# ATOMIC BROADCAST: FROM SIMPLE MESSAGE DIFFUSION TO BYZANTINE AGREEMENT

Flaviu Cristian, Houtan Aghili, Ray Strong
IBM Research Laboratory, San Jose, California 95193

Danny Dolev
Hebrew University, Givat Ram, 91904 Jerusalem, Israel

## ABSTRACT

In loosely coupled distributed systems subject to random communication delays and component failures, *atomic broadcast* protocols can be used to implement the abstraction of a Δ-*common storage*, a replicated storage that displays at any clock time the same contents to every correct processor and that requires Δ time units to complete replicated updates. We term a broadcast protocol atomic if (1) it delivers every message broadcast by a correct sender to all correct receivers within some known time bound (*termination*), (2) it ensures that every message whose broadcast is initiated by a sender is either delivered to all correct receivers or to none of them (*atomicity*), and (3) it guarantees that all delivered messages from all senders are delivered in the same order at all receiving nodes (*order*).

This paper presents a *family* of atomic broadcast protocols that are tolerant of increasingly general fault classes: *omission faults*, which cause a component (processor or communication link) never to deliver a requested service, *timing faults*, which cause a component to deliver a requested service either too early or too late, and *Byzantine faults*, which lead to the delivery of a service other than the one requested. The protocols work for arbitrary point-to-point network topologies, can tolerate any number of faults up to network partitioning, use a small number of messages, and achieve in many cases the best possible termination times.

## INTRODUCTION

Random communication delays and faults prevent distributed processes from having the instantaneous knowledge of the system state that shared storage provides to the processes of a centralized system. This is one of the main reasons behind the perceived difficulty of programming distributed systems. In this paper we discuss broadcast protocols that enable the correct processes of a distributed system to attain consistent (albeit slightly delayed) knowledge of the system state, despite faults and random communication delays. Programming 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 replicate global system state information in all processors, and to broadcast *atomically* every update that a processor makes to this information to all other processors [PSL,L]. We term a broadcast protocol *atomic* if (1) it delivers every message broadcast by a correct sender to all correct receivers after some known time interval (*termination*), (2) it ensures that every message whose broadcast is initiated by a sender is either delivered to all correct receivers or to none of them (*atomicity*), and (3) it guarantees that all messages delivered from all senders are delivered in the same order at all correct receiving nodes (*order*). The atomicity property ensures that every update is either seen by all correct processors or by none of them and the order property ensures that all updates are applied in the same order by all correct processors. Therefore, if their initial views of the system state were consistent, their views of the system state will remain consistent. The termination property ensures that every update performed by a correctly functioning processor is received by all other correct processors after a bounded number of time units, say Δ (for delay). In this way, every correctly functioning processor perceives at any time, the global system state as it was Δ clock time units earlier.

Atomic broadcast protocols can therefore be used to implement the abstraction of a Δ-*common storage*: a replicated, resilient storage that displays at any clock time the same contents to every correct processor and that requires Δ time units to complete replicated updates. The use of such an abstract storage can considerably 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. Moreover, it is relatively straightforward to adapt known parallel programming paradigms for shared storage environments (where parallel processes access the same storage state at any real time) to loosely coupled processor environments providing the abstraction of a Δ-common storage (where all distributed processes access the same Δ-common storage state at any clock time). Several examples of such adaptations are given in [L]. Within the Highly Available System project [A], the primary application of atomic broadcast is for updating replicated system directories and reaching agreement on the failure and recovery of system components.

This paper presents a new family of atomic broadcast protocols from a very simple protocol that only tolerates omission faults to a rather sophisticated protocol that tolerates any type of faults. Naturally, the more types of faults a protocol can tolerate, the more complex and expensive it is to build and run. Our objective is to provide, for all considered fault classes, protocols that are *cost effective* for those classes in terms of message traffic, speed, and complexity.

Despite their simplicity, the presented protocols have interesting performance characteristics. We prove that in many cases our protocols achieve the best possible termination times. We also show that our protocols are efficient in terms of message traffic. Depending on network topology, the number of physical messages sent per broadcast for every considered fault class is between 1 and 2 messages per communication link.

## FAULT CLASSIFICATION

Let $\mathcal{P}$ be the set of *processors* of a distributed system and $\mathcal{L}$ the set of physical *links* among processors in $\mathcal{P}$. The components of the system (e.g. processors, links, processor clocks) undergo state transitions in response to the occurrence of certain trigger events. For components such as processors or links, these events are essentially service request arrivals. For example, a link $l \in \mathcal{L}$ connecting processor $s \in \mathcal{P}$ to processor $r \in \mathcal{P}$ delivers a message to $r$ if $s$ so requests; a processor $p \in \mathcal{P}$ computes a given function if a request is generated by a user. In contrast to such "demand driven" components, clocks are "autonomous" components, in that they undergo state transitions simply in response to the passage of time. For example, a clock passes from a state i to the state i+1 if the event "a time unit has elapsed since the last state change" occurs.

A system component is termed *correct* if, in response to trigger event occurrences, it behaves in a manner consistent with its specification. Component specifications prescribe the state transitions that should occur in response to such events and the real time interval within which these state transitions should occur. A component *failure* occurs when a component does not behave in the manner specified. The cause of a failure is a *fault*. We distinguish among the following fault classes.

A fault that causes a component never to respond to the occurrence of a trigger event is an *omission* fault. A fault that causes a component to respond either too early or too late, i.e., outside the real time interval specified, is a *timing* fault. A fault that causes a component to respond in a manner different from the one specified is a *Byzantine* fault. This

Reprinted from *FTCS 15 1985*, pp.200-206, © IEEE
0-8186-7150-5/96 $5.00 © 1996 IEEE
*Proceedings of FTCS-25, Volume III*

fault classification refines an earlier proposal [MSF], where the term "fault of commission" was used to cover both timing faults and Byzantine faults. A further interesting refinement (not used in this paper) is to distinguish between *early* and *late* (or *performance*) timing faults. Also, observe that the term "failstop", used in [SGS] for components that either behave correctly or stop functioning, corresponds to a proper subset of the set of omission faults, that omission faults are a proper subset of timing faults, and that timing faults are a proper subset of Byzantine faults.

A processor crash, a clock that has stopped running, a link breakdown, a processor which occasionally does not forward a message that it should, and a link that sometimes loses a message, are examples of omission faults. Occasional message delays caused by overloaded relaying processors, and a clock that is slow are examples of late (or performance) timing faults. A fast clock is an example of an early timing fault. An undetectable message corruption on a link (because of electromagnetic noise or because of human sabotage) is an example of a Byzantine fault.

Observe that a fault cannot be classified without reference to a component specification. In particular, if one component is made up of others, then a fault of one type in one of the constituent components can lead to a fault of *another* type in the larger component. For example, a stopped clock that constantly displays the same "time" is an example of an omission fault. If that clock is part of a processor that sends messages with the same timestamp as a result, then the processor may be classed as experiencing a Byzantine fault.

In the remainder of this paper we consider a decomposition of a distributed system into components that are either processors or links. Neither type of component may be considered part of the other. With this convention we can classify faults unambiguously. We are not concerned with tolerating or handling the faults 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 faults, we mean tolerating omission faults on the part of either processors or links, not tolerating omission faults on the part of sub-components like clocks that might cause much worse behavior on the part of their containing processors.

Note that in the above fault classification, we have defined faults with respect to single trigger events. Traditionally, however, faults have been classified into *transient* and *permanent* faults with regard to trigger event sequences. The algorithms to be presented tolerate both transient and permanent faults, provided the set of faults that occur during a broadcast does not cause network partitioning.

## MODEL AND ASSUMPTIONS

Let $P=\{1,2,...,n\}$ be a set of n distributed *processes* which run on a set of n distinct processors $\mathscr{P}=\{1,2,...,n\}$ and broadcast information atomically to implement the abstraction of a $\Delta$-common storage. In what follows, we refer to the processes in $P$ as $\Delta$-common storage managers. Processor names are written using italics (e.g., $p,q,r$) while process names are written using ordinary font letters (e.g., p,q,r). We denote by $p \in P$ the $\Delta$-common storage manager which runs on processor $p \in \mathscr{P}$.

We make the following assumptions:

1. The physical communication network that interconnects the processors is a *point-to-point* network. Unlike most previous work on Byzantine Agreement [SD,F] we do not assume that the communication network is fully connected (either physically or logically - via a network routing layer).

2. Let $\mathscr{P}_f \subset \mathscr{P}$ and $\mathscr{L}_f \subset \mathscr{L}$ be the processors and links that experience faults during a broadcast, and let $(\mathscr{P}_c,\mathscr{L}_c)$ be the surviving network obtained after removing the faulty components $\mathscr{P}_f$ and $\mathscr{L}_f$ from the original network $(\mathscr{P},\mathscr{L})$. We assume that the processors $\mathscr{P}_f$ and links $\mathscr{L}_f$ that are faulty during a broadcast leave the surviving system $(\mathscr{P}_c,\mathscr{L}_c)$ *connected*. In what follows $P_c$ denotes the set of processes that are running on the (correct) processors in $\mathscr{P}_c$.

3. The network transmission delay is bounded. Let $s,r \in P_c$ be two processes running on two correct processors $s,r$ which are connected by a correct link $l \in \mathscr{L}_c$. A message m sent from process s to process r over $l$ is delivered to r and is processed by r in at most $\delta$ time units as measured on the clock of either $s$ or $r$. If m takes more than $\delta$ time units to be delivered to, and be processed, by r, or is never delivered to r, or is delivered corrupted, then at least one of the processors $s$, $r$, or the link connecting them is faulty. We distinguish between *reception* of a message by a processor $r$ and *delivery* of that message to the process r responsible for managing the local $\Delta$-common storage copy on $r$.

4. The clocks of correct processors measure the passage of time accurately, and are synchronized, so that the measurable difference between the readings of correct clocks at any instant is bounded by a known constant $\epsilon$. Specifically, if a correct process p sends a message m at its local time $t_p$ to another correct process q via h-1 intermediate correct processors (over h correct links), and the message is delivered to q at time $t_q$ on $q$'s clock, then the following holds:

$$-\epsilon \leq t_q - t_p \leq \epsilon + h\delta.$$

(For clock synchronization algorithms under various fault hypotheses see [CAS,DHSS].)

5. If two tasks $T_1$ and $T_2$ are scheduled for execution at times $t_1$ and $t_2$ such that $t_1<t_2$, then $T_1$ is executed before $T_2$. We assume that no correct processor issues the same timestamp twice, i.e., that the granularity of time measurement is fine enough to discriminate between separate task invocations.

We denote $d_{p,q}(\mathscr{P}_f, \mathscr{L}_f)$ the distance (i.e., the length of the shortest path) between processors $p$ and $q$ in the surviving network $(\mathscr{P}_c,\mathscr{L}_c)$. The greatest distance among any two processors in the surviving network is then:

$$d(\mathscr{P}_f,\mathscr{L}_f) \equiv \max\{d_{p,q}(\mathscr{P}_f,\mathscr{L}_f) \mid p,q \in \mathscr{P}_c\}.$$

From the above definitions and assumption 3 it follows that the worst case message delay (along a shortest path) between any two correct processors in the presence of at most $\pi$ processor faults and $\lambda$ link faults that do not disconnect the network is given by

$$D_{\pi,\lambda} \equiv \delta d_{\pi,\lambda},$$

where

$$d_{\pi,\lambda} \equiv \max\{d(\mathscr{P}_f,\mathscr{L}_f) \mid |\mathscr{P}_f| \leq \pi \ \& \ |\mathscr{L}_f| \leq \lambda \ \& \ (\mathscr{P}_c,\mathscr{L}_c) \textit{ connected}\}.$$

The important problems of detecting and isolating faulty system components, of detecting the occurrence of network partitions, and of joining isolated or repaired components (processors or links) to existing systems are beyond the scope of this paper. These problems are studied [S] and satisfactory solutions to some of them do exist (see for example the treatment of the join problem in clock synchronization under various fault hypotheses in [CAS] and [DHSS]). Discussions of the $\Delta$-common storage join problem and of partition detection under various fault assumptions is postponed to future reports.

## OBJECTIVES

Our goal is to present broadcast protocols that, for arbitrary (but fixed) numbers $\pi$ and $\lambda$ of processor and link faults that do not disconnect the communication network, and for some $\Delta$ to be determined, possess

the following properties:

- Every message broadcast by a correct process $s \in P_c$ at time t on $s$'s clock is delivered to every correct process $q \in P_c$ at time $t+\Delta$ on $q$'s clock (*termination*);
- Any message broadcast at clock time t is either delivered to every correct process $q \in P_c$ (exactly once) at time $t+\Delta$ on $q$'s clock, or is never delivered to any correct process (*atomicity*).
- All delivered messages are delivered in the same order to all correct processes (*order*).

We consider three properly nested classes of faults: (1) omission faults, (2) timing faults, and (3) Byzantine faults (that is, any type of faults). For each of these three classes we present a fault-tolerant broadcast protocol that possesses the above properties and can tolerate any number of faults in that class, provided the network of correct components remains connected.

All protocols to be presented are based on a simple *information diffusion* technique: (1) a correct sender sends a message on all its (outgoing) links, and (2) when a new message is received on some (incoming) link by a correct processor, the processor forwards that message on all the other (outgoing) links. The difficulty, as we shall see, is in distinguishing between "new" and "old" messages, and between messages sent by correct processors and those sent or forwarded by faulty processors or links.

We refer to a set of programs responsible for implementing an atomic broadcast protocol as a *broadcast layer*. There is a copy of this layer in each processor $p \in \mathcal{P}$ of the network. We structure each such copy into three parallel tasks: a *Start* broadcast task, that initiates the broadcast of a $\Delta$-common storage update $\sigma$ by sending messages on all outgoing links, a *Relay* task, that forwards such messages to adjacent processors, and an *End* task, that delivers these messages to the $\Delta$-common storage managers $p \in P$.

A $\Delta$-common storage manager p on a processor $p \in P$ manages the local copy $\delta cs_p$ of a replicated $\Delta$-common storage $\delta cs$ by invoking routines implemented by the atomic broadcast layer. Similarly the reading or writing of $\delta cs_p$ by higher level processes r is performed by invoking two "read" and "write" routines provided by the local $\Delta$-common storage manager p. Assume that any reader task r has a lower priority than the End or p tasks, so that, before any examination of $\delta cs_p$ within the read routine, a chance is given to the local End task to deliver all pending $\delta cs$ updates to p. Then the termination, atomicity and order properties make it possible to implement a read $\Delta$-common storage routine which possess the following property: for any concurrent execution of the programs $P_1$ and $P_2$ below, on any two processors $p_1$ and $p_2$, if $t_1 = t_2$ holds before reading a local $\Delta$-common storage copy, then $a_1 = a_2$ is true after the read terminates.

$P_1 \equiv t_1 \leftarrow$ clock; $a_1 \leftarrow$ read($\delta cs_1, t_1$);

$P_2 \equiv t_2 \leftarrow$ clock; $a_2 \leftarrow$ read($\delta cs_2, t_2$);

Remark: For reasons of brevity no proofs are given for the theorems included in this extended abstract. The complete proofs for all theorems are included in an extended version [CASD].

## FIRST PROTOCOL: TOLERANCE OF OMISSION FAULTS

We begin by presenting a simple protocol that can tolerate arbitrary (but fixed) numbers $\pi$ and $\lambda$ of processor and link omission faults, provided these faults do not partition the surviving communication network. The protocol is based on the following basic observations:

1. To achieve the order property it is sufficient that in every correct processor messages be delivered in the order of their generation times (or timestamps), and that messages generated at the same clock time be delivered in increasing order of their sender's identifier.

2. To ensure that any message broadcast at clock time t by some processor $s$ and received by at least one correct processor $r \in \mathcal{P}_c$ is received by every correct processor $q \in \mathcal{P}_c$ before $q$ *delivers* any message broadcast at clock time t, it is sufficient for every correct processor $q$ to deliver messages generated at clock time t when $q$'s clock displays the time

$$t + \pi\delta + D_{\pi,\lambda} + \epsilon.$$

The time $\pi\delta$ is the worst case delay between the broadcast initiation and the message acceptance by a first correct processor $r$ (corresponding to the case when every processor between $s$ and $r$, including $s$, is faulty and forwards just one message on one outgoing link). The time $D_{\pi,\lambda}$ is sufficient for diffusing the message from $r$ to all other correct processors (note that any upper bound of the $D_{\pi,\lambda}$ diffusion time, e.g. $(n-\pi-1)\delta$, could be used). The last term, $\epsilon$, accounts for the uncertainty existing on clock synchrony. The *termination time* $\Delta$ of the protocol can, therefore, be given as:

$$\Delta \equiv \pi\delta + D_{\pi,\lambda} + \epsilon.$$

### The Start Task

A high level description of this task is given in Figure 1. A sender process $s \in P$ triggers the broadcast of a value $\sigma$ of some type $\Sigma$ by sending it to the local Start task. The command "take" is used by this task to (locally) receive $\sigma$ from s (line 4). The broadcast of $\sigma$ is identified by

```
1.  task Start;
2.  const Δ = πδ + D_{π,λ} + ε;
3.  var t: Time; σ:Σ; s: P;
4.  cycle   take(σ);
5.          t ← clock;
6.          s ← myid;
7.          send-all(t,s,σ);
8.          H ← H⊕(t,s,σ);
9.          schedule(End,t+Δ,t);
10. endcycle;
```

**Figure 1: Start task of the first protocol**

the local time t at which $\sigma$ is received by the Start task (line 5) and the identity s of the sender, which is obtained by invoking the function "myid" (line 6). Two broadcast identifiers $(t_1,s_1)$ and $(t_2,s_2)$ are *equal* only if $t_1 = t_2$ and $s_1 = s_2$. The processor identifiers returned by the function "myid" on distinct processors are guaranteed to be distinct. So by assumption 5, every correct processor generates broadcast identifiers that are *unique* system-wide.

The broadcast of $\sigma$ is initiated by invoking the "send-all" command which sends messages of the form $(t,s,\sigma)$ on all outgoing links of $s \in \mathcal{P}$, one at a time (line 7). The fact that a broadcast (t,s) of a value $\sigma$ has been initiated is then recorded in a local *history* variable H shared by all broadcast layer tasks of s:

*var* H: *Time* $\rightarrow$ ($P \rightarrow \Sigma$).

We assume that this variable is initialized to the totally undefined function at processor start. The variable H keeps track of ongoing broadcasts by associating with every instant t a function H(t) (of type $P \rightarrow \Sigma$) which records values $\sigma \in \Sigma$ broadcast by processes $p \in P$ at clock time t. That is, the domain of H(t) consists of processes $p \in P$ that have initiated broadcasts at time t on their clocks, and for each such process p, H(t) records the value H(t)(p) broadcast by p at time t.

We define the following two operators on histories. The *update* "$\oplus$" of a history H by a message $(t,s,\sigma)$ yields a (longer) history, denoted $H\oplus(t,s,\sigma)$, that contains all the facts known in H, plus the fact that s has broadcast $\sigma$ at local time t. Formally:

$$dom(H\oplus(t,s,\sigma)) = dom(H) \cup \{t\},$$
$$dom((H\oplus(t,s,\sigma))(t)) = dom(H(t)) \cup \{s\},$$
$$\forall \tau \in dom(H), \tau \neq t, \quad (H\oplus(t,s,\sigma))(\tau) = H(\tau),$$
$$\forall p \in dom((H\oplus(t,s,\sigma))(t)),$$
$$((H\oplus(t,s,\sigma))(t))(p) = \begin{cases} \sigma \text{ if } p = s, \\ H(t)(p) \text{ otherwise.} \end{cases}$$

The *deletion* "\" of some instant t from a history H yields a (shorter) history, denoted $(H\backslash t)$, which removes from the history everything known about the broadcasts initiated at time t:

$$dom(H\backslash t) \equiv dom(H) - \{t\},$$
$$\forall x \in dom(H\backslash t), \quad (H\backslash t)(x) = H(x).$$

Once the history H is updated with the information $(t,s,\sigma)$ as shown in line 8, the End task is scheduled to start at local clock time $t+\Delta$ at which time the value $\sigma$ will be delivered. The first parameter of the "schedule" command (line 9) is the name of the task to be started, the second is the local time at which the task should be started, and the third is a value that will be passed to the task when it starts (as an input parameter).

### The Relay Task

A high level description of the Relay task is given in Figure 2 (in this description, double quotes delimit comments, and the meaning of the

```
1.  task Relay;

2.  const Δ = πδ + D_{π,λ} + ε;
3.  var t,τ: Time; σ:Σ; s: P;

4.  cycle receive(t,s,σ,l);
5.      τ ←clock;
6.      [τ > t + Δ: "too late" iterate];
7.      [t∈dom(H) & s∈dom(H(t)): "deja vu" iterate];
8.      send-all-but(l,(t,s,σ));
9.      H←H⊕(t,s,σ);
10.     schedule(End,t+Δ,t);
11. endcycle;
```

**Figure 2: Relay task of the first protocol**

syntactic construct "[B: *iterate*]" is "if Boolean condition B is true, then terminate the current iteration and begin the next iteration" {C]). The Relay task uses the command "receive" to *receive* messages formatted as $(t,s,\sigma)$ from adjacent processors (line 4). After such a message is received, the parameter l contains the identity of the link over which the message arrived. If the message is not too late (line 6), and is new (that is, its identifier (t,s) is not in the history variable H, line 7), then the message is *accepted*. Otherwise, the message is *discarded*.

Once a message is accepted, (1) it is relayed on all outgoing links except l using the command "send-all-but" (line 8), (2) the history variable H is updated with the information that "s has broadcast $\sigma$ at time t" (line 9), and (3) the End task is scheduled to start at local time $t+\Delta$ when the received value will be delivered (line 10).

### The End Task

The End task (Figure 3) starts at clock time $t+\Delta$ to deliver values broadcast at clock time t. An activation of End with argument t also deletes from the history H everything known about broadcasts initiated at time t so that H does not grow infinitely large. All values broadcast by correct processes $p \in dom(H(t))$ at clock time t are delivered to the local target processes (the loop of lines 5-8) by invoking the "deliver" command (line 6). A value broadcast by a process $s_1$ at time t is delivered before the value broadcast by another process $s_2$ at time t if $s_1 < s_2$.

```
1.  task End(t: Time);

2.  var p: P; val: P → Σ;

3.  val←H(t);
4.  while dom(val)≠ ø;
5.  do p←min(dom(val));
6.      deliver(val(p));
7.      val←val\p;
8.  od;
9.  H←H\t;
```

**Figure 3: End task of the first protocol**

### Correctness

The assumption that there are at most $\pi$ processor and $\lambda$ link omission faults, together with assumptions 2, 3, 4, and 5 lead to the following:

*Lemma 1:* If a message timestamped $t_s$ is inserted in the history of at least one correct processor, then all correct processors insert the message in their history by time $t_s+\Delta$ on their clocks.

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

*Theorem 2:* The first protocol is not tolerant of timing faults.

## SECOND PROTOCOL: TOLERANCE OF TIMING FAULTS

The first protocol cannot tolerate timing faults because there is a fixed clock time interval during which a message is unconditionally accepted by a processor. This creates a real time "window" during which a message might be "too late" for some (early) processors and "in time" for other (late) processors. To achieve atomicity in the presence of timing faults, a timeliness check must ensure that if a first correct processor $p$ accepts a message, then all other correct processors to which $p$ relays that message also accept the message. We attain this property by making the size of the time interval during which a message is acceptable *proportional* to the number of processors that have accepted it. That is, if at (local clock) time $\tau$ a processor receives a message which is timestamped t and was relayed over h intermediate links, then it will accept that message only if:

$$t - h\epsilon \leq \tau \leq t + h(\delta + \epsilon).$$

In this way, 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 $D_{\pi,\lambda}$ time units to reach all other correct processors. Given the $\epsilon$ uncertainty on clock synchrony (see Remark 1) the termination time of the second protocol is therefore:

$$\Delta \equiv \pi(\delta + \epsilon) + D_{\pi,\lambda} + \epsilon.$$

### The Start Task

The start task of the second protocol is identical to that of the first except for the addition of a hop count $\kappa$ initially set to 1 (line 6 of Figure 4).

### The Relay Task

The Relay task of the second protocol is given in Figure 5. In addition to the tests necessary for providing tolerance of omission faults (lines

```
1.  task Start;

2.  const Δ = π(δ + ε) + D_{π,λ} + ε;
3.  var t: Time; σ:Σ; s: P; κ : 1 ... n;

4.  cycle take(σ);
5.          t←clock;
6.          κ ← 1;
7.          s←myid;
8.          send-all(t,s,κ,σ);
9.          H←H⊕(t,s,σ);
10.         schedule(End,t+Δ,t);
11. endcycle;
```

**Figure 4: Start task of the second protocol**

```
1.  task Relay;

2.  const Δ = π(δ + ε) + D_{π,λ} + ε;
3.  var t,τ: Time; σ:Σ; l: link; κ: 1 ... n;

4.  cycle receive(t,s,κ,σ,l);
5.      τ ←clock;
6.      [τ < t − εκ: "too early" iterate];
7.      [τ > t + κ(ε + δ): "too late" iterate];
8.      [τ > t + Δ: "too late" iterate];
9.      [t∈dom(H) & s∈dom(H(t)): "deja vu" iterate];
10.     send-all-but(l,(t,s,κ + 1,σ));
11.     H←H⊕(t,s,σ);
12.     schedule(End,t+Δ,t);
13.  endcycle;
```

**Figure 5: Relay task of the second protocol**

8 and 9 in Figure 2), the Relay task of the second protocol also contains the timeliness tests discussed above (lines 6 and 7). The hop count $\kappa$ is incremented (line 10) every time a message is relayed.

### The End Task

The End task of the second protocol is identical to that of the first protocol given in Figure 3.

### Correctness

Under the assumptions that there are at most $\pi$ processor faults and $\lambda$ link faults, that the faults do not disconnect the network, and that all faults are timing faults, we have:

*Lemma 2:* If a message timestamped $t_s$ is inserted in the history of at least one correct process, then all correct processes r insert the message in their history by time $t+\Delta$ on their clocks.

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

*Theorem 4:* The second protocol is not tolerant of Byzantine faults.

## THIRD PROTOCOL: TOLERANCE OF ANY TYPE OF FAULTS

A processor affected by a Byzantine fault 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 from happening is to authenticate the messages exchanged by processors during a broadcast [PSL,DS]. The objective of message authentication is to ensure that a faulty system component cannot forward an altered message to a correct processor without that processor being able to detect the change.

We assume that each processor $p$ has a *signature* [DS] so that it is highly improbable that $p$'s signature can be generated by a processor other than $p$. We also assume that every processor $q$ has access to an *authentication predicate* in order to verify the *authenticity* of a signature with a high probability. Methods for designing such signature and authentication schemes are discussed in [PW,RSA]. We implement message authentication by means of three procedures "sign", "co-sign", and "authenticate", and by using a new data type "smsg" (signed message). The "sign" procedure is invoked by the original sender $s \in \mathcal{P}$ of a message m to sign that message. The "co-sign" procedure is invoked by a processor $r \in \mathcal{P}$ forwarding m to append that $r$'s signature to the list of signatures on m. The "authenticate" procedure verifies the authenticity of an incoming message m. If no alteration of m's contents is detectable, then it returns the original timestamp and value sent by $s$ as well as a sequence S containing the identity of all processors that have signed m. The identity of the sender 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 $|S|$. If a message is determined to be corrupted, the "authenticate" procedure signals the "forged" exception (the exception mechanism of [C] is assumed). An implementation of the above procedures is given in [CASD].

By using message authentication, messages corrupted by a component (processor or link) affected by a Byzantine fault can be detected and discarded by every correct adjacent processor. In this way, a Byzantine fault occurrence in a component has the same effect as a failure of that component to relay messages to the set of correct processors adjacent to it. Therefore, if authentication processing time is ignored, the termination time of the third protocol is the same as the termination time of the second protocol:

$$\Delta \equiv \pi(\delta + \epsilon) + D_{\pi,\lambda} + \epsilon.$$

### The Start Task

Except for the change concerning the authentication of the messages, the structure of this task (Figure 6) is similar to that of the second protocol.

```
1.  task Start;

2.  const Δ = π(δ + ε) + D_{π,λ} + ε;
3.  var t: Time; σ:Σ; s: P; x: smsg;

4.  cycle take(σ);
5.          t←clock;
6.          s←myid;
7.          sign(t,σ,x);
8.          send-all(x);
9.          H←H⊕(t,s,σ);
10.         schedule(End,t+Δ,t);
11. endcycle;
```

**Figure 6: Start task of the third protocol**

### The Relay task

In order to handle the case in which a faulty processor broadcasts several values with the same timestamp, the type of the history variable H is changed to

$$\textit{var}\ \ H: \textit{Time} \rightarrow (P \rightarrow (\Sigma \cup \{\perp\})),$$

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

```
1.  task Relay;

2.  const Δ = π(δ + ε) + D_{π,λ} + ε;
3.  var t,τ: Time; σ:Σ; l: link; s: P;
3.  var S: Sequence-of-P; x,y: smsg;

4.  cycle receive(x,l);
5.     τ ←clock;
6.     authenticate(x,t,σ,S)[forged: iterate];
7.     [duplicates(S): "duplicates" iterate];
8.     [τ < t − ε|S|: "too early" iterate];
9.     [τ > t + (ε + δ)|S|: "too late" iterate];
10.    [τ > t + Δ: "too late" iterate];
11.    s← first(S);

12.    if t∈dom(H) & s∈dom(H(t)) then
13.       [σ=H(t)(s): "deja vu" iterate];
14.       {H(t)(s)=⊥: "known faulty sender" iterate];
15.       H(t)(s)←⊥;
16.    else
17.       H←H⊕(t,s,σ);
18.       schedule(End,t+Δ,t);
19. fi;

20. cosign(x,y);
21. send-all-but(l,y);
22. endcycle;
```

**Figure 7: Relay task of the third protocol**

The Relay task (Figure 7) of the third protocol works as follows. Upon receipt of a message (line 4), the message is checked for authenticity (line 6). If the message is found to be corrupted, either by a faulty processor or by a faulty link, it is discarded. 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 (line 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 (lines 8, 9, and 10). No confusions such as those illustrated in the proof of Theorem 4 can occur unless the authentication scheme is compromised (see Appendix A for more details on the probability of successful forgery).

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 (line 12). If the received value belongs to a new broadcast, then (1) the history variable H is updated with the information that the sender s=*first*(S) has sent a new value $\sigma$ at time t (line 17), (2) the End task is scheduled to start processing and possibly delivering the received message at (local clock) time $t+\Delta$ (line 18), and (3) the received message is co-signed and forwarded (lines 20 and 21).

If the received value has already been recorded in H (because it was received via an alternate path), then it is discarded (line 13). If the received value is a second value for a broadcast identified (t,s), then the sender must be faulty. This fact is recorded by setting H(t)(s) to the null value (line 15). The message is then co-signed and forwarded so that other correct processors also note the fault (lines 20 and 21). Finally, if the received value is associated with a broadcast identifier to which H has already associated the null value (i.e., it is already determined that the originator of the broadcast (t,s) is faulty), then the received value is discarded (line 14).

### The End task

The End task (Figure 8) is started on every correct processor at (local clock) time $t+\Delta$ for delivering the values broadcast correctly at clock time t. If exactly one value has been received for a broadcast initiated at clock time t, then that value is delivered (line 7). If, on the other hand, at least two values have been received for a broadcast, then the sender must be faulty and no value is delivered. In either case, the values associated with broadcasts (s,t), for all $s \in P$, are deleted from the history (line 11).

```
1.  task End(t: Time);

2.  var p: P; val: P → (Σ ∪ {⊥});

3.  val←H(t);
4.  while dom(val)≠ ø;
5.  do p←min(dom(val));
6.     if val(p)≠⊥ then
7.        deliver(val(p));
8.     fi;
9.     val←val\p;
10. od;
11. H←H\t;
```

**Figure 8: End task of the third protocol**

### Correctness

Under the assumptions that there are at most $\pi$ processor faults and $\lambda$ link faults that do not disconnect the network, and that the faults do not result in a failure of the authentication protocol (e.g., accidental production of a correct signature), the following hold:

*Lemma 3:* If a value $\sigma$ timestamped t is inserted in the history of at least one correct processor, and there does not exist a correct processor which has set its H(t)(s) to the null value by time $t+\Delta$ on its clock, then all correct processors insert $\sigma$ in their histories and no correct processor sets its H(t)(s) to the null value by local time $t+\Delta$.

*Theorem 5:* The third protocol possesses the termination, atomicity, and order properties.

## PERFORMANCE

In this section we investigate the performance of our protocols in terms of message traffic and termination time.

In the absence of faults, a processor $s \in \mathcal{P}$ that initiates a broadcast sends $d_s$ messages, where $d_s$ is the degree of $s$, that is, the number of its adjacent links. Every other processor $r \in \mathcal{P}$ that forwards this broadcast sends $d_r$-1 messages since no message is ever sent back on the link on which it arrived. Thus, the total number of messages sent for a broadcast in the absence of faults by any of the three protocols presented is

$$d_s + \sum_{\substack{r \in \mathcal{P} \\ r \neq s}} (d_r - 1) = \sum_{r \in \mathcal{P}} d_r - (|\mathcal{P}| - 1) = 2|\mathcal{L}| - (|\mathcal{P}| - 1).$$

The average number of messages $\bar{m}$ sent per broadcast per communication link is

$$\bar{m} = 2 - \frac{|\mathcal{P}| - 1}{|\mathcal{L}|}.$$

For sparse networks the value of $\bar{m}$ is close to 1, whereas for highly connected ones it is close to 2. It is always, however, strictly less than 2.

Another important property of an algorithm is its termination time. The following theorem establishes a *lower bound* for the termination time of any atomic broadcast protocol.

*Theorem 6:* Let $\alpha$ be the number of processors on the longest path in a network such that the removal of those processors from the network

(1) does not disconnect the network, and (2) leaves at least 2 processors in that network. An atomic broadcast algorithm tolerant of $\beta$ *failstop* processor faults cannot have a termination time smaller than $\epsilon + \max(\delta(\pi+1), D_{0,0})$, where $\pi \equiv \min(\alpha,\beta)$ and $D_{0,0}$ is the (original) network diffusion time.

In the above theorem, by a path of length $k \geq 2$, we mean an ordered sequence of processors $p_1,...,p_k$ such that (1) all processors in the sequence are distinct, and (2) every pair of processors $p_i$ and $p_{i+1}$, i=1,...,k-1, are adjacent. A path of length 1 is a sequence containing a single processor.

For example, in a network of 8 processors arranged in a 3 dimensional cube, the termination time of *any* correct atomic broadcast algorithm tolerant of 2 processor faults is at least $3\delta + \epsilon$ ($\alpha = 6$, and $\beta = 2$). On the other hand, in order to tolerate 2 omission faults (recall that the class of omission faults include failstop faults) our first protocol terminates in

$$\Delta = 2\delta + D_{2,0} + \epsilon = 5\delta + \epsilon$$

clock time units. That is, for this example, our protocol is within $2\delta$ time units of the provable minimum. For reasonable values of $\delta$, this difference is insignificant. In some cases, however, we can show that our algorithms achieve the best possible termination time:

*Theorem 7:* Any atomic broadcast algorithm for a fully connected network of $n$ processors that tolerates $(n-2)$ *Byzantine* processor faults cannot have a termination time smaller than $(n-1)(\delta+\epsilon)$.

Therefore, in a fully connected network where the objective is to tolerate as many faulty processors as possible (up to network partition), our third algorithm achieves the best possible termination time. For example, according to Theorem 7, in a fully connected network of 4 processors the best possible termination time for handling 2 Byzantine processor faults is $3(\delta+\epsilon)$ which is identical to the termination time of our second and third protocols:

$$\Delta = 2(\delta+\epsilon) + D_{2,0} + \epsilon = 3(\delta+\epsilon).$$

## CONCLUSION

A new family of protocols for atomic broadcast was presented. All protocols share the same specification, but differ in the classes of faults tolerated, ranging from omission faults, to timing faults, and Byzantine faults. Unlike other known atomic broadcast protocols tolerant of Byzantine faults [BD,F,SD] our protocols do not rely on such strong assumptions as exactly synchronized clocks, perfect communication or totally connected networks. On the other hand, unlike other known reliable broadcast protocols based on more realistic hypotheses [CM,SGS,LL], we provide tolerance of timing and Byzantine faults.

For each of the fault classes considered, we derived atomic broadcast protocols which are cost-effective in terms of message traffic, speed, and complexity. All protocols have essentially the same structure independent of the fault classes they tolerate. This sheds new light on the unity which exists between simple diffusion protocols and sophisticated Byzantine Agreement protocols. Clearly, the complexity increases as an algorithm tolerates more complex faults, but the complexity of the final algorithm that handles arbitrary faults is not orders of magnitude greater than that of the initial algorithm. The third algorithm has been implemented and runs on a prototype system designed by the Highly Available Systems project at IBM San Jose Research Laboratory.

We leave open the question of the exact lower bounds on the termination time required for atomic broadcast as a function of the network topology. However, we believe that in practice our algorithms come sufficiently close to optimal performance to make it hard to justify any additional complexity to achieve the absolutely best possible time.

## ACKNOWLEDGEMENTS

We would like to thank Shel Finkelstein, Joe Halpern, Mario Schkolnick, Dale Skeen, and Irv Traiger for a number of useful comments and criticisms.

## BIBLIOGRAPHY

[A] H. Aghili, W. Kim, J. McPherson, M. Schkolnick, and R. Strong, *A Highly Available Database System,* COMPCON Spring 83 (1983).

[BD] O. Babaoglu, and R. Drummond, *Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts,* Technical Report TR 84-613, Department of Computer Science, Cornell University (1984).

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

[CAS] F. Cristian, H. Aghili, and R. Strong, *Approximate Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins,* Research Report in preparation (1985).

[CASD] F. Cristian, H. Aghili, R. Strong and D. Dolev, *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement,* Research Report in preparation (1985).

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

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

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

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

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

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

[LL] G. Le Lann, *A Distributed System for Transaction Processing,* IEEE Computer **Vol. 14, No. 4** (1981) pp. 43-48.

[MSF] C. Mohan, R. Strong, and S. Finkelstein, *Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement within Clusters of Processors,* Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (1983).

[PSL] M. Pease, L. Lamport, and R. Shostak, *Reaching Agreement in the Presence of Faults,* Journal of ACM **Vol. 27, No. 2** (1980) pp. 228-234.

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

[RSA] R. Rivest, A. Shamir, and L. Adelman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,* CACM **Vol 21. No. 2** (1978) pp. 120-126.

[S] R. Strong, *Problems in Fault Tolerant Distributed Systems,* Proceedings of COMPCON Spring 85 (San Francisco, 1985).

[SD] R. Strong, and D. Dolev, *Byzantine Agreement,* Proceedings of COMPCON Spring 83 (San Francisco, 1983).

[SGS] F. Schneider, D. Gries, R. Schlichting, *Fault-Tolerant Broadcast,* Science of Computer Programming **Vol. 4, No. 1** (1984) pp. 1-15.