

Random communication delays and failures prevent distributed processes from having the knowledge of global system states that shared storage provides to the processes of a centralized system. The absence of such knwledge is one of the main reasons why distributed programming is so difficult. The objective of this paper is to discuss broadcast protocols that enable the correct processes of a distributed system to attain consistent (albeit slightly delayed) knowledge of the system state, despite failures and random communication delays. Programming distributed processes that share such consistent views of the system state then becomes similar to programming the processes of a centralized system.

The idea is to synchronize processor clocks, replicate global system state information at several physical processors, and use atomic broadcast for disseminating global state updates to these processors, so that all correct processors have identical views of the global state at identical clock times. An *atomic broadcast* protocol is a protocol which, for some time constant Δ called the broadcast termination time, possesses the following properties: (1) *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 each correct processor at time U on its clock, (2) *order*: all updates delivered by correct processors are delivered in the same order by each correct processor, and (3) *termination*: every update whose broadcast is initiated by a correct processor at time T on its clock is delivered at all correct processors at time $T + \Delta$ on their clocks.

Because of its properties, atomic broadcast can be used to implement the abstraction of *synchronous replicated storage*: a distributed, resilient storage that displays, at any clock time, the same contents at every correct physical processor and that requires Δ time units to complete replicated updates. Indeed, if all updates to synchronous replicated storage are broadcast atomically, the atomicity property ensures that every update is either applied at all correct processors or by none of them and the order property ensures that all updates are applied in the same order at all correct processors. Therefore, if the replicas are initially consistent, they will stay consistent. The termination property ensures that every update broadcast by a correctly functioning processor is applied to all correct replicas Δ clock time units later. If the synchronous replicated storage is used to record global state information, this means that processes running at each correct physical processor can perceive, at any time, the global system state that existed Δ clock time units earlier. Such a storage is therefore very similar to a shared storage, except that it does not represent a single point of failure.

The use of synchronous replicated storage can simplify the programming of distributed processes since it relieves a programmer from the burden of coping with the inconsistency among local knowledge states that can result from random communication delays or faulty processors and links. It is relatively straightforward to adapt known concurrent programming paradigms for shared storage environments to distributed environments that provide the abstraction of a synchronous replicated storage. Several examples of such adaptations are given in [L]. Within the Highly Available System project[1] at the Almaden Research Center, atomic broadcast was designed for updating replicated system directories and reaching agreement on the failure and recovery of system components [Cr], [GS]. In the HAS system prototype, membership information and service directories are maintained as synchronous replicated storage.

Much of the previous work on atomic broadcast has been performed within the Byzantine Generals framework [LSP] (see [F],[SD] for surveys of this work). Typical models within this framework have assumed guaranteed communication in a completely connected network of perfectly synchronized processors. They assume that communication takes place in synchronous rounds of information exchange, where a round was defined as the time interval needed by an arbitrary processor to receive and process all messages sent by all processors in a previous round. In contrast to these perfectly synchronized rounds models, this paper considers networks of arbitrary topology subject to link as well as processor failures. Immediate response to a message is allowed rather than forcing a processor to wait for the end of a round. Clocks are assumed to be only approximately synchronized and we consider a variety of component failure behaviors that are likely to occur in practice and can be tolerated at a cost less than that required to tolerate the worst case (Byzantine) failures.

We classify failures observable in distributed systems into several nested classes, so that the complexity of a fault-tolerant protocol increases with the size of the class of failures it tolerates. We derive a new family of atomic broadcast protocols ranging from a fairly simple protocol that tolerates omission failures to a rather sophisticated protocol that tolerates authentication-detectable Byzantine failures, we prove the correctness of each protocol in the family, and we discuss their performance. We also prove two lower bounds on the termination times of atomic broadcast protocols tolerant of omission and authentication-detectable Byzantine failures. One objective in writing the paper was to structure it so as to allow a reader who is not interested in the technicalities inherent in correctness and lower bound proofs to achieve a reasonable understanding of our protocols without reading the proofs in Sections 5.1, 6.1, 7.1, and 9.

2 Failure Classification

We classify failures with respect to a decomposition of a distributed system into processors and communications links. These components are specified to produce output in response to the occurrence of certain specified input events, such as service request arrivals or the

[1] 1981-1985

passage of time. For example, a link connecting processor s to processor r is specified to deliver a message to r within a certain number of time units whenever s so requests, and a processor p equipped with a timer can be specified to output messages on all its adjacent links every n time units. A component specification prescribes what output should be produced in response to any sequence of input events as well as the real-time interval within which this output should occur (for a more formal definition, see Section 9).

A system component is *correct* if, its response to inputs is consistent with its specification. A component *failure* occurs when a component does not behave in the manner specified. An *omission* failure occurs when, in response to a sequence of inputs, a component never gives the specified output. A *timing* failure occurs when the component gives the specified output too early, too late, or never. A *Byzantine* failure [LSP] occurs when the component does not behave in the manner specified: either no output occurs, or the output is outside the real-time interval specified, or some output different from the one specified occurs. An important subclass of Byzantine failures are those for which any resulting corruption of messages relayed by components such as processors and links is detectable by using a message authentication protocol. We call failures in this class *authentication-detectable* Byzantine failures (cf. [LSP]). Error detecting codes [PW] and public-key cryptosystems based on digital signatures [RSA] are two examples of authentication techniques which can ensure that both unintentional and intentional message corruptions are detected with very high probability. For the rest of the paper we will assume the existence of a fixed authentication protocol (it will be further described below). The class of authentication-detectable failures is defined with respect to this protocol.

A processor crash, a link breakdown, a processor that occasionally does not forward a message that it should, and a link that occasionally loses messages, are examples of omission failures. An excessive message transmission or processing delay due to a processor or network overload is an example of a *late* (or *performance*) timing failure. When some coordinated action is taken by a processor too soon (perhaps because of a timer that runs too fast), we talk of an *early* timing failure. A message alteration by a processor or a link (because of a random fault) is an example of a Byzantine failure. If the authentication protocol employed enables the receiver of the message to detect the alteration we have an authentication-detectable Byzantine failure. If the message alteration is so ingenious that the authentication protocol fails to detect the forgery, we have an example of a non-authentication-detectable Byzantine failure.

Crash failures are a proper subclass of omission failures (a *crash* failure occurs when after a first omission to give output a component systematically omits to respond to all subsequent input events), omission failures are a proper subclass of timing failures (a component which suffers an omission failure can be understood as having an infinite response time), timing failures are a proper subclass of authentication-detectable Byzantine failures (no messages that are output are corrupted), and authentication-detectable Byzantine failures are a proper subclass of the class of all possible failures, the Byzantine failures. The nested nature of the failure classes defined above makes it easy to compare "the power" of fault-tolerant protocols. If A and B are two protocols that implement the same service (e.g. atomic broadcast or clock synchronization) and A tolerates only a proper subclass A' of the class of failures B' that B tolerates, A is less fault-tolerant than B. Our failure classification was chosen so that the complexity of B is greater than that of A whenever the class of failures A' tolerated by A is a proper subclass of the class of failures B' tolerated by B. Thus, the larger the class of failures that a protocol tolerates, the more expensive the protocol is.

Observe that a failure cannot be classified without reference to a component specification. In particular, if one component is made up of others, then a failure of one type in one of its constituent components can lead to a failure of another type in the containing component. For example, a clock that displays the same "time" is an example of a crash failure. If that clock is part of a processor that is specified to associate different timestamps with different synchronous replicated storage updates, then the processor may be classed as experiencing a Byzantine failure. In our decomposition of a distributed system into processors and links, neither type of component is considered part of the other. Also, when considering output behavior, we do not decompose messages, so a message is either correct or incorrect, as a whole. With these conventions we can classify failures unambiguously. We are not concerned with tolerating or handling the failures experienced by such sub-components as clocks directly. We discuss fault tolerance in terms of the survival and correct functioning of processors that meet their specifications in an environment in which some other processors and some links may not meet theirs (usually because they contain faulty sub-components). Thus when we speak of tolerating omission failures, we mean tolerating omission failures on the part of other processors or links, not tolerating omission failures on the part of sub-components like timers or clocks that might cause much worse behavior on the part of their containing processors.

3 Assumptions

We consider a system of *processes* that maintain synchronous replicated storage. Processes disseminate updates to storage replicas by using the atomic broadcast service implemented by n distributed *processors*. Some pairs of processors can communicate through point-to-point *links*. We do not assume that links exist between all pairs of processors. We use the symbol G to represent the communications network of processors and links. We let n be the number of processors and m be the number of links in G. We call *neighbors* those processors that share a link. A synchronous replicated storage manager can ask a processor to atomically broadcast an update σ by invoking a *broadcast*(σ) command. To send a message m containing the update on an adjacent link l, a processor p invokes a *send*(m) *on* l command. The link is assumed to contain any buffers and queueing involved in message transmission and receipt. To receive a message containing some update from a link, a processor must invoke a *receive*(m) *from* i command. The output parameters of this command are the message m received as well as the identity i of the link on which m was received. Updates received in such messages are *delivered* to the higher level synchronous storage managers.

We make the following assumptions.

1. All processor names in G are distinct and there is a total order on processor names.
2. Let F be a set of processors and links that experience failures during an execution of an atomic broadcast protocol, and let G-F be the surviving network consisting of the remaining correct processors from G and the remaining correct links that connect them to each other. We assume that G-F is connected. (When the surviving network is partitioned into disconnected subnetworks, our protocols can no longer guarantee atomicity. An alternative view is that our protocols work on the connected components of the network.)
3. Each processor has access to a clock. We denote by C_p the clock of processor p and use $C_p(t)$ to denote p's local clock time at real time t. In writing time values, we adopt the convention of using capital letters for writing clock times and lower case letters for real times. We assume that the clocks of correct processors are monotone increasing functions of real time and the resolution of processor clocks is fine enough, so that separate clock readings yield different values (this will ensure that no correct processor issues the same timestamp twice). For simplicity, we assume that the lifetime of the system is bounded so that clocks don't wrap around; however, this assumption is not necessary. As an alternative we could have defined an ordering on the timestamps issued by cyclic clocks that would be sufficient to prove the correctness of our algorithms under the assumption that faulty processors do not live longer than half the wrap around time (this is the way the order relation on timestamps is implemented in the system prototype described in [GS]). We also assume that the clocks of correct processors are approximately synchronized: for any correct processors p and q, and for any real time t, clocks are within a maximum deviation ϵ

$$|C_p(t) - C_q(t)| < \epsilon$$

   and are within a linear enveloppe of real time. (Diffusion based clock synchronization protocols tolerant of omission, performance, and authentication-detectable Byzantine failures that satisfy these requirements are presented in [CAS,DHSS]; for a survey see [Sc].)
4. Processors run under the control of a real time operating system which provides multi-tasking. To schedule a task A with input parameters B at local time T, the operating system provides a "*schedule* A(B) *at* T" command. 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.

5. For the message types used in our protocols, transmission and processing delays (as measured on any correct processor's clock) are bounded by a constant δ. This assumption can be stated formally as follows. Let p and q be two correct processors linked by a correct link l and let r be any correct processor. If u is the real time at which p invokes a *send*(m) command on l and v is the real time at which q finishes receiving and processing m, then

$$0 < C_r(v) - C_r(u) \leq \delta.$$

   The δ upper bound includes the time spent by m in the message queues or buffers of l, the time needed to transmit the message on the link l, and the time needed by q to receive and process m. Since in practice the CPU time required for processing a message m by a processor task is negligible compared to the queueing delays that affect m while in transit between different processors, we make the simplifying assumption that message processing time is zero. The δ upper bound also accounts for imprecisions in measuring real time delays which result from clock drift or the need to periodically adjust clocks to keep them synchronized. The magnitude of this constant reflects the worst case load (maximum number of events per time unit) a system is specified to handle. Note that in order to satisfy this assumption, a correct processor must fairly allocate processing time to each link so that it is not possible for faulty processors to swamp the network with so many messages that messages from correct processors never get through to other correct processors. Moreover, neither correct nor faulty processors can generate so many updates so rapidly that any processor or link becomes overloaded and cannot meet its specifications.

6. At any component of the system the number of send, receive, update, and deliver events that take place during any finite amount of time is finite. (This assumption is needed so that we can argue by induction on sequences of events.) Moreover, we assume a fixed upper bound on the number of updates generated at any processor per unit time. (This assumption corresponds to a maximum specified load that our system must handle.)

4 Information Diffusion

We consider three properly nested failure classes: (1) omission failures, (2) timing failures, and (3) authentication-detectable Byzantine failures. For each one of these classes, we present an atomic broadcast protocol that tolerates up to π faulty processors and up to λ faulty links, where π and λ are arbitrary nonnegative integers. Note that π is the assumed maximum number of processors that may suffer failures in any run of our protocol. The number of events that could be considered failures at a faulty processor is not bounded. Likewise, λ is the assumed maximum number of links that may fail in any run of the protocol. The number of events (e.g. lost messages) that could be considered failures at a faulty link is not bounded. The termination time Δ of each protocol is computed as a function of the failure class tolerated, of the π and λ parameters, of the known constants δ and ϵ, and of the largest diameter d of a surviving communication network G-F, for all possible subnetworks F containing up to π processors and λ links (the *diameter* of G-F is the longest distance between any two processors in G-F).

All protocols are based on a common communication technique called *information diffusion*: (1) when a correct processor learns new information, it propagates the information to its neighbors by sending messages to them, and (2) if a correct neighbor does not already know that piece of information, it in turn propagates the information to its neighbors by sending them messages. This ensures that, in the absence of network partitions, information diffuses throughout the network to all correct processors. This technique is called propagation of information and characterized relatively abstractly in a 1983 paper by Segall [SE]. However, the concept of diffusion or "flooding" has been used in distributed systems work at least since the early seventies, usually without reference to a particular source.

Information diffusion is a communication technique, that is, a method for conveying information among processors. What is conveyed is not a message (i.e. a sequence of bits), but rather a proposition. For now, we only give motivating examples of terms such as "proposition" and "learn." These terms are defined precisely in the formal sections dealing with correctness proofs. A processor p can convey to a neighbor q a proposition such as "processor s has initiated the atomic broadcast of an update σ at time T on its clock" by sending q a message (T,s,σ) containing the arguments of the proposition. Processors learn the truth of the propositions conveyed by a distributed protocol by receiving messages or by observing the passage of time on their local clocks. For example, to learn the proposition "s has initiated the broadcast of σ at time T on its clock" a processor q has to receive a new message (T,s,σ). As another example, consider a processor p specified to initiate atomic broadcasts at times T_1, T_2, ... known to another processor q. If q receives broadcast T_i by $T_i + \Delta$, but does not receive broadcast T_{i+1} by $T_{i+1} + \Delta$, q learns the proposition "p is not correct at time T_{i+1} on its clock" [Cr]. For each correctness proof, we will use the term "learn" for an action that can only happen once at a processor for any particular proposition. When we speak of a "known constant", like known broadcast termination time Δ, we mean a constant that is recorded in the main storage of each processor.

The specific meanings of "proposition" and "learn" will vary with protocols and failure models. However, the correctness proofs of all the protocols presented in this paper are based on a theorem we call the diffusion induction principle, that treats these terms as primitives. For purposes of this principle, "propositions" are primitive objects from some fixed set Prop and "learn" is a primitive relation on the set Processor $\times$ Prop $\times$ Real-Time, expressed informally by saying, "processor p learns proposition ϕ at real time t." We say a message m *conveys* proposition ϕ to processor p if, whenever m is received by p at time t, there is a time $u \leq t$ such that p learns ϕ at u. We say that a proposition ψ *propagates* (among neighbors) if, for every correct pair of neighbors p and q linked by a correct link, and for every real time u, if p learns ψ at u, then there is a real time v such that q learns ψ at v and, $C_r(v) - C_r(u) \leq \delta$, where r is any correct processor. Note that, to ensure that a proposition ψ propagates it is sufficient (but not necessary) for p to send a message conveying ψ to q at u. This ensures that q learns ψ within *delta* clock time units from the moment p learned ψ, unless q learned ψ earlier, for example by receiving another message conveying ψ from another processor. A proposition ψ *diffuses* (in G) if for any correct processors p, q, and r, and for any real time u, if p learns ψ at u, then there is a real time w such that q learns ψ at w and $C_r(w) - C_r(u) \leq d\delta$, where d is the largest diameter of a surviving communication network G-F, for all possible subnetworks F containing up to π faulty processors and λ faulty links. From our assumptions we can now easily prove the **Diffusion Induction Principle**.

Theorem: If proposition ψ propagates among neighbors, then ψ diffuses in G.

Proof: Assume ψ propagates to neighbors. Let r be any correct processor in G. Let d be the maximum possible diameter of a surviving subnetwork of G. We say ψ *diffuses from p to q in subnetwork G'* if, when p learns ψ at u, then there is a real time w such that q learns ψ at w and $C_r(w) - C_r(u) \leq d(p,q)\delta$, where d(p,q) is the distance from p to q in G'. It suffices to prove that, for any surviving network G' of correct components, and for any processors p and q of G', ψ diffuses from p to q. We prove this by induction on the distance d(p,q) in G'. For d(p,q)=1, diffusion is immediate from our hypothesis that ψ propagates. We now suppose that ψ diffuses from p to q for all processors p and q in G' with d(p,q) < k, where k > 1. Let p and q be correct processors in G' such that the distance d(p,q) between p and q is k. The hypothesis k>1 implies the existence of an intermediate correct processor s in G' such that d(p,q)=d(p,s)+d(s,q), where the distances d(p,s) and d(s,q) are both smaller than k. If p learns ψ at u, then, by the induction hypothesis, there is a time v such that s learns ψ at v and $C_r(v) - C_r(u) \leq d(p,s)\delta$. Again, by the induction hypothesis, there is a time w such that q learns ψ at w and $C_r(w) - C_r(v) \leq d(s,q)\delta$. It follows that $C_r(w) - C_r(u) \leq d(p,q)\delta$. $\Box$

We call the time dδ the *diffusion time* of the surviving network G-F in the presence of at most π processor failures and λ link failures. We use the Diffusion Induction Principle to infer diffusion from propagation in our proofs of protocol correctness. The principle is independent of the choice of the failure class to be tolerated; however, the definitions of "propagates" and "diffuses" depend on the definition of "learn," which will vary depending on the failure class considered. While this principle captures informal reasoning that has been used for years, we believe our formulation of the principle is novel. Note that the correctness of the principle depends on bounds on the clock time measured at *any* correct processor, not simply at one of the participants in a message transmission or receipt.

5 First Protocol: Tolerance of Omission Failures

Each message sent according to our first protocol carries its initiation time (or *timestamp*) T, the name of the source processor s, and a replicated storage update σ. Each atomic broadcast is uniquely identified by its timestamp T and its initiator's name s (by Assumptions 1 and 3). As messages are received by a processor, they are stored in a *history* log H local to the processor until delivery to the local synchronous replicated storage manager. The order property required of atomic broadcasts is achieved by letting each processor deliver the updates it receives in the order of their timestamp, by ordering the delivery of updates with identical timestamps in increasing order of their initiator's name, and by ensuring that no correct processor begins the delivery of updates with timestamp T before it is certain that it has received all updates with timestamp at most T that it may ever have to deliver (to satisfy the atomicity requirement).

Note that any message that is received in the omission failure only context must have been sent correctly and must be deliverable. For omission failures, the local time by which a processor is certain it has received copies of each message timestamped T that could have been received (and hence, delivered) by some correct processor is $T + \pi\delta + d\delta + \epsilon$. We call this clock time the *delivery deadline* for updates with timestamp T. The intuition behind this deadline is as follows. The term $\pi\delta$ is the worst case delay between the initiation of a broadcast (T,s,σ) and the moment a first correct processor r learns of that broadcast. It corresponds to the case when the broadcast source s is a faulty processor and between s and r there is a path of π faulty processors, each of which forwards just one message (T,s,σ) on one outgoing link where each of these messages experiences a delay of δ clock time units. The term $d\delta$ is the time sufficient for r to diffuse information about the broadcast (T,s,σ) to any correct processor p in the surviving network. The last term ensures that any update accepted for delivery by a correct processor q whose clock is in advance of the sender's clock is also accepted by a correct processor p whose clock is behind the sender's clock. We assume all processors know the protocol termination time $\Delta_o \equiv \pi\delta + d\delta + \epsilon$.

To keep the number of messages needed for diffusing an update finite, each processor p that receives a message (T,s,σ) *relays* the message (to all its neighbors except the one that sent the message) only if it receives (T,s,σ) *for the first time.* If p inserts all received messages in its local history H (and never removes them), p can easily test whether a newly arrived message m was or was not seen before by evaluating the test $m \in H$. We call this test the "deja vu" acceptance test for the message m. The main drawback of the "deja vu" solution described above is that it causes local histories to grow arbitrarily. To keep the length of H bounded, a history garbage collection rule is needed. A possible solution would be to remove from H a message (T,s,σ) as soon as the deadline $T + \Delta_o$ for delivering σ passes on the local clock. 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 deadline $T+\Delta_o$ has passed on p's clock. Such duplicates would then pass the "deja vu" acceptance test and would be inserted again in the history of p. Since such "residual" duplicates will never be delivered (see Assumption 4), they can cause p's history to grow without bound.

The reader might at this point wonder how is it possible that, a message timestamped T could arrive at a correct processor p after its delivery deadline $T+\Delta_o$ has passed, when this deadline was precisely computed to ensure that p receives before $T+\Delta_o$ a copy of each message timestamped T that it will ever have to deliver? The following scenario shows that this is possible. Consider a fully connected network of three processors p, q, r and a protocol tolerant of one omission failure, with termination time $2\delta+\epsilon$. Consider that p initiates the broadcast of an update σ at local time T, the message (T,p,σ) from p to r is lost (due to an omission failure), and the (T,p,σ) messages from p to q, from q to r, and from r to p relayed according to the above "deja vu" rule all take δ clock time units. If $\epsilon<\delta$ (this is possible for probabilistic clock synchronization algorithms [Cri]) then p receives a message (T,p,σ) from r at local time $T+3\delta$ after the delivery deadline $T+2\delta+\epsilon$ has passed. To prevent such residual messages from accumulating in local histories, we introduce a "late message" acceptance test. This test discards a message (T,s,σ) if it arrives at a local time U past the delivery deadline $T+\Delta_o$, i.e. if $U\geq T+\Delta_o$. The "deja vu" and "late message" acceptance tests ensure together that updates are broadcast by using a finite number of messages and that local histories stay bounded (by Assumption 6, processors broadcast only a bounded number of updates per time unit).

A detailed description of our first atomic broadcast protocol is given in Figures 1, 2, and 3. Each processor runs three concurrent tasks: a Start broadcast task (Figure 1) that initiates an atomic broadcast, a Relay task (Figure 2) that forwards atomic broadcast messages to neighbors, and a Delivery task (Figure 3) that delivers broadcast updates to the synchronous replicated storage layer. All tasks of a processor have access to a constant L of type Set-of-Link containing the identity of all links adjacent to the processor. All tasks of all processors have access to the termination time constant Δ_o defined earlier. In what follows we refer to line j of Figure i as (i.j).

```
1  task Start;
2  var T: Time; σ: Update; s: Processor; l: Link;
3  cycle wait-for-broadcast(σ); T ← clock;
4        for all l ∈ L do send(T,myid,σ) on l od;
5        add (T,s,σ) to H;
6        schedule Delivery(T) at $T + \Delta_o$;
7  endcycle;
```

Figure 1. Start Task of the first protocol

A process triggers the broadcast of an update σ by invoking a *broadcast*(σ) command on its underlying processor. This will activate the Start task at the matching *wait-for-broadcast* entry point with σ as input (1.3). The broadcast of σ is identified by the local time T at which σ is received (1.3) and the identity of the sending processor, obtained by invoking the function "myid" (1.4). This function returns different processor identifiers when invoked by distinct processors (Assumption 1). The broadcast of σ is initiated by invoking *send* commands on all outgoing links L (1.4). We do not assume that the execution of the command in line (1.4) is atomic with respect to failures: a processor failure can prevent messages from being sent on some links. The fact that the broadcast of σ has been initiated is then recorded in a history variable H, a set of triples shared by all broadcast layer tasks:

var H: Time $\times$ Processor $\times$ Update.

We assume that H is initialized to the empty set {} at processor start. Once the history H is updated (1.5), the Delivery task is scheduled to execute with input parameter T at local clock time $T + \Delta_o$ to deliver the update (1.6).

```
1  task Relay;
2  var U,T: Time; $\sigma$: Update; s: Processor; i,l: Link;
3  cycle receive(T,s,$\sigma$) from i; U $\leftarrow$ clock;
4        if U $\geq T + \Delta_o$ then "late message" iterate fi;
5        if (T,s,$\sigma$) is in H then "deja vu" iterate fi;
6        for all l $\in$ L-{i} do send(T,s,$\sigma$) on l od;
7        add (T,s,$\sigma$) to H;
8        schedule Delivery(T) at $T + \Delta_o$;
9  endcycle;
```

Figure 2. Relay Task of the first protocol

The Relay task uses the command *receive* to receive messages formatted as (T,s,σ) from neighbors (2.3). After a message is received, the output parameter i contains the identity of the incoming link over which the message arrived. If the message is a duplicate of a message that was already received (2.5) or delivered (2.4) then it is discarded (the meaning of the *iterate* command is to skip the execution of the rest of the loop body and begin a new iteration). A message is *accepted* if it passes the late message (2.4) and deja vu (2.5) tests of the Relay task. If (T,s,σ) is accepted, then it is relayed on all outgoing links except i (2.6), it is inserted in the history variable (2.7), and the Delivery task is scheduled to execute with input parameter T at local time $T + \Delta_o$ to deliver the received update (2.8).

```
1  task Delivery(T:Time);
2  var val: Processor $\times$ Update;
3  val $\leftarrow$ {(s,$\sigma$) | (T,s,$\sigma$) $\in$H};
4  sort val by processor name lexicographically;
```

```
5  for all (s,$\sigma$) $\in$ val in order do deliver($\sigma$) od;
6  delete all triples with first element T from H;
```

Figure 3. Delivery Task

The Delivery task (Figure 3) starts at clock time $T + \Delta_o$ to deliver updates timestamped T in increasing order of their sender's identifier ((3.3)-(3.5)) and to delete all information about these broadcasts from the local history H (3.6).

5.1 Proof of Correctness for the First Protocol

We use the diffusion induction principle to prove the correctness of our first protocol (denoted $\mathcal{O}$ in this proof) under the assumption that during any protocol execution there can be at most π processors that suffer omission failures and at most λ links that suffer omission failures. The propositions diffused by $\mathcal{O}$ are of the form $\psi \equiv$ "processor s broadcasts σ at local time T." We say correct processor p *learns* ψ at a real time t if either (a) p=s, p initiates the diffusion of messages (T,s,σ) at t, and $C_s(t) = T$, or (b) $p \neq s$ and p receives at t a message (T,s,σ) for the first time. We denote by $\mathcal{O}$' the protocol with infinite local history obtained by removing from $\mathcal{O}$ the "late message" acceptance test (2.4) and the local history garbage collection (3.6).

Lemma 1: If all processors follow protocol $\mathcal{O}$', then ψ propagates among neighbors.

Proof: Assume that correct processor p learns ψ at real time t and let q be a correct neighbor of p linked by a correct link to p. If p=s, then p sends (1.4) a message (T,s,σ) to q at t. By Assumption 5, there is a real time u at which the message is received by q, such that for any correct processor r: $C_r(u) - C_r(t) \leq \delta$. If (T,s,$\sigma$) is not in the history of q, q learns ψ at u and inserts the message in its history. If (T,s,σ) is in q's history at u, then q must have learned ψ at an earlier real time $v \leq u$. By the monotonicity of the C_r clock, q learned ψ at a clock time $C_r(v) \leq C_r(u)$, so the inequality $C_r(v) - C_r(t) \leq \delta$ holds in this case too. If $p \neq s$, let t be the real time at which p learns ψ by receiving a message (T,s,σ) for the first time from some neighbor q'. If q=q', there is a time u<t such that q learned ψ at u and $C_r(u) - C_r(t) < 0$, so $C_r(u) - C_r(t) \leq \delta$. If $q \neq q'$, by an argument analogous to that for the case p=s before, we can show that q learns ψ within δ clock time units either because q receives the message (T,s,σ) sent by p at t for the first time or because q learned ψ earlier. □

Lemma 2: If all processors follow protocol $\mathcal{O}$' and a correct processor inserts a message (T,s,σ) in its history, then each correct processor inserts (T,s,σ) in its history before local time $T + \Delta_o$.

Proof: Let u be the earliest real time at which a correct processor p inserts (T,s,σ) in its history. If p=s, then by diffusion induction, each correct processor q inserts (T,s,σ) in its history by a real time v such that $C_q(v) \leq C_q(u)+d\delta$. Since $C_q(u) < C_s(u)+\epsilon$, it follows that the local time by which q inserts (T,s,σ) in its history is $C_q(v) < C_s(u) + d\delta + \epsilon \leq T + \Delta_o$. Consider now the other case $p \neq s$. In this case p receives a message (T,s,σ) for the first time at real time u. Since only omission processor and link failures are possible, the messages supposed to be sent by a processor are either received on time (as if the processor were correct) or are never received (for example, because the processor has crashed before sending them or a link suffers an omission failure). Let t be the real-time at which processor s initiates the broadcast of update σ, $C_s(t) = T$. If correct processor p learns ψ at real time u by processing a message that traversed a path of h hops (i.e. h links) after it was sent by s, then it is easy to prove by induction on h (in a manner similar to that illustrated for the proof of the diffusion induction principle) that, for any processor r: $C_r(u) - C_r(t) \leq h\delta$. By hypothesis, there can be at most π faulty processors, so the longest possible acyclic path (i.e. sequence of processors and links without repetition) that a message originating at processor s can traverse before being accepted by a first correct processor contains π hops. Since u is the earliest real time at which some correct processor p learns ψ, it follows that $C_r(u) - C_r(t) \leq \pi\delta$, for any processor r. By diffusion induction every correct processor q will insert (T,s,σ) in its history at a real time v such that, $C_r(v) - C_r(u) \leq d\delta$, for any processor r. From the above, it follows in particular that $C_q(v) \leq C_q(t) + \pi\delta + d\delta$. Since $C_q(t) < C_s(t) + \epsilon$, we finally have $C_q(v) < C_s(t) + \pi\delta + d\delta + \epsilon = T + \Delta_o$. □

Lemma 3: If all processors follow protocol $\mathcal{O}$ and a correct processor inserts a message (T,s,σ) in its history, then each correct processor inserts (T,s,σ) in its history before local time $T + \Delta_o$. The history H of any processor remains bounded.

Proof: We first prove that the protocols $\mathcal{O}$ and $\mathcal{O}$' are equivalent, in the sense that a message (T,s,σ) is accepted by a correct processor p which follows $\mathcal{O}$ if and only if the same message is accepted by p following $\mathcal{O}$'. Clearly, each message inserted in the history by a processor following $\mathcal{O}$ will also be inserted if p were following $\mathcal{O}$', since the set of messages that pass the two acceptance tests "late message" and "deja vu" is included in the set of messages that pass the "deja vu" test. Assume that a correct processor p follows $\mathcal{O}$' and inserts a message (T,s,σ) in its history. If p=s, then p also inserts the message in its history following $\mathcal{O}$, since the $\mathcal{O}$ and $\mathcal{O}$' Start tasks are identical. Consider now that $p \neq s$. By Lemma 2, the "late message" test (2.4) at p will evaluate to false, so (T,s,σ) will pass the test and will be inserted in the history if p followed $\mathcal{O}$. The first part of Lemma 3 now follows from Lemma 2.

Because of the "late message" acceptance test, and the assumption that only omission failures can occur, the history of each correct processor contains at any local time T only messages with timestamps in the range $[T - \Delta_o, T + \epsilon)$. Since the number of processors is bounded, and each processor can broadcast only a bounded number of messages in any bounded time interval, it follows that the total number of messages which can exist at any point in time in H is bounded. □

Theorem 1: The first protocol possesses the termination, atomicity, and order properties.

Proof: If an update σ is received by the Start task of correct processor s at local time T, then by Lemma 3, every correct processor adds (T,s,σ) to its (bounded) history H before local time $T + \Delta_o$. Moreover, examination of the Start and Relay tasks shows that a correct processor cannot add (T,s,σ) to H without scheduling its Delivery task for local time $T + \Delta_o$ with input T. Lemma 3 also implies that, whether the initiator is correct or not, if any correct processor adds (T,s,σ) to its history, then all do. Thus Lemma 3 implies both termination and atomicity for the protocol. Since (by Assumption 4) the Delivery task delivers updates with different timestamps in timestamp order, and orders the delivery of updates with same timestamp in increasing order of their originator's name, our protocol also satisfies the order property. □

5.2 The First Protocol is Not Tolerant of Timing Failures

We construct a counter-example showing that the occurrence of a timing failure can lead to a violation of the atomicity property. Consider a totally connected network of four processors s (sender), f (faulty), e (early, correct), and l (late, correct), such that e's clock is z, 0<z< ϵ, time units in advance of l's clock, i.e. when l's clock indicates U, e's clock indicates U+z. Suppose that the Start task of s is interrupted by a crash so that a message (T,s,σ) is sent only to f. Suppose that the faulty processor f, delays forwarding messages (T,s,σ) to the correct processors e and l in such a way that the messages sent by f arrive when e's clock shows $T + \Delta_o + z/2$ and l's clock shows $T + \Delta_o - z/2$. Clearly, σ cannot be delivered at e (2.4), but since the message arrives at l before l's clock shows $T + \Delta_o$, the message is accepted at l, where the update σ is delivered at local time $T + \Delta_o$. The atomicity requirement is therefore violated.

6 Second Protocol: Tolerance of Timing Failures

The first protocol is not tolerant of timing failures because there is a fixed clock time interval, independent of the number of faulty processors, during which a message is unconditionally accepted by a correct processor. As illustrated in the counter-example above, this creates a real time "window" during which a message might be "too late" for some (early) correct processors and "in time" for other (late) correct processors. To achieve atomicity in the presence of timing failures, we must ensure that if a first correct processor p accepts a message m, then all other correct neighbors q to which p relays m also accept m. A neighbor q does not know whether the message source p is correct or not. However, if p is correct, q must accept m if the information stored in it tells q that the clock time delay between p and q is at least -ϵ (case when p's clock is very close to being ϵ time units behind q's and the message propagation delay between p and q is very close to being 0) or at most $\delta + \epsilon$ (case when p's clock is very close to being ϵ time units in advance of q's and the message from p to q takes δ time units). To be able to evaluate the time a message spends between two neighbors, we store in the message the number h of hops traversed by it.

A processor will reject messages that have taken longer than $\delta + \epsilon$ clock time units per hop or are more than ϵ clock time units per hop early. The need for a lateness test is motivated by the example in Section 5.1. We need an earliness test because we wish to maintain the property that the history log at any correct processor remains bounded. Otherwise, some faulty processor with a very fast clock might send the updates it has to send much earlier in real-time than it is supposed to, forcing other correct processors to keep them in their history logs for an unbounded amount of time. This type of faulty behavior may not be very common in practice, but it does fit the definition of early timing failure.

Formally, we use the following *timeliness* acceptance test: a correct processor q accepts a message timestamped T with hop count h if it receives it at a local time U such that:

$$\mathrm{T} - \mathrm{h}\,\epsilon < \mathrm{U} < \mathrm{T} + \mathrm{h}(\delta + \epsilon).$$

Since, by hypothesis, there can be at most a path of π faulty processors from a (faulty) sender s to a first correct processor p, and the message accepted must pass the above test at p, it follows that a message can spend at most $\pi(\delta + \epsilon)$ clock time units in the network before being accepted by a first correct processor. From that moment, it needs at most dδ clock time units to reach all other correct processors. Given the ϵ uncertainty on clock synchrony the termination time of the second protocol is therefore: $\Delta_t = \pi(\delta + \epsilon) + d\delta + \epsilon$.

The Start task of the second protocol (Figure 4) is identical to that of the first except for the addition of a hop count to all messages. At origin, this hopcount is initialized to 1 (4.4).

```
1  task Start;
2  var T: Time; $\sigma$: Update; s: Processor; l: Link;
3  cycle wait-for-broadcast($\sigma$); T $\leftarrow$ clock;
4      for all l $\in$ L do send(T,myid,1,$\sigma$) on l od;
5      add (T,s,$\sigma$) to H;
6      schedule Delivery(T) at $T + \Delta_t$;
7  endcycle;
```

Figure 4. Start Task of the second protocol

In addition to the tests used for providing tolerance of omission failures (5.6, 5.7) the Relay task of the second protocol (Figure 5) also contains the timeliness tests discussed above (5.4, 5.5). The hop count h carried by messages is incremented (5.8) every time a message is relayed. The Delivery task of the second protocol is identical to that of the first protocol.

```
1  task Relay;
2  var U,T: Time; $\sigma$: Update; s: Processor; h: Integer; i,l: Link;
3  cycle receive(T,s,h,$\sigma$) from i; U $\leftarrow$ clock;
4      if U > T-h$\epsilon$ then "too ealry" iterate fi;
5      if U < T+h($\delta + \epsilon$) then "too late" iterate fi;
6      if U $\geq T + \Delta_t$ then "late message" iterate fi;
7      if (T,s,$\sigma$) is in H then "deja vu" iterate fi;
8      for all l $\in$ L-i do send(T,s,h+1,$\sigma$) on l od;
9      add (T,s,$\sigma$) to H;
10     schedule Delivery(T) at $T + \Delta_t$;
11 endcycle;
```

Figure 5: Relay Task of the second protocol

6.1 Correctness of the Second Protocol

The propositions ψ diffused by our second protocol (which we denote $\mathcal{T}$ in this proof) have the same form as those diffused by the first protocol: "processor s broadcasts update σ at time T on its clock." We say that correct processor p *learns* ψ at real time t if either (a) s=p, s initiates the broadcast of σ at t, and $C_s(t) = T$, or (b) $p \neq s$ and p receives for the first time at t a message (T,s,h,σ) that passes both the "too early" T-h$\epsilon < C_p(t)$ and the "too late" $C_p(t) < T + h(\delta + \epsilon)$ timeliness acceptance tests. We denote by $\mathcal{T}$' the protocol with infinite local histories that is obtained by removing from $\mathcal{T}$ the "late message" acceptance test (5.6) and the local history garbage collection (4.6).

Lemma 4: If all correct processors follow protocol $\mathcal{T}$' then ψ propagates among neighbors.

Proof: Assume correct processor p learns ψ at real time t and let q be a correct neighbor linked by a correct link to p. We analyze two cases: p=s and $p \neq s$ If p=s then p sends a message (T,s,1,σ) to q at t (line 4.4 of $\mathcal{T}$'). We want to show that the message q receives will pass the timeliness acceptance tests of q. By Assumption 5, the (T,s,1,σ) message is received by q at a real time u such that $C_q(u) \leq C_q(t) + \delta$. Since, by Assumption 3, $C_q(t) < C_s(t) + \epsilon$, it follows that $C_q(u) < C_s(t) + \delta + \epsilon$, i.e. the "too late" acceptance test is passed at q. The "too early" acceptance test at q is passed because, by Assumption 5 and the monotonicity of C_s (Assumption 3), we have $C_s(t) - \epsilon < C_s(u) - \epsilon$ and by Assumption 3 we have $C_s(u) - \epsilon < C_q(u)$. Consider now the case $p \neq s$. Assume p learns ψ by receiving from some processor q' a message (T,s,h,σ) which passes the "too early" and "too late" acceptance tests, i.e. T-h $\epsilon < C_p(t)$ and $C_p(t) < T + h(\delta + \epsilon)$. Let q be a correct neighbor linked to p by a correct link. If q=q', q already learned ψ by t, so let us only investigate the more interesting case $q \neq q'$. By line (5.8) p sends at t a message (T,s,$h+1$,σ) to q, and by Assumption 5 the message is received at q at a real time u such that $C_q(u) \leq C_q(t) + \delta$. We have to show that this message passes the timeliness acceptance tests at q. Since, by Assumption 3, $C_q(t) < C_p(t) + \epsilon$, it follows that $C_q(u) < C_p(t) + \delta + \epsilon$. From the above inequality and our hypothesis $C_p(t) < T + h(\delta + \epsilon)$, it now follows that $C_q(u) < T + (h+1)(\delta + \epsilon)$, i.e., the "too late" acceptance test is passed at q. The "too early" acceptance test is passed because, by Assumptions 3 and 5, $C_p(t) - \epsilon < C_q(u)$, which, together with our hypothesis T-h $\epsilon < C_p(t)$, implies T-(h+1) $\epsilon < C_q(u)$. □

Lemma 5: If all correct processors follow protocol $\mathcal{T}'$ and a correct processor inserts (T,s,σ) in its history, then each correct processor inserts (T,s,σ) in its history before local time $T + \Delta_t$.

Proof: Let t be the earliest real time at which some correct processor p inserts (T,s,σ) in its history. If p=s, then by a reasoning similar to that for the case p=s in the proof of Lemma 2, we conclude that each correct processor q inserts (T,s,σ) in its history before local time $T + d\delta + \epsilon \leq T + \Delta_t$. If $p \neq s$, let (T,s,h,σ) be the message by which p learns ψ. Since in the worst case there can be a path of at most π faulty processors between s and p (i.e. h$\leq \pi$) and the message passes p's "too late" acceptance test, we have $C_p(t) < T + \pi(\delta + \epsilon)$. By diffusion induction there exists a real time u such that each correct processor q learns ψ at u and $C_p(u) \leq C_p(t) + d\delta$. Since, by Assumption 3, $C_q(u) < C_p(u) + \epsilon$, it follows that $C_q(u) < T + \pi(\delta + \epsilon) + d\delta + \epsilon$. □

Lemma 6: If all processors follow protocol $\mathcal{T}$ and a correct processor inserts a message (T,s,σ) in its history, then each correct processor inserts (T,s,σ) in its history before local time $T + \Delta_t$. If the maximum number of updates broadcast by a processor per time unit is bounded, the history H of any processor stays bounded.

Proof: The proof is similar to that of Lemma 3, the main difference being that, in the case of timing failures, for any local time T, histories can contain messages with timestamps in the range [T-$\Delta_t, T + (\pi + 1)\epsilon$). □

Theorem 2: The second protocol possesses the termination, atomicity, and order properties.

The proof is similar to that of Theorem 1. □

6.2 The Second Protocol is Not Tolerant of Byzantine Failures

We construct a counter-example showing that a Byzantine failure occurrence can lead to a violation of the atomicity property. Consider a totally connected system of four processors s (sender), f (faulty), e (early, correct), and l (late, correct), such that e's clock is z time units in advance of l's clock, 0 < z < ϵ, i.e. when l's clock indicates U, e's clock indicates U+z. Assume π=2, and λ=0; so Δ_t is $3(\delta + \epsilon)$. Suppose that the broadcast of (T,s,1,σ) by s is interrupted by a crash so that the message is sent only to f. Suppose that f "by mistake" increments the hop count by two instead of one, and forwards the message (T,s,3,σ) to the correct processors e and l in such a way that these messages arrive when e's clock shows T+$\Delta_t + z/2$ and l's clock shows T+$\Delta_t - z/2$. The update σ will be delivered at l but not at e. The atomicity requirement is thus violated.

7 Tolerance of Authentication-Detectable Byzantine Failures

As illustrated by the previous counter-example, a "Byzantine" processor can confuse a network of correct processors by forwarding appropriately altered messages on behalf of correct processors at appropriately chosen moments. One way of preventing this phenomenon is to authenticate the messages exchanged by processors during a broadcast [DS], [LSP], so that messages corrupted by "Byzantine" processors can be recognized and discarded by correct processors. In this way, we are able to handle authentication-detectable Byzantine failures in a manner similar to the way we handle timing failures. Ignoring (for simplicity) the increase in message processing time due to authentication, we set the termination time of the third protocol to be the same as the termination time of the second protocol: $\Delta_b = \pi(\delta + \epsilon) + d\delta + \epsilon$. The reader should be warned that the δ in this formula is likely to be significantly larger than the corresponding term for the previous protocol because of the cost of authentication processing.

The detailed implementation of our third protocol is given in Figures 6-12. We assume that each processor p possesses a *signature* function Φ_p, which, for any string of characters x, generates a string of characters y=$\Phi_p(x)$ (called the signature of p on x). Every processor q knows the names of all other processors in the communication network, and for each p$\in$G, q has access to an *authentication* predicate $\Theta(x, p, y)$ which yields true if and only if y= $\Phi_p(x)$. We assume that if processor q receives a string (x,p,y) as part of a message m from any processor, and $\Theta(x, p, y)$ is true, then p actually sent the string (x,p,y) in m. (If the authentication predicate fails to detect message forgery, then our last protocol can no longer guarantee atomicity in the presence of Byzantine failures.) The proper selection of the Φ_p and Θ functions for a given environment depends on the likely cause of message corruption. If the source of message corruption is unintentional (e.g., transmission errors due to random noise on a link or hardware malfunction) then simple signature and authentication functions like the error detecting/correcting codes studied in [PW] are appropriate. If the source of message corruption is intentional, e.g., an act of sabotage, then more elaborate authentication schemes like those discussed in [RSA] should be used. In any case there is always a small but non-zero probability that a corrupted message will be accepted as authentic.

We implement message authentication by using three procedures "sign", "cosign", and "authenticate", and a new signed message data type "Smsg" (Figure 6). These are all described in a Pascal-like language supporting recursive type declaration.

```
type Smsg =
record case : tag: (first,relayed) of
    first: (timestamp: Time; update: Update);
    relayed: (incoming: Smsg);
    procid: Processor;
    signature: string;
end;
```

Figure 6: The Signed-Message data type

A signed message (of type Smsg) that has been signed by k processors $p_1, ..., p_k$ has the structure

$(relayed, \ldots (relayed, (first, T, \sigma, p_1, s_1), p_2, s_2), \ldots p_k, s_k)$

where T and σ are the timestamp and update inserted by the message source p_1 and s_i are signatures.

```
procedure sign(in T:Time; $\sigma$:Update; out x: Smsg);
begin x.tag $\leftarrow$'first'; x.timestamp$\leftarrow$T;
    x.update$\leftarrow$ $\sigma$; x.procid$\leftarrow$myid;
    x.signature$\leftarrow$ $\Phi_{myid}$(x.tag,T,$\sigma$);
end;
```

Figure 7: The sign procedure

The sign procedure (Figure 7) is invoked by the originator of a broadcast (T,s,σ) to produce a message x containing the originator's signature. The co-sign procedure (Figure 8) is invoked by a processor r which forwards an incoming message x already signed by other processors; it yields a new message y with r's signature appended to the list of signatures on x.

```
procedure co-sign(in x:Smsg; out y: Smsg);
begin y.tag $\leftarrow$'relayed'; y.incoming $\leftarrow$ x;
    y.procid $\leftarrow$ myid; y.signature$\leftarrow$ $\Phi_{myid}$(y.tag,x);
end;
```

Figure 8: The co-sign procedure

The authenticate procedure (Figure 9) verifies the authenticity of an incoming message. It assigns the Boolean output parameter a the value false if an alteration of the original message is detected. If no alteration of the original message content is detected, the final value of a is true and the remaining output parameters T, σ and S are assigned the timestamp, the original update included in the message and the sequence S of processor names that have signed the message, respectively. The identity of the initiator is the first element of the sequence, denoted first(S), and the number of hops (i.e., the number of intermediate links) traversed by the message is the length of the sequence, denoted |S|.

```
procedure authenticate(in x:Smsg; out a: Boolean; T:Time;
    $\sigma$:Update, S:Sequence-of-Processor);
begin if x.tag='first' and $\neg\Theta$((x.tag,x.timestamp,x.update),x.procid,x.signature)
    or x.tag='relayed' and $\neg\Theta$((x.tag,x.incoming),x.procid,x.signature)
    then a$\leftarrow$false
    else if x.tag='first'
        then T$\leftarrow$x.timestamp; $\sigma$ $\leftarrow$x.update; S$\leftarrow$<>; a$\leftarrow$true;
        else authenticate(x.incoming,a,T,$\sigma$,S)
        fi;
    fi;
    append(S,x.procid);
end;
```

Figure 9: The authenticate procedure

Except for the change concerning the authentication of messages, the structure of the Start task of the third protocol (Figure 10) is the same as that of the second protocol. In order to handle the case in which a faulty processor broadcasts several updates with the same timestamp, the type of the history variable H is changed to

$$var\ \text{H: Time} \times (\text{Processor} \times (\text{Update} \cup \{\ \perp\ \})),$$

where the symbol $\perp$ denotes a "null" update ($\perp \notin$ Update). Specifically, if a processor receives several distinct updates with the same broadcast identifier, it associates the null update with that broadcast. Thus, a null update in the history is an indication of a faulty sender.

```
task Start;
var T: Time; $\sigma$: Update; x: Smsg; l: Link;
cycle wait-for-broadcast($\sigma$); T $\leftarrow$ clock;
    sign(T,$\sigma$,x);
    for all l $\in$ L do send(x) on l od;
```

```
6          add (T,s,$\sigma$) to H;
7          schedule Delivery(T) at $T + \Delta_b$;
8     endcycle;
```

Figure 10: Start Task of the third protocol

```
1     task Relay;
2     var U,T: Time; $\sigma$: Update; s: Processor; i,l: Link;
3          x,y: Smsg; a: Boolean; S: Sequence-of-Processor;
4     cycle receive(x) from i; U $\leftarrow$ clock;
5          authenticate(x,a,T,$\sigma$,S);
6          if a=false then "forged message" iterate fi;
7          if duplicates(S) then "duplicate signatures" iterate fi;
8          if U > T-|S| $\epsilon$ then "too ealry" iterate fi;
9          if U < T+|S| $(\delta + \epsilon)$ then "too late" iterate fi;
10         if U $\geq T + \Delta_b$ then "late message" iterate fi;
11         s$\leftarrow$first(S);
12         if there is $\sigma$' such that (T,s,$\sigma$') is in H
13         then if $\sigma$'=$\perp$ then "faulty sender" iterate fi;
14              if $\sigma$'=$\sigma$
15              then "deja vue" iterate
16              else replace (T,s,$\sigma$') by (T,s,$\perp$) in H
17              fi
18         else add (T,s,$\sigma$) to H;
19              schedule Delivery(T) at $T + \Delta_b$
20         fi;
21         co-sign(x,y);
22         for all l $\in$ L-{i} do send(y) on l od;
23    endcycle;
```

Figure 11. Relay Task of the third protocol

The Relay task of the third protocol (Figure 11) works as follows. Upon receipt of a message (11.4), the message is checked for authenticity (11.5) and if corrupted, the message is discarded (11.6) Then, the sequence of signatures of the processors that have accepted the message is examined to ensure that there are no duplicates; if there are any duplicate signatures, the message is discarded (11.7). Since processor signatures are authenticated, the number of signatures |S| on a message can be trusted and can be used as a hop count in determining the timeliness of the message (11.8, 11.9). No confusions such as those illustrated in the previous counter-example can occur unless the authentication scheme is compromised. If the incoming message is authentic, has no duplicate signatures, and is timely, then the history variable H is examined to determine whether the message is the first of a new broadcast (11.18). If this is the case, the history variable H is updated with the information that the sender s=first(S) has sent update σ at time T, the Delivery task is scheduled to start processing and possibly delivering the received update at (local clock) time T+Δ_b (11.19), and the received message is cosigned and forwarded (11.21, 11.22). If the received update σ has already been recorded in H (because it was received via an alternate path), it is discarded (11.15). If σ is a second update for a broadcast identified (T,s), then the sender must be faulty. This fact is recorded by setting H(T)(s) to the null update (11.16). The message is then cosigned and forwarded so that other correct processors also learn the sender's failure (11.21, 11.22). Finally, if σ is associated with a broadcast identifier to which H has already associated the null update (i.e., it is already known that the originator of the broadcast (t,s) is faulty), then the received update is simply discarded (11.13).

```
1  task Delivery(T:Time);
2  var val: Processor $\times$ (Update$\cup\{\perp\}$);
3  val $\leftarrow$ $\{$(s,$\sigma$) | (T,s,$\sigma$) $\in$H and $\sigma \neq \perp\}$;
4  sort val by processor name lexicographically;
5  for all (s,$\sigma$) $\in$ val in order do deliver($\sigma$) od;
6  delete all triples with first element T from H;
```

Figure 12: Delivery Task of the third protocol

The Delivery task (Figure 12) delivers at local time T+Δ_b all updates broadcast correctly at time T. If exactly one update has been accepted for a broadcast initiated at clock time T, then that update is delivered (12.3, 12.5), otherwise no update is delivered (12.3). In either case, the updates associated with broadcasts initiated at clock time T are deleted from H (12.6) to ensure H stays bounded.

7.1 Correctness of the Third Protocol

Our third protocol (denoted $\mathcal{B}$ in this proof) diffuses two kinds of propositions: a proposition ψ(T,s,σ) of the (by now familiar) form "processor s broadcast update σ at local time T" and a proposition ϕ(T,s,σ) of the form "either ψ(T,s,σ) or there exist two distinct updates σ_1 and σ_2, such that processor s has initiated the broadcast of these updates with *the same* timestamp T." A correct processor *learns* ψ(T,s,σ) at real time t if either (a) p=s and p inserts (T,s,σ) in its history at t, or (b) p $\neq$ s and p receives a message x such that the authenticate procedure terminates successfully by returning (true,T,σ,S), s=first(S), and the processing of x results in an update of the local history (either by adding (T,s,σ) or (T,s,$\perp$) to H). For any update σ, a correct processor p *learns* ϕ(T,s,σ) at real time t if it learns ψ(T,s,σ) at t or if there exist distinct updates σ_1 and σ_2, such that p learns ψ(T,s,σ_1)

at u $\leq$ t and p learns ψ(T,s,σ_2) at t. Since, if p=s and p is correct, it is not possible for p to learn that it broadcast two different updates with identical timestamps, correct processor p learns ϕ(T,s,σ) at t if, and only if, (a) p adds (T,s,σ) to H at t or (b) p receives at t a message x such that the authenticate procedure terminates successfully by returning (true,T,σ,S), s=first(S), and the processing of x results in the action H(T)(s)$\leftarrow\perp$ (11.16). We denote by $\mathcal{B}$' the protocol with infinite local history obtained from $\mathcal{B}$ by removing from $\mathcal{B}$ the "late message" acceptance test (11.10) and the local history garbage collection (12.6).

Lemma 7: If all correct processors follow protocol $\mathcal{B}$' and processor s is correct, then ψ(T,s,σ) propagates among neighbors.

The proof is analogous to that of Lemma 4. It relies on the observation that if the sender s is correct, and a correct processor p learns ψ(T,s,σ), then no correct neighbor q of p can ever learn ψ(T,s,σ_1) with $\sigma_1 \neq \sigma$. Thus, the message p sends to q either causes q to insert (T,s,σ) in its history or is simply discarded if q has already inserted (T,s,σ) in its history.

Lemma 8: If all correct processors follow protocol $\mathcal{B}$' then ϕ(T,s,σ) propagates among neighbors.

Proof: Assume correct processor p learns ϕ(T,s,σ) at real time t, and let q be a correct neighbor linked by a correct link to p. If p=s, then p actually learns ψ(T,s,σ) (recall a correct processor uses different timestamps for different updates), and, by Lemma 7, ψ, and hence ϕ, propagate to q within δ clock time units. Consider now the other (more interesting) case p $\neq$ s and let x be the message received by p at t from some neighbor q'. We have to analyze two cases: a) p learns ϕ(T,s,σ) by learning ψ(T,s,σ) and b) p learns ϕ(T,s,σ) by learning ψ(T,s,σ_1) after earlier it learned ψ(T,s,σ_2), where $\sigma_1 \neq \sigma_2$. If p learns ϕ(T,s,σ) from q' by learning ψ(T,s,σ), then p forwards a message y with its signature appended to those on x to all neighbors except q' (lines (11.18)-(11.22)). If q=q', then q already learned ψ(T,s,σ), and hence ϕ(T,s,σ), earlier. If q $\neq$ q', q receives at some real time u a message y conveying ψ(T,s,σ). Our assumption that the link between p and q is correct implies that the message y passes the acceptance tests (11.8)-(11.10) at q. If, when y is received, H(T)(s)=$\perp$, then q has already learned ϕ(T,s,σ) by u, else if the history of q contains (T,s,σ_1) for some $\sigma_1 \neq\perp$, then q learns ψ(T,s,σ), and hence ϕ(T,s,σ), by time u, else, q learns ψ(T,s,σ), and hence ϕ(T,s,σ), at time u. Consider now the case b) when p learns ϕ(T,s,σ) by learning ψ(T,s,σ_1) after it learned ψ(T,s,σ_2) earlier, $\sigma_1 \neq \sigma_2$, and let q"$\neq$q' be the neighbor which sent the message conveying ψ(T,s,σ_2) to p. If q $\notin$ q',q", then p has by t forwarded to q a message y" conveying ψ(T,s,σ_2) and at t p forwards another message y' conveying ψ(T,s,σ_1), so q learns ϕ(T,s,σ). If q=q', then q learned ψ(T,s,σ_1) before t, and since p forwarded a message conveying ψ(T,s,σ_2) to q' before receiving from q' the message x, it follows that q learns ϕ(T,s,σ) by the time it receives that message. If q=q", then q learned ψ(T,s,σ_2) before t, and since p forwarded a message y' conveying ψ(T,s,σ_1) to all neighbors except q', q learned ϕ(T,s,σ) by the later of the time of receipt of y' or the time q learned ψ(T,s,σ_2). The last case that remains to be analyzed is q'=q", i.e. the same neighbor q' sent to p the two messages conveying p ψ(T,s,σ_1) and ψ(T,s,σ_2). If q=q', then q learned ψ(T,s,σ_1) and ψ(T,s,σ_2), and hence ϕ(T,s,σ) by t, else, if q$\neq$q', then p forwards to q two messages conveying ψ(T,s,σ_1) and ψ(T,s,σ_2), so q learned ϕ(T,s,σ) by the time it received the later of the two messages. □

Lemma 9: If all correct processors follow protocol $\mathcal{B}$' and the initiator s of a broadcast (T,s,σ) is correct, then each correct processor p inserts (T,s,σ) in its local history before local time T+Δ_b.

The proof, which relies on Lemma 7, is analogous to the proof of Lemma 5 and is omitted.

Lemma 10: If all correct processors follow protocol $\mathcal{B}$' and, for any σ, there is a correct processor p such that its history contains (T,s,σ) or (T,s,$\perp$), then, for each correct processor q, q's history contains (T,s,σ) or (T,s,$\perp$) before time T+Δ_b on q's clock.

Proof: If p=s, then p necessarily has (T,s,σ) in its history and by diffusion induction and Assumption 3, each correct q inserts (T,s,σ) in its history before T+$d\delta + \epsilon \leq$ T+Δ_b. Consider now p $\neq$ s. Recall that "p learns ϕ(T,s,σ) at t" is equivalent to "p inserts in its history either (T,s,σ) or (T,s,$\perp$) at t." Let t be the earliest real time at which a correct processor p learns ϕ(T,s,σ). By a reasoning analogous to that in the proof of Lemma 5 we have $C_p(t) < T + \pi(\delta + \epsilon)$, and by, Lemma 8, diffusion induction and Assumption 3 it follows that every correct processor q learns ϕ(T,s,σ) before time T+$\pi(\delta + \epsilon) + d\delta + \epsilon$ on its clock. It thus follows that, when q's clock displays time T+Δ_b, q's history is such that it contains (T,s,σ) or (T,s,$\perp$). □

Lemma 11: If all correct processors follow protocol $\mathcal{B}$' and, for some T and s, there exists a correct processor p that inserts (T,s,$\perp$) in its history, then each correct processor q has (T,s,$\perp$) in its history before local time T+Δ_b.

Proof: Suppose processor p inserts (T,s,$\perp$) in its history. Then there exist two updates σ_1, σ_2, $\sigma_1 \neq \sigma_2$, such that p learns ψ(T,s,σ_2) after learning at an earlier time ψ(T,s,σ_1). Note that, by definition, for any σ, if some correct processor learns ψ(T,s,σ), then it has (T,s,σ) or (T,s,$\perp$) in its history. Thus, by Lemma 10, each correct processor has (T,s,σ_1) or (T,s,$\perp$) in its history and each correct processor has (T,s,σ_2) or (T,s,$\perp$) before local time T+Δ_b. Since $\sigma_1 \neq \sigma_2$, each correct processor must have (T,s,$\perp$) in its local history before local time T+Δ_b □

Lemma 12: If all processors follow protocol $\mathcal{B}$ and a correct processor s initiates at local time T the broadcast of some update σ, then all correct processors insert (T,s,σ) in their history before time T+Δ_b on their clock. If a correct processor inserts (T,s,$\perp$) in its history, then each correct processor inserts (T,s,$\perp$) in its history before local time T+Δ_b. If the maximum number of updates broadcast by a processor per time unit is bounded, the history H of any processor stays bounded.

Proof: The proof, similar to that of Lemma 6, is omitted.

Theorem 3: The third protocol possesses the termination, atomicity, and order properties.

The proof, which relies on Lemma 12, is similar to that of Theorem 2 and is therefore omitted.

8 Performance

8.1 Messages

In the absence of failures, the initiator s of an atomic broadcast sends d_s messages to its neighbors, where d_s denotes the degree of s (i.e. the number of its adjacent links). Each processor $q \neq s$ that receives a message from a processor p sends d_q-1 messages to all its neighbors (except p). Since the sum of all node degrees of a network is twice the number of network links, it follows that each atomic broadcast costs 2m-(n-1) messages, where m is the number of links and n is the number of nodes of the network. For example, an atomic broadcast among 8 processors arranged in a 3-dimensional cube requires 17 messages in the absence of failures.

While we cannot compare the message cost of our algorithms directly with those of a round based model, we can compare them with the results of a straightforward conversion. There are two issues in such a conversion. The first is the issue of complete connectivity. For algorithms that are only designed for completely connected networks, some routing scheme must be used to simulate the complete connectivity. In general messages must be sent along many disjoint routes to overcome the failure of intermediate processors. Otherwise, the number of component failures tolerated by the converted protocol may be dramatically reduced from that of the original. However, there are round based protocols that do not depend on complete connectivity (e.g. the fourth protocol of [LSP] based on authentication). In this case our message costs are as good but may not be better, since these protocols usually use some variant of diffusion.

For illustrative purposes, we consider the round based protocol in [DS] that was designed for complete connectivity. Moreover, we consider a straightforward conversion of this protocol to our model, using an arbitrary minimal length routing scheme to simulate complete "logical" connectivity between processors. Since "logical" messages sent by processors are implemented as sequences of (one-hop) messages sent among neighbors, some of the messages sent in each round will be redundant. Indeed, if a "logical" message has to be sent from a processor s to a non-neighbor processor r, and p is the neighbor of s on the path to r selected by the message routing algorithm used, then the message s sends to p to be forwarded to q is redundant with the message that s sends to p for direct consumption. For the example of 8 processors arranged in a 3-dimensional cube, a round of logical messages sent by one processor to the 7 others costs 12 (one-hop) messages. Thus, for $\pi \geq 1$, our converted round based agreement protocol tolerant of timing or authentication-detectable Byzantine failures sends in the absence of failures at least $12 + 7 \times 12 = 216$ messages, compared to the 17 messages needed by a diffusion based protocol for any $\pi \geq 1$.

8.2 Termination Time

The termination time for an atomic broadcast depends on the network topology and on the class of failures to be tolerated. In the absence of information about the network topology except that the number of processors is bounded above by n, n-1 can be taken to be an upper bound on $\pi + d$. Clock synchronization algorithms which can provide an ϵ close to $d\delta$ are investigated in [CAS,DHSS]. For simplicity, we assume here an ϵ of $(\pi + d)\delta$. Thus, for omission failures, the termination time of an atomic broadcast is linear in n: $\Delta_o = 2(\pi + d)\delta$ is bounded above by $2(n - 1)\delta$. For timing and Byzantine failures, the termination time is proportional to the product of the number of processors and the number of processor failures to be tolerated: $\Delta_t = (\pi + 2)(\pi + d)\delta$ is bounded above by $(\pi + 2)(n - 1)\delta$. As a numerical example, consider the case of 8 processors arranged in some arbitrary way to form a network. Assume that the link delay bound δ is 0.01 seconds and that we want to tolerate up to two processor failures. The termination time for omission failures is 0.14 seconds, and for timing failures is 0.28 seconds. (For authentication-detectable Byzantine failures, we might scale these numbers up by a factor of 10, reflecting the increase in δ due to authentication processing.) If more information about network topology is available, then a better expression can be computed for the network diffusion time d δ . Note that the expression $\pi+d$ corresponds to a worst case path consisting of π hops between faulty processors followed by d hops along a shortest path in the surviving network of correct processors and links. For example, if the eight processors above are arranged in a 3-dimensional cube and we need tolerate no link failures, the approximate termination times for omission and timing failures are cut to 0.10 and 0.20 seconds respectively. This is because $\pi + d$ is bounded above by 5: if the two faulty processors are adjacent then the diameter of the surviving network is at most 3, and if they are not adjacent the diameter can be 4 but 2 faulty processors cannot be encountered on a path before a correct processor is encountered.

Straightforward conversions of rounds based protocols into our system model would require that each round include not only the worst case time for sending a message between processors but also an extra delay corresponding to the worst case duration between the end of a round on one processor clock and the end of the same round on another processor clock. For example the fourth algorithm of [LSP] which terminates in $\pi + d$ rounds by using diffusion requires a termination time of $(\pi + d)(\delta + \epsilon)$ clock time units [LSP]. A round based protocol such as [DS] which assumes full network connectivity would require that each round lasts for at least $d\delta + \epsilon$ clock time units. To tolerate π failures, the [DS] protocol needs at least $\pi + 1$ rounds, that is, a conversion to our model would require at least $(\pi + 1)(d\delta + \epsilon)$ clock time units. The above termination times are always equal or greater than the termination time $\pi(\delta + \epsilon) + d\delta + \epsilon$ of our third protocol with equality for a fully connected surviving network with $d = 1$. When only omission failures can occur, our first protocol has a better termination than any straightforward conversion of the above rounds based protocols, even in a fully connected network, since $\pi\delta + d\delta + \epsilon$ is smaller than the minimum of $(\pi + d)(\delta + \epsilon)$ and $(\pi + 1)(d\delta + \epsilon)$ whenever $\epsilon > 0$. When ϵ is large compared to δ, the difference can be dramatic and is one of the justifications for studying tolerance to omission failures in our model.

We had long conjectured that the termination times provided by our algorithms would turn out to be optimal. However, we were only able to prove the lower bounds of the next section. Recent results have shown that our second and third algorithms do not in fact provide optimal termination times [SDC]. Our termination times hold for each execution uniformly and are not simply worst case. Recent work has shown that it is often possible to terminate earlier than the worst case time using acknowledgement in addition to diffusion [PG, GSTC]. Thus our algorithms neither achieve the best worst case time nor the best expected or average time. However, they still remain competitive from the point of view of simplicity. Work on very closely related problems suggests that the time complexity attributed to algorithms operating in our model is very sensitive to the definition of termination time. Recent results [ADLS, Po] on the real time required for the consensus problem are not readily comparable with our termination time results even though our models describe exactly the same phenomena. A better understanding of the relationship between our results and these real time results is the subject of current research. While our upper and lower bounds coincide for omission failures, a gap remains between the corresponding upper and lower bounds for the real time required for consensus.

9 Lower Bounds

9.1 Runs and Specifications

In this section we present a slightly more formal model for the execution of a distributed system. We have postponed the following formalities in the interest of readability for the rest of the paper.

Formally we view a distributed system as being composed of processors and links. The processors and links of our system could be described by using the language of IO automata [LT]; however, we are only interested in the input/output behavior of these components so our model can be somewhat simplified. An *execution* of the distributed system consists of a sequence of events for each component of the system. An *event* is a pair consisting of an *action*, and a *real-time* indicating when the action completes. An action is either a state transition specific to a component, such as the receipt or the sending of a message, or is a duration action corresponding to the passage of a unit of real-time. The events at a component are divided into two sets: *input* and *output*. For processors the input events are the receipt of a message from a link, the receipt of an update and the passage of a unit of real time. For links the input events are the receipt of a message from a sending processor and the passage of a unit of real time. The output events for processors are the sending of a message on a link and the delivery of an update (to a process outside the system). The output events for links are the sending of messages to processors. If an output event from a processor to a link (or vice versa) occurs in an execution then the corresponding input event at the link (processor) also occurs in the execution.

Executions are deemed to provide semantics for our pseudocode algorithms in the obvious way: correct processors execute the pseudocode and experience output events within the timing constraints specified by our assumptions. Correct links correlate their input events at one end with output events at the other, again within the timing constraints specified. The behavior of faulty components is only governed by the failure class under consideration. For brevity, we will use the term *run* as interchangeable with the term execution.

We assume some starting time called real time 0 for any run. Note that, even if there are no messages or updates, a run is necessarily infinite because it includes the events corresponding to the passage of units of real time.

We use the term *local history* to denote any finite prefix of a sequence of events that take place at one component during a run. A *specification* is a relation between local histories and output events (the actions specified to occur as a result of these histories) that satisfies the following property: let s be a specification, let h be a local history in the domain of s, let u be the latest time associated with an event in h, and let e be the set of real times associated by the relation with events in s(h), then e is bounded above by some real time (a deadline by which the output events are supposed to occur) and bounded below by u. A component *satisfies* its specification s in a run R if the following two properties hold: (1) for every local history h that is a prefix of R in the domain of s, there is an output event e $\in$ R such that e is in s(h); and (2) for every output event e $\in$ R, there is a local history h such that h is a prefix of R in the domain of s and e is in s(h). Recall that when a component satisfies its specification in a run it is called correct in that run.

Note that Assumptions 3 and 5 from Section 3 constrain the possible specifications for links and for sub-component clocks. Our results will hold for any specifications that satisfy our assumptions. For example, it would be sufficient to specify that (a) each correct clock maintain linear envelope synchronization (v. [DHSS], [CAS]) so that $\alpha(u-v) \leq C_p(u) - C_p(v) \leq \beta(u-v) + \gamma$, for all real times $u > v$, and that (b) each message be delivered by a link within $(\delta - \gamma)/\beta$ time units of the real time at which it was sent on the link. However, such a specification is not necessary.

9.2 A time lower bound for crash failures

Here we generalize the last example of the previous Section and provide a lower bound on the termination time required by any atomic broadcast algorithm in the presence of omission failures. We then prove a second lower bound for the termination time of any atomic broadcast in the presence of authentication-detectable Byzantine failures. Our first lower bound proof is based on a proof in [DS] for a much simpler model; but the conversion from the simple model to our system model, especially from completely connected to arbitrary networks, was not at all trivial. In fact we leave open a characterization of the networks for which our algorithms provide optimal termination time, though we conjecture that they include at least the *symmetric* networks in which, for any pair of nodes p and q, there is an automorphism mapping p to q.

For our first result we now fix π andλ, the numbers of processors and links that suffer omission failures. We say that P-{p} is an *h-path* with source s that leads to p if, and only if, P is an acyclic path of h $\geq$0 hops (i.e. h links) between two processors s and p. An *at most h-path* is any k-path with $0 \leq k \leq h$. For any communication network G, we define an *adverse case* to be a selection of a set E $\subset$ G of processors and links, a processor s in G-E, and positive integers h and k, such that (1) the number of links in E is at most λ, (2) the sum of h and the number of processors in E is at most π, (3) there exists an h-path from s in G-E, and (4) for any at most h-path P in G-E originating in s and leading to some processor p $\in$ G-E, the network X = G-(E $\cup$ P) is connected and contains a processor q $\neq$ p at least k hops away from p. A *weakly adverse case* for G is a selection <E,s,h,k> as above such that (1) the number of links in E is at most λ, (2) the sum of h and the number of processors in E is at most π, (3) there exists an h-path P in G-E originating in s and leading to some processor p $\in$ G-E, such that the network X = G-(E $\cup$ P) is connected and contains a processor q $\neq$ p at least k hops away from p. The intuition behind the use of an adverse case <E,s,h,k> in our lower bounds proof is as follows: E contains all processors and links of G that crash before the initiation of an atomic broadcast by processor s, no links in G-E crash during the broadcast, the at most h-path P from s to some processor p contains all processors that suffer failures during the broadcast, p is the "closest" correct processor to s on the path, and q is used as a witness for equivalent protocol runs.

We say that a network G *requires x steps* if there is an adverse case <E,s,h,k> for G with h+k=x. We say that network G *allows x steps* if there is a weakly adverse case <E,s,h,k> for G with h+k=x. If x_{max} is the largest number of steps allowed by a network, then we can set the termination time Δ of an omission failure tolerant protocol specifically tailored for that network to $x_{max}\delta + \epsilon$ and prove its correctness in a manner analogous to the way we proved the correctness of our first protocol, which uses the general upper bound π+d for x_{max}.

Theorem 4: If the communication network G requires x steps, then any atomic broadcast protocol tolerant of up to π processor and λ link omission failures has a termination time of at least $x\delta + \epsilon$.

Proof: Suppose that, for given π and λ, G requires x steps, that is, G has an adverse case <E,s,h,x-h>, but there exists some protocol A that achieves atomic broadcast in the presence of up to π processor and λ link omission failures for some termination time D < x $\delta + \epsilon$. Let y and z be times such that 0 < y < δ, 0 < z < ϵ, and D < xy + z. Without loss of generality, we will assume that A is a deterministic protocol. If A were randomized we would restrict our attention to constructing a universe of runs (or executions) for A in which all random choices are made in the same way at all processors (e.g. all coins tossed by all processors are "heads").

We assume, without loss of generality, that the real times at which messages may be sent form a discrete countable subset of the set of all real times so that we can use induction on this set. Suppose otherwise that the set of real times at which A requires messages in all runs is dense. We can perturb A slightly to wait for the next instant in our discrete set and speed up the delivery so that the message is received at the same time. This transformation will produce runs that are indistinguishable by message receipt history, provided no runs are considered in which messages are delivered faster than the time between closest points in the discrete set. By Assumption 6, the number of messages A may require at any instant in the discrete set is finite.

Let $\mathcal{S}$ be the set of atomic broadcast runs (or executions) for protocol A, with initiator s starting the broadcast of some update $\sigma \in$ Update, | Update | $\geq$ 2, at real time 0, that satisfy the following properties: (1) all processor clocks run at the rate of real time, (2) for any t $\geq$0, $C_s(t)$ = t, (3) for any r $\neq$ s, and for any t $\geq$ 0: $C_r(t)$ = t+z, (4) all messages exchanged between processors take exactly y (real or clock) time units for transmission and processing, (5) each component of E fails before real time 0 and the only faulty links are in E, (6) all failures are crash failures (recall that after a crash failure a component ceases to send or relay any messages prescribed by A) (7) there is an at most h path α originating in s such that if H is the set of processors in G-E that fail, then (7.1) the processors in H are among the nodes of α, (7.2) any failure of a node f $\in$ H situated k $\geq$0 hops from s on α occurs at a real time t $\geq$ ky, (7.3) if processor f situated k hops from s on α crashes at time t (possibly after sending some messages required by A at that time), then no processor situated fewer than k hops from s on α sends any messages after t, and (7.4) if H is not empty, then the last processor on α is in H.

Note that since <E,s,h,x-h> is an adverse case, conditions (1) through (7) imply that there are no more than π processor failures and λ link failures during the broadcast of σ and that these failures do not disconnect the surviving network (see part 4 of the definition of adverse case). Note also that the empty path is assumed to be an at most h-path originating in s.

For each run Q, processor q, and real time t, let Msgs(Q,q,t) denote the sequence of pairs

$< m_i, r_i >$, i $\in$ {1,...,k}, such that protocol A requires q to send message m_i. to processor r_i at t in Q, the order of the sequence being the order in which A requires the messages to be sent. Given a run Q $\in \mathcal{S}$, we denote by Q($\leq$ t) the *partial run* through time t in which all processors and links behave as in Q. We also denote by Q(<t) the corresponding partial run up to time t. Since we have assumed A is deterministic, we assume that for any runs Q and Q' with Q(<t) = Q'(<t), we have Msgs(Q,q,t) = Msgs(Q',q,t).

For any partial run Q($\leq$ t) that satisfies conditions (1) through (7), we define the *conservative extension* of Q($\leq$t) to be the unique run in $\mathcal{S}$ in which each component (processor or link) continues to behave correctly according to A unless it has already crashed by time t. If some processor q crashes in Q($\leq$t) by time t that is, for some time t'$\leq$t q omits to send at least one message in Msgs(Q,q,t'), then q remains crashed in the conservative extension of Q($\leq$t) to the end of time. We say that two runs are *output equivalent* if any updates delivered by correct processors are the same in both runs. We define the relation *witness equivalence* to be the transitive closure in $\mathcal{S}$ of the relation that holds between two runs when there is a processor q correct in both that cannot distinguish between them on the basis of its message history through time D on q's clock. Under the assumption that A is correct, if two runs are witness equivalent, then they are output equivalent.

To prove Theorem 4, it is sufficient to show the output equivalence of two runs S_c and S_f in $\mathcal{S}$ such that in S_c processor s initiates the atomic broadcast correctly at time 0 and each processor in G-E is correct to time D on its clock, and in S_f processor s crashes without initiating atomic broadcast and does not send any messages at or after time 0 while each other processor is correct to time D on its clock. (Note that the real time of a crash of processor q in run Q is defined as the first time t such that Msgs(Q,q,t) is nonempty and q does not send some message from Msgs(Q,q,t) in Q. Since we assume processor s must send its update if it is correct, there is such a time that processor s crashes in S_f; but the time of crash may be after 0.) The output equivalence of these two runs of A contradicts the hypothetical correctness of protocol A, since if A were correct, in S_c all correct processors would deliver the update σ at time D on their clocks and in S_f. no processor would deliver the update at time D on its clock.

In order to prove that the runs S_c and S_f are witness (and hence output) equivalent, we define by mutual recursion a *crash correction* operator, τ_c, and a *crash insertion* operator, τ_f, each of type

$$\mathcal{S} \times \text{Path} \times \text{Processor} \times \text{Time} \rightarrow \mathcal{S}.$$

When applied to a quadruple (Q,α,q,t) in its domain, each of these operators yields a run R that is witness equivalent to Q. Moreover, the operators are defined in such a way that there exist paths α and β and a real time t $\geq$ 0 such that $S_c = \tau_c(S_f, \alpha, s, t)$ and $S_f = \tau_f(S_c, \beta, s, 0)$. In particular, α is any path consisting of s and a link from s to another processor not in E, t is the first time $\geq$ 0 such that Msgs(S_f,s,t) is not empty (so t is the time s crashes in S_f), and β is the empty path (viewed as leading to s).

Definition of the crash correction operator: The domain of τ_c is the set of all quadruples < Q,α,q,t>, where Q $\in \mathcal{S}$, α is an at most h-path from s that satisfies condition (7) in Q, q is the last processor on α, and t is a real time $\geq$ 0 such that q crashes at real time t in Q. Let $< m_i, r_i >$ i $\in$ {1,...,k} be the subsequence of Msgs(Q,q,t) that q did not send in Q at t. The run R = τ_c(Q,α,q,t) is defined as follows. Let Q_0 = Q.

- *Case a:* t $\geq$(h-1)y.
  - *Step 1:* For j = 1 to k, add m_j to $Q_{j-1}(\leq t)$ and let Q_j be the conservative extension of the resulting partial run.
- *Case b:* t < (h-1)y.
  - For j=1 to k,
    - *Case c:* r_j is on α or in E or the link from q to r_j is in E.
      - *Step 2:* Add m_j to Q_{j-1} to obtain Q_j.
    - *Case d:* r_j is neither on α nor in E and the link from q to r_j is not in E.
      - *Step 3:* Let β be the result of changing α to lead to r_j (with no other changes) and let $Q_{j-1.1}$ = τ_f(Q_{j-1},β,r_j,t+y).
      - *Step 4:* Add m_j to $Q_{j-1.1}$ to obtain $Q_{j-1.2}$.
      - *Step 5:* If r_j does not crash in $Q_{j-1.2}$, then let Q_j = $Q_{j-1.2}$; otherwise, let t' be the time that r_j crashes in $Q_{j-1.2}$, let β' be an extension by r_j and one link of β to produce an at most h-path that satisfies (7) in $Q_{j-1.2}$, and let Q_j = τ_c($Q_{j-1.2}$,β',r_j,t').
- *Step 6:* If q crashes in Q_k, then let t' be the time q crashes in Q_k and let R = τ_c(Q_k,α,q,t'); otherwise, let R = Q_k.

Definition of the crash insertion operator: The domain of τ_i is the set of all quadruples <Q,α,q,t> such that α is an at most (h-1)-path that leads to processor q, α satisfies condition (7) in run Q $\in \mathcal{S}$ and, if t' is the smallest real time $\geq$ t such that Msgs(Q,q,t') is nonempty, then the conservative extension of the result of removing Msgs(Q,q,t') from Q($\leq$t') is in $\mathcal{S}$ with an extension of α satisfying condition (7). The run R = τ_f(Q,α,q,t) is defined as follows. If there is no t' $\geq$t such that Msgs(Q,q,t') is nonempty, then R = Q; otherwise, let t' be the smallest such time and let $< m_i, r_i >$ i $\in$ {1,...,k} be the sequence Msgs(Q,q,t').

- *Case a:* t' $\geq$(h-1)y.
  - *Step 1:* Let Q_0 = Q. For j = k to 1, let Q(k-j+1) be the conservative extension of the result of removing m_j from $Q_{k-j}(\leq t')$. Let R = Q_k.
- *Case b:* t'<(h-1)y.
  - *Step 2:* If t' is the latest time such that Msgs(Q,q,t') is nonempty, then let Q_0 = Q; otherwise, let t" be the next time after t' with Msgs(Q,q,t") nonempty and let Q_0 = τ_f(Q,α,q,t").

* For j = k to 1,
  - *Case c:* r_j is on α or in E or the link from q to r_j is in E.
  - *Step 3:*Let Q_{k-j+1} be the result of removing m_j from Q_{k-j}.
  - *Case d:* r_j is neither on α nor in E and the link from q to r_j is not in E.
  - *Step 4:*Let β be the extension of α by q and one link from q to r_j and let $Q_{k-j.1} = \tau_f(Q_{k-j},\beta,r_j,t+y)$.
  - *Step 5:*Remove m_j from $Q_{k-j.1}$ to obtain $Q_{k-j.2}$.
  - *Step 6:*If r_j does not crash in $Q_{k-j.2}$, then let $Q_{k-j+1} = Q_{k-j.2}$; otherwise, let t' be the real time that r_j crashes in $Q_{k-j.2}$, let β' be an extension of β by r_j and one link that leads to another processor not in the extension, and let $Q_{k-j+1} = \tau_c(Q_{k-j.2},\beta',r_j,t')$.

- Finally, let R = Q_k.

For runs Q and R, processor p, and time t, we define the relation Q $\sim_{p,t}$ R to hold exactly when Q(<t) = R(<t) and, except for messages sent by p at time t, Q($\leq$t) = R($\leq$t). This shorthand will be useful in describing the results of applying our operators.

Lemma 13 : The two operators are well defined. If <Q,α,q,t> is in the domain of τ_c and R = τ_c(Q,α,q,t), then Q and R are witness equivalent, Q $\sim_{q,t}$ R and some prefix of α without q satisfies condition (7) in R. If <Q,α,q,t> is in the domain of τ_f and R = τ_f(Q,α,q,t), then Q and R are witness equivalent, Q $\sim_{q,t}$ R, q sends no messages in R at or after time t, and, either q does not crash in R and α satisfies condition (7) in R, or an extension of α by q and a link from q satisfies (7) in R.

Proof: We prove Lemma 13 by induction on the number of links in α between s and q or on the time t. Note that no step in the definition of either operator changes conditions (1) through (5). Thus we need check only conditions (6) and (7) to make sure that the run produced by each step is a member of $\mathcal{S}$

As a base case for the induction, assume that t $\geq$ (h-1)y. Note that by condition (7), if the number of links between s and q on α is h-1, then t must be $\geq$ (h-1)y.

Consider the crash correction operator applied to <Q,α,q,t> in its domain. Here we have case a and R is determined by step 1 and step 6. Also α satisfies (7) in Q_0=Q $\in \mathcal{S}$ and Msgs(Q,q,t) = Msgs(Q_0,q,t). Assume α satisfies (7) in $Q_{j-1} \in \mathcal{S}$ and Q $\sim_{q,t}$ Q_{j-1} so that Msgs(Q,q,t) = Msgs(Q_{j-1},q,t). Adding m_j to Q_{j-1}($\leq$t) does not change the fact that the partial run satisfies conditions (1) through (7), so taking the conservative extension produces a run Q_j in $\mathcal{S}$. Note that Q $\sim_{q,t}$ Q_j. Hence, Msgs(Q,q,t) = Msgs(Q_j,q,t); so q crashes in Q_j if, and only if, j $\neq$ k. Since < E,s,h,x-h > is an adverse case, there must be a processor p correct in both Q_{j-1} and Q_j that is at least x-h hops away from r_j. This p cannot distinguish between Q_{j-1} and Q_j until real time xy, which is after clock time D for p. Thus Q_{j-1} and Q_j are witness equivalent. Moreover, if q crashes in Q_j, then α satisfies

(7) in Q_j; otherwise j=k and the prefix of α that leads to q satisfies (7) in Q_k. By induction on j, Q is witness equivalent to Q_k, Q $\sim_{q,t}$ Q_k, and either α satisfies (7) in Q_k or q does not crash in Q_k and the prefix of α that leads to q satisfies (7) in Q_k. Note that Q_k is the conservative extension of a partial run in which q does not fail. Thus q does not fail in Q_k and R = Q_k according to step 6.

Second, consider the crash insertion operator applied to <Q,α,q,t> in its domain. If there is no time t'$\geq$t with Msgs(Q,q,t') nonempty, then R = Q, processor q sends no messages in R at or after t, and α satisfies (7) in R. Assume, without loss of generality, that R $\neq$ Q. Thus there is a smallest t'$\geq$t with Msgs(Q,q,t') nonempty. Again we have case a, since t' $\geq$ t $\geq$ (h-1)y, so R is determined by step 1 alone and Q_0 = Q. Thus α satisfies (7) in Q_0, Q $\sim_{q,t'}$ Q_0, and Msgs(Q,q,t') = Msgs(Q_0,q,t'). Let β be any extension of α by q and a link from q to another processor correct in Q. Since <Q,α,q,t> is in the domain of τ_f, β satisfies (7) in Q_1, which is the conservative extension of the result of removing m_k from Q_0($\leq$t'). Also Q $\sim_{q,t'}$ Q_1, so Msgs(Q,q,t') = Msgs(Q_1,q,t'). Since <E,s,h,x-h> is an adverse case, Q=Q_0 is witness equivalent to Q_1 by the argument for the crash correction operator. Assume β satisfies (7) in Q_{k-j}, Q $\sim_{q,t'}$ Q_{k-j}, and Q is witness equivalent to Q_{k-j}. Since Q_{k-j+1} is the conservative extension of the result of removing m_j from Q_{k-j}($\leq$t') and Msgs(Q,q,t') = Msgs(Q_{k-j},q,t'), β satisfies (7) in Q_{k-j+1} and Q $\sim_{q,t'}$ Q_{k-j+1}. Again, since <E,s,h,x-h> is an adverse case, Q is witness equivalent to Q_{k-j+1}. Thus by induction on j (from k to 1), β satisfies (7) in Q_k = R, Q $\sim_{q,t'}$ R, and Q is witness equivalent to R. Since R is the conservative extension of a partial run in which q crashes at t', q crashes at t' in R.

Now assume that <Q,α,q,t> is in the domain of τ_c and we have as induction hypothesis that the lemma holds for both operators when the number of links between s and the processor argument on the path argument is greater than the number of links between s and q on α or when the time argument is greater than t. We assume that the number of links between s and q on α is less than h-1 and that t < (h-1)y. Thus we have case b and R is determined by steps 2 through 6. Since Q_0 = Q, α satisfies condition (7) in Q_0, Q $\sim_{q,t}$ Q_0, and Q is witness equivalent to Q_0. Assume that α satisfies condition (7) in Q_{j-1}, Q $\sim_{q,t}$ Q_{j-1}, and Q is witness equivalent to Q_{j-1}. If case c holds, then either no message from q reaches r_j or r_j sends no messages after t (by condition (7)) in Q_{j-1}. According to step 2, Q_j is the result of adding m_j to Q_{j-1}. Thus Q $\sim_{q,t}$ Q_j so Msgs(Q,q,t)=Msgs(Q_j,q,t). If j<k then q crashes at t in Q_j and α satisfies condition (7) in Q_j. If j=k then either q crashes at a later time in Q_j, in which case α satisfies condition (7) in Q_j, or q does not crash in Q_j, in which case the prefix of α that leads to q satisfies (7) in Q_j. Since either no message reaches r_j from q or r_j sends no messages after t, no processor correct in both (excluding r_j) can distinguish Q_{j-1} from Q_j. Since <E,s,h,x-h> is an adverse case, there is a processor other than r_j correct in both. Thus they are witness equivalent. Assume case d holds. Q_j is determined by steps 3 through 5. Since q is the last node on α and there is a message from q to r_j in Msgs(Q,q,t), β is well defined. Since t<(h-1)y, there are at most h-1 processors on β and β satisfies (7) in Q and hence in Q_{j-1}. Let β' be the extension of β by the addition of r_j and a link from r_j to some processor that does not crash in Q_{j-1}. If t' is the first time $\geq$ t+y such that

Msgs(Q_{j-1},r_j,t') is nonempty, then β' satisfies (7) in the conservative extension of the result of removing Msgs(Q_{j-1},r_j,t') from Q_{j-1}($\leq$t'). Thus <Q_{j-1},β,r_j,t+y> is in the domain of τ_f. By the induction hypothesis, Q_{j-1} is witness equivalent to $Q_{j-1.1}$, $Q_{j-1} \sim_{r_j,t} Q_{j-1.1}$, either β or β' satisfies condition (7) in $Q_{j-1.1}$, and r_j sends no messages at or after t+y in $Q_{j-1.1}$. Thus no processor correct in both (excluding r_j) can distinguish between $Q_{j-1.1}$ and $Q_{j-1.2}$, which is obtained by adding m_j to $Q_{j-1.1}$ according to step 4. Also we have $Q_{j-1.1} \sim_{q,t} Q_{j-1.2}$, and either β or β' satisfies (7) in $Q_{j-1.2}$, since q sends no messages after t in $Q_{j-1.2}$ even if j=k and q does not crash at t. Since <E,s,h,x-h> is an adverse case, there is a processor other than r_j that is correct in both; so $Q_{j-1.1}$ is witness equivalent to $Q_{j-1.2}$. If r_j crashes in $Q_{j-1.2}$, then <$Q_{j-1.2}$,β',r_j,t'> is in the domain of τ_c and β' is longer than α; so Q_j, which is τ_c($Q_{j-1.2}$,β',r_j,t') according to step 5, is witness equivalent to $Q_{j-1.2}$ with $Q_j \sim_{r_j,t'} Q_{j-1.2}$ and a prefix of β satisfies (7) in Q_j by the induction hypothesis. Thus, by induction on j (from 1 to k), we have Q witness equivalent to Q_k, Q $\sim_{q,t}$ Q_k, and a prefix of β satisfies (7) in Q_k. If q does not crash in Q_k, then R = Q_k according to step 6, and a prefix of β without f, which is a prefix of α without f, satisfies (7) in R; otherwise, <Q_k,α,q,t'> is in the domain of τ_c, where t' is the time > t at which q crashes in Q_k. In the latter case, by the induction hypothesis, Q $\sim_{q,t}$ R, Q and R are witness equivalent, and a prefix of α without q satisfies (7) in R, since R = τ_c(Q_k,α,q,t').

Now assume that <Q,α,q,t> is in the domain of τ_f and we have the previous induction hypothesis. We assume that the number of links between s and q on α is less than h-1 and that t < (h-1)y. Thus at most h-2 processors crash in Q. (This is important because we will want to crash as many as an additional two processors during the execution of τ_f.) If there is no time t'$\geq$t such that Msgs(Q,q,t') is nonempty, then R = Q and we are done. If there is such a t' but only greater than t, then again we are done by the induction hypothesis, because R = τ_f(Q,α,q,t) = τ_f(Q,α,q,t'). Thus we assume that there is such a t' and that t' = t. Hence we have case b and R is determined by steps 2 through 6. According to step 2, if t is the latest time such that Msgs(Q,q,t) is nonempty, then Q_0 = Q; otherwise, there is a later such time t." Since <Q,α,q,t> is in the domain of τ_f, <Q,α,q,t"> is also. Thus, by the induction hypothesis, Q_0 = τ_f(Q,α,q,t") is witness equivalent to Q, Q $\sim_{q,t"}$ Q_0, f sends no messages at or after t" so q crashes in Q_0 at t", and an extension of α by q and a link from q satisfies (7) in Q_0. Assume Q is witness equivalent to Q_{k-j}, Q $\sim_{q,t}$ Q_{k-j}, f sends no messages after t in Q_{k-j}, and β satisfies (7) in Q_{k-j} where β is any extension of α by q and a link from q leading to a processor outside the extension. If case c holds, then Q_{k-j+1} is obtained according to step 3 by removing m_j from Q_{k-j} and either no message reaches r_j from q or r_j sends no messages after t in both Q_{k-j} and in Q_{k-j+1}. Thus no processor (excluding r_j) correct in both could distinguish between Q_{k-j} and Q_{k-j+1}. Since <E,s,h,x-h> is an adverse case, there is a processor other than r_j that is correct in both runs; so Q_{k-j} is witness equivalent to Q_{k-j+1}. Also, $Q_{k-j} \sim_{q,t} Q_{k-j+1}$, Path β defined above satisfies (7) for Q_{k-j+1}, and q sends no messages after t in Q_{k-j+1}. If case d holds, then Q_{k-j+1} is determined by steps 4 through 6. In this case β leads to r_j and <Q_{k-j},β,r_j,t+y> is in the domain of τ_f because β has at most h-1 links. According to step 4, $Q_{k-j.1}$ = τ_f(Q_{k-j},β,r_j,t+y). By the induction hypothesis, Q_{k-j} is witness equivalent to

$Q_{k-j.1}$, $Q_{k-j} \sim_{r_j,t+y} Q_{k-j.1}$, r_j sends no messages at or after t+y in $Q_{k-j.1}$, and either β or an extension of β by r_j and a link from r_j to a processor not on the extension satisfies (7) for $Q_{k-j.1}$, depending on whether r_j actually crashes in $Q_{k-j.1}$. Since $Q_{k-j} \sim_{r_j,t+y} Q_{k-j.1}$, f sends no messages after t in $Q_{k-j.1}$. According to step 5, $Q_{k-j.2}$ is obtained from $Q_{k-j.1}$ by removing m_j Thus no processor (excluding r_j) correct in both runs can distinguish between them. Since <E,s,h,x-h> is an adverse case, there is a processor other than r_j correct in both runs. Thus $Q_{k-j.1}$ and $Q_{k-j.2}$ are witness equivalent, $Q_{k-j.1} \sim_{q,t} Q_{k-j.2}$, f sends no messages after t in $Q_{k-j.2}$, and either β or the extension of β defined above satisfies (7) for $Q_{k-j.2}$. If r_j does not crash in $Q_{k-j.2}$, then $Q_{k-j+1} = Q_{k-j}$ according to step 6. Assume r_j crashes in $Q_{k-j.2}$ and let β' and t' be defined according to step 6. Note that t'$\geq$t+y. Then β' satisfies (7) in $Q_{k-j.2}$ and <$Q_{k-j.2}$,β',r_j,t'> is in the domain of τ_c. By the induction hypothesis, $Q_{k-j.2}$ is witness equivalent to $Q_{k-j+1} = \tau_c$(Q"(k-j),β',r_j,t'), Q"(k-j) $\sim_{r_j,t'}$ Q_{k-j+1}, and a prefix of β satisfies (7) in Q(k-j+1). Thus Q is witness equivalent to Q(k-j+1), Q $\sim_{q,t}$ Q(k-j+1), and β satisfies (7) in Q(k-j+1) since q crashes at t in Q(k-j+1). By induction on j (from k to 1), Q is witness equivalent to Q_k =R, Q $\sim_{q,t}$ R, the appropriate extension of α satisfies (7) in R, and q sends no messages at or after t in R.

This completes the proof of the lemma. □

Lemma 13 suffices to prove Theorem 4 as outlined above.□

Note that because of condition (6) we only used crash failures in our proof. Thus Theorem 4 holds for crash failures as well as for omission failures. Moreover, we were careful not to disturb the order in which messages were required to be sent by A; so Theorem 4 holds for orderly crash failures (in which the failing processor cannot send messages out of order).

It is easy to show that a completely connected network of n nodes requires π+1 steps (to tolerate π <n-1 processor omission failures and λ =0 link failures), so Theorem 4 is consistent with the result in [DS].

9.3 A time lower bound for authentication-detectable Byzantine failures

We now move from omission failures to authentication-detectable Byzantine failures and show that our protocols are best possible for the case of an n processor Hamiltonian network that must tolerate n-2 processor authentication-detectable Byzantine failures. A Hamiltonian network is one that has an acyclic path containing all the network nodes. A fully connected network and a 3-dimensional cube are examples of Hamiltonian networks.

Theorem 5 Any atomic broadcast protocol for a Hamiltonian network with n processors that tolerates n-2 authentication-detectable Byzantine processor failures cannot have a termination time smaller than (n-1)($\delta + epsilon$).

Proof: Let G be a Hamiltonian network with processors numbered 0 to n-1 on some acyclic path. Let D be the hypothesized termination time for such an atomic broadcast protocol A and suppose that D<(n-1)($\delta + epsilon$). As in the proof of Theorem 4, let 0 < y < δ, 0 < z < $epsilon$, and D<(n-1)(y+z). Let $\mathcal{R}$. be the set of runs of A in which processor 0 initiates the atomic broadcast of update σ at time 0 which satisfy the following properties: (1) all processor clocks run at the rate of real time, (2) for any j, 0 $\leq$ j $\leq$ n-1: processor j behaves as if $C_j(0) = jz$ (when z is very close to ϵ, at most two processors can be correct since we assume that the clocks of correct processors run within ϵ of each other); (3) all messages sent that are received take exactly y time units, (4) in each run only two processors i-1 and i are correct, 1 $\leq$ j $\leq$ n-1 (we denote by S(i) the run in which processors i-1 and i are correct), (5) each processor j follows A according to the clock C_j, except that if j is not correct in S(i), then j omits to receive or send any messages from or to processors other than j-1 and j+1. Again we assume without loss of generality that any random choices are made in the same way in each run at all processors. By construction, each adjacent pair of runs S(i), S(i+1) 1 $\leq$ i $\leq$ n-2 has identical message histories at the correct processor i they share. Thus, the runs S(i) are witness (and hence output) equivalent. At real time 0 when the atomic broadcast is initiated by processor 0, the clock of processor (n-1) reads (n-1)z. Since the broadcast is timestamped 0, run S(n-1) cannot have processor (n-1) deliver the update. Indeed, since the faulty processors confine any information exchange to be between neighbors j,j+1 only, it takes (n-1)y time units for any information from 0 to reach n-1, and when such information reaches n-1, the termination time D is already past on n-1's clock. Thus in run S(n-1), in which processors n-2 and n-1 are the only ones correct, neither processor delivers σ. An induction on i (using witness equivalence) shows that the same property holds of each run S(i): neither correct processor delivers σ. Thus in run S(1) neither processor 0 nor processor 1 delivers σ, contradicting the termination property of the hypothesized protocol A. □

Therefore, in a Hamiltonian n-node network where the objective is to tolerate up to n-2 authentication-detectable Byzantine failures our third protocol achieves the best possible termination time. For example, according to Theorem 5, in a fully connected network of 4 processors the best possible termination time for handling 2 authentication-detectable Byzantine processor failures is $3(\delta + \epsilon)$ which is identical to the termination time of our second and third protocols, since π =2 and d=1.

10 Conclusion

This paper has specified the atomic broadcast problem, has proposed a classification of the failures observable in distributed systems, has investigated three protocols for atomic broadcast in systems with bounded transmission delays and no partition failures, has proven their correctness and discussed their performance, and has also proved two lower bound theorems which show that, in many cases, these protocols provide the best possible termination times.

Atomic broadcast simplifies the design of distributed fault-tolerant programs by enabling correct processes to access global state information in synchronous replicated storage. This notion reduces the problem of distributed programming to that of "shared storage" programming without having a single point of system failure. In the Highly Available Systems prototype, we used synchronous replicated storage to store crucial system configuration information that must remain available despite (possibly multiple) processor failures.

The three protocols derived share the same specification, have the same diffusion-based structure, but differ in the classes of failures tolerated, ranging from omission failures, to authentication-detectable Byzantine failures. Besides being of pedagogical value for those not familiar with the intricacies of achieving Byzantine agreement, our derivation sheds new light on the continuum that exists between rather simple message diffusion protocols and more complex Byzantine agreement protocols. Clearly, the complexity increases as more failures are tolerated, but the complexity of the final protocol that handles authentication-detectable Byzantine failures is not orders of magnitude greater than that of the initial protocol. A variant of this protocol (which uses error correcting codes to authenticate messages) has been implemented and runs on a prototype system designed by the Highly Available Systems project at the IBM Almaden Research Center [GS]. The experience accumulated during the implementation and test of this prototype showed us that the failures most likely to be observed in distributed systems based on general purpose operating systems such as VM or Unix are performance (or late timing) failures caused by random variations in system load. Since we were aware of the difficulty of debugging distributed protocols (especially when time-dependent), we proved the correctness of ours by using a common diffusion induction principle for all protocols. We believe this proof technique is applicable to many other distributed protocols based on information diffusion.

Given our implementation objective, we have based our protocols on a more realistic system model (i.e. arbitrary network topology, approximately synchronized clocks, unreliable communication links) than previous algorithms for achieving agreement based on the rounds model. Abandoning the rounds model has led to better performance than we obtained by a straightforward conversion of the rounds based protocol in [DS]. Even better performance could be achieved by adopting a clock synchronization approach developed later [Cri] which enables the achievement of synchronization precisions superior to those achievable by algorithms such as those discussed in [CAS], [DHSS], and [Sc].

At the time when our protocols were invented (1983), we were unaware of other protocols for atomic broadcast designed for system models more realistic than those assumed in the Byzantine agreement literature [F], [LSP], [SD]. Since then, several other protocols for atomic broadcast in system models similar to ours have been proposed (e.g. [BJ], [BSD], [Ca], [CM], [D], [GSTC], [PG], [SDC]). All protocols proposed so far can be divided into two classes: time oriented protocols providing bounded termination times even when failures occur during broadcast, and acknowledgement-based protocols that do not provide bounded termination times if failures occur during a broadcast. Examples of protocols in the first class (other than those given in this paper) are [BSD], [GSTC], [PG], and [SDC]. Examples of acknowledgement-based protocols are [BJ], [Ca], [CM], and [D]. While the acknowledgement-based protocols have the potential of tolerating performance failures that can cause network partitioning, diffusion protocols cannot tolerate partition failures. We have investigated methods for detecting and reconciling inconsistencies caused by partitions in systems using diffusion based atomic broadcast (e.g. [SSCA]), but such "optimistic" approaches cannot be used in applications in which there are no natural compensation actions for the actions taken by some processors while their state was inconsistent with the state of other processors. The existence of these two classes of protocols pose a serious dilemma to distributed system designers: either avoid network partitioning by using massive network redundancy and real-time operating systems to guarantee bounded reaction time to events in the presence of failures, or accept partitioning as an unavoidable evil (for example because the operating systems to be used are not hard real-time) and abandon the requirement that a system should provide bounded reaction times to events when failures occur.

11 Acknowledgements

We would like to thank Joe Halpern, Fred Schneider, Mario Schkolnik, Dale Skeen, Irv Traiger, and the referees for a number of useful comments and criticisms. We would also like to thank Nick Littlestone for suggesting the disjunctive form of proposition ϕ in the proof of Theorem 3. This made it possible to use essentially the same proof technique (diffusion induction) for proving the correctness of all our protocols.

12 References

[ADLS] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, "Bounds on the time to reach agreement in the presence of timing uncertainty," Proceedings of ACM Symposium on Theory of Computing, pp. 358-369, 1991.

[BSD] O. Babaoglu, P. Stephenson, R. Drumond: "Reliable Broadcasts and Communication Models: Tradeoffs and Lower Bounds," *Distributed Computing*, No. 2, pp. 177-189, 1988.

[BJ] K. Birman, T. Joseph: "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 47-76, 1984.

[C] F. Cristian, "Correct and Robust Programs," *IEEE Transactions on Software Engineering*, Vol. SE-10, no. 2, pp. 163-174, 1984.

[Ca] R. Carr: "The Tandem Global Update Protocol," *Tandem Systems Review*, pp. 74-85, June 1985.

[CAS] F. Cristian, H. Aghili, and R. Strong, "Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins," *16th International Conference on Fault-Tolerant Computing*, Vienna, Austria, 1986.

[Cr] F. Cristian, "Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System," *18th International Conference on Fault-Tolerant Computing*, Tokyo, Japan, 1988.

[Cri] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing*, Vol. 3, pp. 146-158, 1989.

[CM] J.M. Chang, and N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 251-273, 1984.

[D] "The Delta-4: Overal System Specification," D. Powell, editor, Delta-4 Project Consortium, Bull-SA, BP 208, 38432 Echirolles, France, January 1989.

[DS] D. Dolev, and R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal of Computing*, Vol. 12, No. 4, pp. 656-666, 1983.

[DHSS] D. Dolev, J. Halpern, B. Simons, and R. Strong, "Fault-Tolerant Clock Synchronization," *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 1984.

[F] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems," *Proceedings of the International Conference on Foundations of Computing Theory*, Sweden, 1983.

[GSTC] A. Gopal, R. Strong, S. Toueg, and F. Cristian, "Early-delivery atomic broadcast," Proc. 9th ACM Symp. on Principles of Distributed Computing, pp. 297-309, Quebec City, 1990.

[GS] A. Griefer, and H. R. Strong, "DCF: Distributed Communication with Fault-tolerance," *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988.

[L] L. Lamport, "Using Time instead of Time-outs in Fault-Tolerant Systems," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, pp. 256-280, 1984.

[LSP] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401, July 1982.

[PG] F. Pittelli, H. Garcia-Molina, "Recovery in a Triple Modular Redundant Database System," Technical Report CS-076-87, Princeton University, January, 1987.

[PW] W. Peterson, and E. Weldon, “Error Correction Codes,” (2nd Edition), *MIT Press*, Massachusetts, 1972.

[Po] S. Ponzio, “Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures (Extended abstract),” Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing, 1991.

[RSA] R. Rivest, A. Shamir, and L. Adelman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *CACM*, 21:2, pp. 120-126, 1978.

[Se] A. Segall, “Distributed Network Protocols,” *IEEE Trans. on Information Theory*, IT-29:1, pp. 23-35, 1983.

[Sc] F. Schneider: “Understanding Protocols for Byzantine Clock Synchronization,” Technical report 87-859, Cornell University, August 1987.

[SD] R. Strong, and D. Dolev, “Byzantine Agreement,” *Proceedings of COMPCON*, Spring 1983.

[SDC] R. Strong, D. Dolev, and F. Cristian, “New Latency Bounds for Atomic Broadcast,” Proceedings of the 11th IEEE Real Time Systems Symposium, Orlando, 1990.

[SSCA] R. Strong, D. Skeen, F. Cristian, H. Aghili, “Handshake Protocols,” *7th International Conference on Distributed Computing*, pp. 521-528, September 1987.