An Optimal Atomic Broadcast Protocol and an Implementation Framework

Paul EZHILCHELVAN† Doug PALMER† Michel RAYNAL⋆

† Dept of Computing Science, University of Newcastle, NE1 7RU, UK
⋆ IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
paul.ezhilchelvan,d.j.palmer@newcastle.ac.uk, raynal@irisa.fr

Abstract

Atomic Broadcast (where all processes deliver broadcast messages in the same order) is a very useful group communication primitive for building fault-tolerant distributed systems. This paper presents an atomic broadcast protocol that can be claimed to be optimal in terms of failure detection, resilience, and latency. The protocol requires only the weakest of the useful failure detectors for liveness, and permits upto (n-1)/2 processes to crash in a system of n processes; at most two communication steps and n broadcasts are needed in a run during which process crashes and failure-suspicions do not occur. We also introduce the notion of Notifying Broadcast which can reduce the message overhead further in 'nice' runs in which all processes are operational and communication delays do not exceed the bound assumed. If nice runs persist, the average message overhead is just one broadcast. That is, the protocol extracts no message overhead for providing crash-tolerance if process failures and unanticipated fluctuations in communication delays do not occur. We are currently implementing our protocol as a CORBA component. All known ORBs use IIOP as the standard protocol for inter-process communication, which in turn uses TCP/IP as the common transport protocol. It turns out that the Notifying Broadcast is straightforward to implement on top of TCP transport layer.

Keywords*: Asynchronous Distributed System, Atomic Broadcast, Communication Step, Consensus, Crash Failure, Notifying Broadcast, Reliable Broadcast.*

1 Introduction

Atomic Broadcast is an important building block in developing fault-tolerant distributed applications. It is a group communication primitive that allows a set of application processes to broadcast and deliver messages in an identical order. It is particularly useful to implement fault-tolerant services by using software-based replication. By employing this primitive to disseminate updates, all correct service replicas are delivered the same set of updates in the same order, and consequently they are kept in a mutually consistent state.

It has been shown that *Atomic Broadcast* and *Consensus* are equivalent problems in asynchronous systems prone to process crash failures [3]. The Consensus problem can informally be defined in the following way. Each process proposes a value and has to decide a value (termination property) in such a way that a single value is decided and that value is one of the proposed values (safety property). From both theoretical and practical points of view, the equivalence between Atomic Broadcast and Consensus is important. On the theoretical side, it means that all the results attached to the Consensus problem also hold for Atomic Broadcast and *vice versa*. From a practical point of view, it means that Atomic Broadcast protocols can be built from a Consensus routine.

A Consensus-based Atomic Broadcast protocol was first proposed in [3]. This protocol proceeds by determining a sequence of message batches. More specifically, processes sequentially agree on a first batch of messages, then on a second one, and so on. Moreover, they deliver the messages of a given agreed batch according to a predefined deterministic order. The agreement on a message batch is obtained by using a Consensus routine. Hence, the Chandra-Toueg's Atomic Broadcast protocol reduces the Atomic Broadcast problem to a sequence of Consensus instances. Consequently, the cost of agreeing on each batch is directly related to the cost of the underlying Consensus. Interestingly, this Atomic Broadcast protocol can work with any underlying Consensus protocol.

An approach closer to [3] is considered in [14]. This approach also reduces the Atomic Broadcast problem to a sequence of Consensus, but here Consensus (named *Optimistic Consensus*) is appropriately redefined to allow an expedited decision when the sequences of messages proposed by the processes to the Consensus share a common prefix. This approach can be particularly efficient when processes are naturally poised to receive the broadcast messages in the

same order from the network, as would normally be the case in a single-segment broadcast LAN, and propose them to the Consensus according to this received order. In the best scenario, this protocol requires two communication steps to agree on a batch of messages.

Mostefaoui and Raynal present a protocol that is relatively both more efficient and more modular [11]. As in the previous protocols, their protocol proceeds by sequentially determining batches of messages, but differently from them, uses an underlying Consensus routine only in some circumstances. Moreover, differently from [14], it works with any underlying Consensus protocol. Say, *n* be the number of processes, and *f* the maximum number of processes that can crash. In the best case, agreeing on a message batch requires two communications steps when $f < n/2$, and a single one when $f < n/3$. In the worst case, it requires an additional Consensus execution. The principle that underlies the protocol is simple and useful (in the sense it saves Consensus executions). It results in Consensus being bypassed when enough processes propose the same initial message batch. The Consensus is used only when processes happen to differ, due to asynchrony or process crashes, on their initial proposals.

Dutta and Guerraoui develop an interesting variation on the principles of [11] to design a consensus protocol. Their consensus protocol [4] can exhibit the best case behaviour (two communication steps [9]) even if processes make different initial proposals. This is achieved by having each process propose a message batch and also a process which it trusts to be the *leader*; if all processes initially propose the same *leader*, the message batch proposed by that *leader* is taken to be the agreed batch. Consensus is inevitable, if the commonly perceived $leader$ is crashed or if processes proposed different $leader$s.

Note that when the system enters transient periods of fluctuations in network delays, it is quite common for correct processes to *falsely* suspect each other to have crashed. During such periods of false suspicions, the protocol of [4] cannot be guaranteed to exhibit the best behaviour. The protocol presented in this paper removes this inefficiency at no extra cost in message or time overhead: message ordering and delivery are not disturbed even after the correct $leader$ is falsely suspected, and proceed unhindered until another process claims the leadership; putting it another way, atomic delivery of messages is guaranteed not to be blocked by false suspicions if Consensus does not result in a false agreement on the operational status of the existing correct leader. Thus, our protocol is immune to a small number of false suspicions; it also has two, very desirable, closely-related aspects of practical significance.

The protocols we have surveyed above are all *symmetric* in the sense that each process executes the same program (except for process identifiers) and consequently broadcasts to every other process in a communication step. Ours is an *asymmetric* one and only the leader broadcasts in the first step - thus reducing the message overhead by $n - 1$ broadcasts. Secondly, when no process has crashed and when message delays are less than the worst case estimates, the average message cost of an atomic broadcast reduces to one broadcast if the underlying communication subsystem provides a *notification* facility after a broadcast is initiated: the broadcasting process is notified, at some specified point in time, whether or not the network has correctly transmitted the message to all destination hosts. The notification is accurate when it is positive (i.e., a *success* notification does mean that all hosts have received), but a negative notification can be imprecise (all hosts may have received). We term this the *Notifying Broadcast*, and use it to improve the best case behaviour when there are no process crashes. We also show that it can be easily built on top of TCP/IP.

The paper is organised as follows. Section 2 introduces the system model and formally defines the Atomic Broadcast problem. Then, Section 3 decsribes the protocol and the chosen framework for our on-going implementation. Finally, Section 4 concludes the paper.

2 Computation Model and Atomic Broadcast

2.1 Asynchronous Distributed System with Process Crash Failures

The distributed system consists of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.*, by prematurely halting. It behaves correctly (*i.e.*, according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash; *f* denotes the maximum number of processes that can crash. Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is permanently connected by a channel. Messages can be *occasionally* lost or corrupted while being transmitted over a channel. These (transient) failures can be masked through a finite but unknown number of retransmissions (temporal redundancy). When a channel correctly transports a message eventually (with or without retransmissions), the transfer delay cannot be known *a priori*. Similarly, no assumption is made about the relative speed of processes. The absence of explicit timing assumptions in our model makes the distributed system *asynchronous*.

2.2 Underlying Building Blocks

We make use of a few underlying services, of which *Notifying Broadcast* is one, in order to build our atomic broadcast protocol. Here, we describe the functionalities these services offer and the primitives they export, beginning with the communication sub-system.

Communication Subsystem. The communication subsystem, denoted as *CS*, facilitates message exchange over the channels on the best effort basis. It offers the primitive Send(m) for *m* to be sent, and executes Receive() for receiving messages. We assume that the *CS* deposits a received message in an appropriate receive buffer for a higher-level process to consume. We also assume the following properties:

- Termination. An invocation of Send(m) terminates within a finite time returning either a *success* or a *failure* notification. The former assures that *m* has been received by the *CS* of the destination process and the latter provides no such assurance.
- Validity. When Receive() deposits *m* in a receive buffer, *m* has been sent by some process.

The properties we have assumed for *CS* are readily satisfied by TCP (Transmission control protocol) transport layer - a protocol that supports nearly all Internet applications - which we also use in our implementation. On sending a message *m* using TCP, *m* is fragmented into IP datagrams which are transmitted to the destination. Using acknowledgements, timeouts and selective retransmissions, the integrity of message transmission is monitored and ensured. TCP dynamically estimates the timeout period for receiving acknowledgements as a smoothed average over the round trip times (RTT) observed in the past. This estimation allows for small fluctuations in message communication delays. If all the sent datagrams are not acknowledged, the connection is deemed to have failed, and a notification is given. At the destination side, the uncorrupt datagrams received are assembled back into *m* which is buffered for higher-level consumption.

If the undelying communication system does not use TCP (say UDP instead), it needs to be enhanced to satisfy the properties assumed for *CS*. Though this enhancement is not so simple as using TCP, it frees the system of many restrictions the 'user friendly' TCP imposes. For example, the timeouts can be robustly estimated and the maximum number of retransmissions be dynamically fixed, thus reducing the scope for unwarranted failure notifications while network delays fluctuate widely.

Notifying Broadcast. The aim of this service is to allow a message to be sent to a set of processes and to notify the sending process whether or not the message has been received by the *CS* of all destination processes. In addition, this service ensures that broadcast messages of a correct process are eventually delivered to all correct destination processes, and that messages from a given process are delivered in the sent order. Formally, Notifiying Broadcast module exports two primitives: N_Broadcast and N_Deliver. The semantic of these primitives is defined by five properties, namely, Validity, Integrity, Termination, Order, and Notification. When a process executes N_Broadcast(m) (resp. N_Deliver(m)) we say that it N-broadcasts m (resp. N-delivers m). We assume that all messages N-broadcst in the system are different.

- Validity. *If a process N-delivers m, then some process has N-broadcast m.* (No spurious messages.)
- Integrity. *A process N-delivers a message m at most once.* (No duplication.)
- Termination. *If a correct process N-broadcasts m, then all correct processes N-deliver m.* (No message N-broadcast by a correct process is missed by a correct process.)
- Order. *If a process N-broadcasts m_1 followed by m_2, then any process that N-delivers m_2 would have N-delivered m_1 first.*
- Notification. *If a process N-broadcasts m, it will be notified at some time whether or not m has been received by the CS of all destination processes by that time.* A *success* notification is correct and a *failure* notification means that no assurance on message reception can be given at the time of notification.

The Notifying Broadcast without the *Notification* property is usuallly called the *Unreliable Fifo Broadcast* in the literature [7]. It is unreliable because it offers no guarantee that all correct processes will N-deliver *m* if an execution of N_Broadcast(m) is interrupted by the crash of the N-broadcasting node.

We implement the Notifying Broadcast service above the *CS* and meet the Notification property by making use of the notifications returned by Send(m). An invocation of N_Broadcast(m) leads to invoking Send(m) to all destinations and concurrently logging *m* in the disk for possible future re-transmissions. If *CS* of every destination is known to have received then a *success* notification is returned; else a *failure* one. Within a destination host, every new ($i.e.$, non-duplicate) N-broadcast message buffered by the local Receive() is logged so that a request for transmission can be responded to. If the logged message indicates that some earlier messages from the same source are missing, a request for re-transmission of missing messages is made to the source which, if not crashed, will re-transmit (unicast) the missing messages to the requesting destination.

Also, in our implementation, processes periodically N-broadcasts an I-AM-ALIVE message in an attempt to avoid being falsely suspected. Thus, correct processes are lively - continually exchanging messages with each other. This liveliness together with the ability to detect and retrieve

missing messages ensures that the *Termination and Order* properties are met.

Reliable Broadcast. The aim of the underlying *Reliable Broadcast* service is to allow a message to be reliably sent to processes. "Reliably" means here that if the message is delivered by a correct process, then it is delivered by all correct processes. Reliable Broadcast module exports two primitives [7]: R_Broadcast and R_Deliver. The semantics of these primitives is defined by three properties, namely, Validity, Integrity and Termination. As earlier, when a process p executes R_Broadcast(m) (resp. R_Deliver(m)) we say that it R-broadcasts m (resp. R-delivers m). Again, we assume all messages R-broadcast within the system are different.

- Validity. *If a process R-delivers m, then some process has R-broadcast m.* (No spurious messages.)
- Integrity. *A process R-delivers a message m at most once.* (No duplication.)
- Termination. *If (1) a correct process R-broadcasts m, or if (2) a correct process R-delivers m, then all correct processes R-deliver m.* (No message R-broadcast by a correct process or R-delivered by a process is missed by a correct process.)

We implement Reliable Broadcast on top of the Notifying Broadcast module, using the latter for unreliable fifo broadcast service (i.e., without making use of the notification facility: if *m* is a R-broadcast message, it is R-delivered only after it has been N-broadcast.

Consensus. In the *Consensus* problem each process proposes a value and all correct processes have to decide on some value v that is related to the set of proposed values [6]. Formally, the Consensus problem is defined in terms of providing two primitives: C_Propose and C_Decide. As in previous works (*e.g.*, [3]), when a process p invokes C_Propose(w), where w is its proposal to the Consensus, we say that p "proposes" w. In the same way, when p invokes decide and gets v as a result, we say that p "decides" v (denoted C_decide$()$).

The semantics of propose and decide is defined by the following properties[1]:

- Validity. *If a process decides v, then v was proposed by some process.* (No "spurious" value can be decided.)
- Integrity. *A process decides at most once.* (No "duplicates": from the point of view of a process, there is a *single* decision per Consensus instance.)
- Termination. *All correct processes eventually decide.*
- Uniform Agreement. *No two processes (correct or not) decide differently.* (This property gives its global meaning to the Consensus: from the point of view of all processes there is a *single* decision per Consensus instance.)

It has been shown that the Consensus problem cannot be solved in asynchronous distributed systems in a deterministic manner if processes (as we've assumed) are prone to crash [6]. The system has to be equipped with additional devices for the problem to be solvable. To our knowledge the main approaches investigated to design Consensus protocols for such systems are the following. One is based on random oracles (the progress of a process can be determined according to random numbers) [2, 5]. Another catagory is based on the use of Chandra-Toueg's unreliable failure detectors [3] (several failure detector-based consensus protocols have been designed, e.g., [3, 8]). Finally, the last category of Consensus protocols uses a hybrid approach. It combines random oracles and failure detectors [1, 13]. We build the consensus service by implementing [5].

2.3 Handling Buffer Overflow

Recall that the Receive process of the *CS* deposits a received *m* in an appropriate receive buffer which is the interface between the *CS* and the kernel of the receiving host. An implementation can handle this 'full buffer' syndrome in one of three ways.

- *Post-receive blocking.* Receive process blocks until enough free space becomes available in the receive buffer for the received *m* to be deposited. No new message can be received while blocking; hence this option leads to *failure* notifications being returned in the sending hosts. In TCP, the sending side will regard the connection with the receiver to have failed and will attempt to establish a new connection for subsequent message transmission; communication and time overhead for opening a new connection can be high.
- *Hot Potato.* Receive process simply drops the received *m* if there is no free buffer space. This means that any *success* notification returned in response to Send(m) or N_Broadcast(m) cannot be taken to indicate that m will be delivered to the destination process. Recall that a $success$ notification is defined only as an indication that m has been received by the CS of the destination host, not by the destination process itself.
- *Pre-send blocking.* Processes of Notifying Broadcast modules execute a flow control algorithm ($e.g.$, [10])

[1] The set of properties we present here actually defines what is called *Uniform Consensus* by some authors [3].

lest m is N-broadcast unless buffer space is known to be available at all destination hosts. Though sending of message can be blocked, little message overhead is incurred if flow-control information is piggybacked onto out-going application and other control messages.

We have chosen the second option in our implementation for two reasons: first, a dropped message will be detected as a missing one (due to $fifo$ delivery) and can be retrieved through selective transmission (unicast) if the sender is correct; secondly, since the network is asynchronous, all destination hosts are unlikely to receive a given message in identical situations and hence all of them are unlikely to drop it. More precisely, we make a pessimistic assumption that a given m is not dropped by a majority of destination hosts whose CS has received m. Below, we state this assumption in the context of Notifying Broadcast:

Assumption. If a process that N-broadcasts m is returned a $success$ notification, then m can be dropped in at most f destination hosts.

We remark here that dropping a received m due to lack of buffer space, does not constitute a failure; it is a specified behaviour in the $Hot\ Potato$ approach we have adopted. Further, the above assumption refers to a context where every CS receives m; otherwise, a $success$ notification would not have been returned. Suppose that the CS of p_i does not drop the received m. Since p_i can crash before m can be delivered to it, and at most f such processes can crash, the assumption implies that when N_Broadcast(m) returns a $success$ notification, m is delivered to at least one correct process in Π.

2.4 The Atomic Broadcast Problem

The *Atomic Broadcast* problem is defined by two primitives, namely, A_Broadcast and A_Deliver. The first one allows a process to send a message to the entire set of processes. The second one allows a process to deliver a message sent by the invocation of A_Broadcast. As previously, when a process executes A_Broadcast(m) (resp. A_Deliver(m)) we say that it "A-broadcasts" m (resp. "A-delivers" m). Actually, Atomic Broadcast is "Reliable Broadcast + total order on message delivery + uniform message delivery ". This means that the A_Broadcast and A_Deliver primitives satisfy the Validity, Integrity and Termination properties defining Reliable Broadcast, plus the following two properties [7]:

- Total Order Delivery. *If two processes p_i and p_j both A-deliver m and m', then p_i A-delivers m before m' if and only if p_j A-delivers m before m'.*
- Uniform Message Delivery. *If a process (correct or not) R-delivers m, then all correct processes R-deliver m.*

The first property makes the Atomic Broadcast problem non-trivial. It requires an agreement to ensure that there be no conflict in the order in which processes deliver A-broadcast messages.

3 Protocol Description

3.1 Background and Notations

As stated in Section 1, the protocol is asymmetric in structure and is a leader-based one. The $leader$ is one of the processes in Π that has the role of assigning the order in which A-broadcast messages ought to be A-delivered. The delivery order is expressed by assigning to a given m a unique sequence number, denoted as $m.seq\#$ which is a non-zero, positive integer. When non_leader processes agree that the $leader$ is crashed, the leadership is offered to the process that is next in the process ordering which is assumed to be: $p_1, p_2, \ldots, p_n$. Thus, p_1 is the $leader$ when the system is initialised.

Since it is always possible for a correct $leader$ to be (falsely) agreed by non_leader processes to have crashed, if p_n is agreed to have crashed, the leadership offer cycles back to p_1. To distinguish the leadership activities of the same process in different cycles, a $leader$ is associated with a leadership number, denoted as $leader\#$. When process p_i is the $leader$ in cycle $c, c \geq 1$, its $leader\#$ is simply $(c-1)n + i$.

The $leader$ announces the $m.seq\#$ it assigns to a given m, by N-broadcasting message SEQ(m) which contains, in addition to $m.seq\#$, its $leader\#$ and a list $notification\_list$ that contains the notifications (if any) which it had received for any SEQ messages it had N-broadcast earlier. Thus,

$$\text{SEQ}(m) = \{m.seq\#,\ leader\#,\ notification\_list\}.$$

If $m'.seq\#$ is smaller than (resp. larger than) $m.seq\#$ then m' is said to $precede$ (resp. $succeed$) m. We will denote the message whose sequence number is one less than (resp. one more than) $m.seq\#$ as m_- (resp. m_+).

The rest of the description is organised into answering three important questions:

(i) How are messages A-delivered when the current $leader$ is correct and is seen by others to be correct?

(ii) What provisions are there to handle process crashes, in particular, the situation where some processes orderd after the current crashed $leader$ are also crashed ($telescopic\ crashes$)?

(iii) How is a new $leader$ prevented from re-ordering messages which are already A-delivered by some process?

The final sub-section describes some optimisations that can be made, and some that can only be attempted with caution due to the lossy, asynchronous network model.

3.2 Atomic Delivery

When a process invokes A_Broadcast(m), it is marked as an ATOMIC_BROADCAST message and is R-broadcast[2]. When the $leader$ R-delivers an m that is marked as ATOMIC_BROADCAST, it constructs and N-broadcasts SEQ(m). This broadcast is the first communication step carried out to A-deliver m.

Each non_leader process, say p_i, maintains in the variable $leader_i$ the identifier of the process which it considers to be the current leader and in the variable $leader\#_i$ the associated $leader\#$. When p_i N-delivers SEQ(m) such that SEQ(m).$leader\# = leader\#_i$, it accepts SEQ(m) and waits for it to become stable.

Definition: Stable Message. A SEQ(m) is stable for p_i when p_i knows that either

- SEQ(m) has been received by CS of all destination processes ($stability\_condition1$), or
- SEQ(m) has been N-delivered by $(n - f)$ processes ($stability\_condition2$).

To stabilise an N-delivered SEQ(m), p_i waits to N-deliver a SEQ message whose $notification\_list$ contains m. If the N-broadcast of SEQ(m) by $leader_i$ is indicated to be a $success$, $stability\_condition1$ is met. If it is indicated to be a failure or if $leader_i$ is locally suspected (to have crashed) due to a prolonged absence of an I-AM-ALIVE message, p_i N-broadcasts SEQ(m) and waits for $stability\_condition2$ to be satisfied. Note that when p_i performs the N-broadcast, the second communication step occurs, which will not be needed if $leader_i$ is correct and not suspected and its N-broadcast of SEQ(m) is notified to be a $success$.

Definition: Deliverable Message. A SEQ(m) is deliverable by p_i when all SEQ(m') for m', $1 \leq m'.seq\# < m.seq\#$ are stable for p_i. That is, SEQ($m\_$) is deliverable and SEQ(m) is stable. The variable $last\_del\#_i$ keeps the maximum $seq\#$ of the messages that became deliverable for p_i.

p_i A-delivers the deliverable messages as per their sequence number. For our protocol to be correct, we need to show that when a SEQ(m) becomes stable for p_i, (i) no SEQ(m') is ever constructed with $m'.seq\# = m.seq\#$, and (ii) SEQ(m) eventually becomes stable for every correct p_j.

[2]This R-broadcast is essential for A-delivering m, and its cost is not counted in the overhead for A-delivery. See the sub-section 3.5 for minimising this cost.

With (i) and (ii) shown, the correctness can be easily established by induction on $m.seq\#$, which we omit in this paper. To show (i) and (ii), we first state a claim:

Claim. Let SEQ(m) become stable for p_i at time t. Let S be any set of $(n-f)$ processes which have not crashed until some finite time after t. It is claimed that S has at least one process which has N-delivered SEQ(m) or whose CS has not dropped SEQ(m) when SEQ(m) was N-broadcast.

If SEQ(m) becomes stable for p_i due to the $stablity\_condition2$, the claim is straightforward since any two sets of $(n - f)$ processes intersect. Now, consider that SEQ(m) becomes stable for p_i due to the $stablity\_condition1$. Let $Crashed_t$ be the set of processes that crashed by t, and $Dropped_m$ be the set of processes whose CS dropped SEQ(m) when SEQ(m) was N-broadcast. The claim is true if we show that S and $(\Pi - Crashed_t - Dropped_m)$ intersect.

Consider $(S - Dropped_m)$, the set of those processes in S whose CS has not dropped SEQ(m). The $stablity\_condition1$ implies that the CS of all processes in Π received SEQ(m); by assumption 1 of Sub-section 2.3, $| Dropped_m | \leq f$. Since $n > 2f$ and $| S | = n - f$, $(S - Dropped_m)$ cannot be empty. Let $p \in (S - Dropped_m)$. By its definition, p did not crash at t; so, $p \in (\Pi - Crashed_t)$; that is, $p \in (\Pi - Crashed_t - Dropped_m)$.

3.3 Handling Failures

p_i maintains in the variable OC_i the identifier of the $observed\_candidate$ - a process that is being observed for failure which, if agreed on, triggers some process to claim the leadership. OC_i is mostly the same as the $leader_i$, but will be different during the period between $leader_i$ is agreed to have failed and a new leader is yet to be installed. The variable $OC\#_i$ will have the $leader\#$ of OC_i; at the start, OC_i and $OC\#_i$ are initialised to $leader_i$ and $leader\#_i$, respectively. The third variable, $last\#_i$ records the largest of $m.seq\#$ N-delivered from OC_i; when no SEQ(m) has been yet N-delivered from OC_i, $last\#_i$ is set to -1.

If OC_i is locally suspected, p_i initiates a binary consensus instance over the assertion: "Has $OC\#_i$ crashed after N-broadcasting $last\#_i$?". p_i's proposal to this consensus instance will obviously be yes. Similarly, another p_j might initiate a consensus instance over $OC\#_j$ for $last\#_j$. p_i will propose to the consensus initiated by p_j only if $OC\#_i = OC\#_j$, and the (binary) value it proposes will be based on local conditions and the outputs of the local failure-suspector. Note that several consensus instances may be in progress at a given time and the ones over a given $OC\#$ will be distinguished by the $last\#$. If p_i decides yes in a consensus run over $OC\#_i$ for any $last\#$, OC_i is set

to its successor, $OC\#_i$ incremented by 1, and $last\#_i$ set to -1.

Consider an example in which OC_i is initially p_1 and $OC\#_i$ is 1. If a consensus instance over $OC\#_i = 1$ for some $last\#$ results in a yes decision, then OC_i will change to p_2 which will now become the candidate being observed. If p_2 is already crashed, then a consensus instance will be initiated for $OC\#_i = 2$ with $last\# = -1$. Since all processes that propose will propose yes, correct p_i will decide yes in the consensus instance for $OC\#_i = 2$ and set $OC_i = p_3$ and $OC\#_i = 3$. Thus, the processes in Π are observed one after the other, until there exists a correct process that is not suspected by any correct process. Note that a change in $OC\#_i$ or OC_i does not automatically change $leader_i$; this means that if $leader_i$, p_1 in the example, is correct, messages sequenced by p_i will continue to be A-delivered, unhindered even by $telescopic\ crashes$. In the next Sub-section, we describe how a process that is due to take the leadership role, attempts to install itself as the $leader$.

3.4 New Leader Installation

When a process, say p_j finds itself the observed candidate (*i.e.*, $p_j = OC_j$), it attempts to install itself as the $leader$, by N-broadcasting an INSTALL message which has two fields:

• $OC\#_j$. This will be the $leader\#$ if p_j succeeds in installing itself as the $leader$. This field thus indicates the $leader\#$ proposed by p_j's INSTALL message and is termed as the $new\_leader\#$.

• $last\_del\#_j$. This is the largest $seq\#$ of m that became deliverable for p_j. This field indicates the $seq\#$ with which p_j's leadership intends to close the old leadership epoch. It is termed as the $closing\#$. Thus,

$$\text{INSTALL} = \{new\_leader\#,\ closing\#\}.$$

Process p_i accepts an N-delivered INSTALL message, only if $OC\#_i \leq new\_leader\#$. If $OC\#_i > new\_leader\#$, then p_i has already decided yes in a consensus run for $new\_leader\#$, and the INSTALL message is ignored. If p_i accepts INSTALL it executes the following three steps:

(i) p_i sets $leader_i$ and $leader\#_i$ to the $sender$ and the $new\_leader\#$ of the accepted INSTALL message, respectively; OC_i and $OC\#_i$ to $leader_i$ and $leader\#_i$ respectively; and, $last\#_i$ to 0. (An accepted INSTALL message is assumed to have been given the $seq\#$ of zero.)

(ii) It forms a set msg_bag consisting of all SEQ(m) messages that have been deposited by the local Receive process, whose $m.seq\#$ is larger than $closing\#$ of the accepted INSTALL message. Recall that every N-broadcast message that is taken out of the receive buffers is logged; to inspect those still in the buffers, a primitive PEEK_BUFFER() is provided.

(iii) It then sends to $leader_i$ the message OK $= \{\text{INSTALL}, msg\_bag_i\}$.

When p_j delivers $(n - f)$ OK messages, it forms Msg_Bag as the union of msg_bags in the OK messages. It then looks for $conflicting$ SEQ messages in Msg_Bag: two SEQ messages are said to $conflict$ if they indicate the same sequence number for different m or different sequence numbers for a given m. Between conflicting SEQ messages, the one that contains the smaller $leader\#$ is discarded from Msg_Bag. After purging Msg_Bag of conflicting entries, p_j starts sequencing messages starting from $(closing\#+1)$, ensuring that the sequence number indicated by a SEQ(m) in Msg_Bag is (re)assigned only to m.

Based on the claim made in Sub-section 3.2, the following is obvious if S is the set of $(n - f)$ processes whose OK messages p_j delivered: if SEQ(m) is stable for a process, it is in Msg_Bag and therefore no newly installed $leader$ will generate a SEQ(m') such that $m.seq\# = m'.seq\#$. Conversely, if SEQ(m) $\notin Msg\_Bag$ and $m.seq\# > closing\#$, then SEQ(m) could not have become $stable$ for any process.

Consider two conflicting SEQ messages: SEQ1and SEQ2. Their $leader\#$ fields must be different; otherwise, it would mean that they were constructed by the same process which is not possible. Let the $leader\#$ fields of SEQ1and SEQ2 be $leader\#1$ and $leader\#2$, respectively and $leader\#1 < leader\#2$. This means that $leader\#2$ did not have SEQ1 in its Msg_Bag when it was installed the new leader. So, SEQ1 could not have become stable for any process, and therefore p_j can discard SEQ1 from its Msg_Bag.

3.5 Optimisation Remarks

When p_j invokes R_Broadcast(m), m is marked as an RELIABLE_BROADCAST message and is N-broadcast. When a correct p_i, $i \neq j$, N-delivers m, it must N-broadcast m as its own message before it R-delivers m. This 'echo' broadcast ensures the termination property of the Reliable Broadcast: what is R-delivered by one correct process is R-delivered by all correct ones. It can be carried out in an indirect manner, avoiding an explicit N-broadcast of m: p_i prepends its sequence number for the echo broadcast and the header of of m, onto an I-AM-ALIVE message which it periodically N-broadcasts. Any correct p_k that receives the I-AM-ALIVE message will interpret the echo broadcast as a 'missing' message; it will ask p_i for re-transmission only if it has not N-delivered m from p_j; otherwise it will reconstruct the 'missing' message from the prepended information. (Note: the sequence number p_i assigns to the echo broadcast of m is for the $fifo$ delivery of its N-broadcasts and should not be confused with the $m.seq\#$ assigned by the leader for atomic delivery of m.)

When p_i learns that m has become deliverable for some (correct or crashed) p_j , it can also treat m as a deliverable message even if m has not locally satisfied the conditions stated in sub-section 3.2. This observation can be used to speed up the A-delivery of messsages. To achieve this, processes should include their $last\_del\#$ in their A-broadcast or I-AM-ALIVE messages.

The exchange of $last\_del\#$ by processes is necessary to keep the log from growing. (Recall that every N-delivered message is logged.) An N-delivered message is removed from the log only after it is known to have become deliverable for all processes in Π. This means that once a process p_k, $p_k \in \Pi$, crashes, the operational processes cannot remove the logged messages which, strictly speaking, need to stay in the log for ever. This is a difficulty in practice and arises due partly to the $Hot\ Potato$ approach in handling buffer overflow problem (sub-section 2.3) and partly to our (realistic) consideration of channels being prone to transient failures. We intend to address this problem in future.

4 Concluding Remarks

Atomic Broadcast is a very useful middleware primitive for building fault-tolerant distributed systems. This paper presents an Atomic Broadcast protocol that can be claimed to be optimal in many aspects. It is efficient in situations where process crashes and wide fluctuations in network delays are rare. This efficiency has been obtained by introducing the notion of $Notifying\ Broadcast$ which makes the network-level signals visible to the higher level processes for valuable interpretations to be made; also, by adopting a leader (or sequencer) based approach, the use of binary Consensus is limited to handling actual or suspected leader crash. The implementation framework describes how $Notifying\ Broadcast$ can be built as an underlying service in a modular manner. It appears that this service is not hard to implement, and is straightforward on top of TCP transport layer.

The proposed protocol assumes that more than half the number of processes do not crash and that false suspicions cease eventually. In the most favorable situation of all processes being operative and no false suspicions, the message cost of atomic broadcast averages to a single broadcast (by the $leader$). With no new crashes and no false suspicions, the protocol requires two communication steps. A very useful aspect of the protocol is that when a correct leader is falsely suspected only by a 'small' number of processes, atomic delivery of messages is not blocked.

Acknowledgements. The work is partly supported by the UK EPSRC project $PACE$.

References

[1] Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

[2] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *2nd ACM Symposium on Principles of Distributed Computing, (PODC'83)*, Montréal (CN), pp. 27-30, 1983.

[3] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, March 1996.

[4] Dutta P. and Guerraoui R., Fast Indulgent Consensus with Zero Degradation. *Proc. 4th European Dependable Computing Conference (EDCC4)*, Toulouse, France, October 2002.

[5] Ezhilchelvan P., Mostefaoui A. and Raynal M., Randomized Multivalued Consensus. *Proc. 4th Int. Symposium on Object-oriented Real-time distributed Computing, ISORC01*, Magdeburg, Germany, pp. 195-201, May 2001.

[6] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[7] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[8] Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395-408, 2002.

[9] Keidar I. and Rajsbaum S., On the Cost of Fault-Tolerant Consensus when There are No Faults: a Tutorial. *SIGACT News, Distributed Computing Column*, 32(2):45-63, 2001.

[10] Macedo R., Ezhilchelvan P., and Shrivastava S., Flow-Control Schemes for a Fault-Tolerant Multicast Protocol. *Proc. of the IEEE Pacific Rim Int. Symposium on Distributed Computing (PRDC'95)*, pp. 85-90, 1995.

[11] Mostefaoui A. and Raynal M., Low Cost Consensus-Based Atomic Broadcast. *Proc. of the IEEE Pacific Rim Int. Symposium on Distributed Computing (PRDC'00)*, IEEE Computer Press, pp. 45-52, Los Angeles (CA), 2000.

[12] Mostéfaoui A., Rajsbaum S. and Raynal M., A Versatile and Modular Consensus Protocol. *Proc. Int. IEEE Conference on Dependable Systems & Networks (DSN'02)*, IEEE Computer Press, pp. 364-373, Washington D.C., 2002.

[13] Mostefaoui A., Raynal M. and Tronel F., The Best of Both Worlds: A Hybrid Approach to Solve Consensus. *Proc. Int. Conf. on Dependable Systems and Networks (DSN'00)*, IEEE Press, pp. 513-522, 2000.

[14] Pedone F. and Schiper A., Optimistic Atomic Broadcast. *Proc. 12th Int. Symposium on Distributed Computing (DISC'98)*, Springer-Verlag, LNCS #1499, pp. 318-332, 1998.