A Fault-Tolerant Dynamic Atomic Broadcast Algorithm with QoS Guarantees

Ziv Bar-Joseph, Idit Keidar, and Nancy Lynch
`zivbj@mit.edu, {idish, lynch}@theory.lcs.mit.edu`
MIT Laboratory for Computer Science,
545 Technology Square, Cambridge, MA 02139.

December 18, 2000

Abstract

We present a fault-tolerant algorithm for an *atomic broadcast service* with a *dynamic* set of participants; that is, reliable totally ordered multicast for dynamic groups. The algorithm preserves QoS guarantees. We offer a detailed theoretical study of the QoS guarantees of our algorithm under different circumstances. In particular, we show that in periods with no failures, the latency for the ordered multicast is within a constant of the latency of the underlying network (independently of the number of participants). This is an improvement over the latency exhibited by previous algorithms. When failures do occur, the latency is linear in the number of processes that fail within a bounded time interval, as dictated by a lower bound. Unlike most group communication systems providing similar services, in our algorithm processes can join and leave without introducing delays in the communication between active participants. A major challenge was avoiding communication delays for active processes when joins occur, while at the same time preserving consistency if failures occur near the time of a join.

Keywords: Fault-Tolerant algorithms, Real time, Fault-Tolerant communication, Performance Analysis, Quality of Service (QoS).

1 Introduction

Atomic broadcast [12, 5], allows multiple processes to send messages, in such a way that all the correct processes deliver all the messages sent or delivered by correct processes, and in the same order. An important use of atomic broadcast is to implement *replicated state machines* [13, 18], which provide an important paradigm for state-oriented applications. Much work has been dedicated to atomic broadcast algorithms in different failure models [8]. *Dynamic atomic broadcast* is an extension of atomic broadcast that supports requests by application processes to *join* or *leave* the algorithm, in addition to tolerating process failures; dynamic atomic broadcast is often implemented using *group communication systems* (e.g., [10, 21, 6]). In this paper we present a novel *dynamic atomic broadcast (DAB)* algorithm that preserves *quality of service (QoS)* guarantees.

In the past few years, we have witnessed new applications that require QoS guarantees from the network (e.g., [16]). Some need strict guarantees on available bandwidth, others need a bound on the latency a packet can suffer when transmitted over the network. ATM networks [3] allow applications to reserve QoS parameters such as bounded latency, guaranteed bandwidth and bounded loss rate. The *IETF Integrated Services* working group is concerned with adding similar QoS support to the Internet. The QoS parameters that the new services will support include, among others, bounded latency, guaranteed bandwidth reservation and bounds on message loss (see [19]).

There are applications that replicate some state with a certain degree of consistency and yet also require predictable message delays. Such applications can benefit from DAB, as long as it does not introduce excessive delays. Examples of such applications include a shared text editor [20], a shared white-board [16], and online strategy games [11, 1].

Current implementations of applications such as those described above seldom exploit atomic broadcast. This is because achieving atomic broadcast requires delaying messages until agreement upon their order is reached, and many believe that this delay is too large. For example, in his book Internetworking Multimedia [7], Crowcroft writes:

> *"The requirements of resilience and scalability dictate that total consistency of view is not possible unless mechanisms requiring unacceptable delays are employed."*

The idea that consistency and predictable delays are mutually exclusive is at the root of design decisions made in building such applications [16, 20]. Such applications usually settle for weak consistency constraints and run application-specific algorithms to detect and resolve inconsistencies.

In this paper we show that atomic broadcast can coexist with guaranteed predictable delays in some situations, albeit not in all situations. We consider a network that provides QoS guarantees, and build on top of it a DAB service. Unfortunately, no fault tolerant algorithm for atomic broadcast can guarantee a constant message delay; when processes fail, the delay is bound to grow linearly with the number of failures (see Section 11 for a proof of this lower bound). We present a fault tolerant DAB algorithm with QoS guarantees that match this lower bound.

We offer a theoretical study of the QoS guarantees of our algorithm under different circumstances. We show that with our algorithm, communication latency is within a constant of the latency of the underlying network (independent of the number of participants) in periods with no failures. When processes do fail, the delay of the algorithm increases linearly with the number of time intervals of a given size, x, during which failures occur. If one failure occurs, and during the following time interval of length x no failures occur, then the delay is bounded by a constant. The message delay can further increase by $f * x$ only if during each of the f subsequent time intervals of length x a new failure occurs. In practice, we do not expect sequences of failures to occur very often. Thus, the expected delay of our algorithm is very low, and it is very close to the delay achieved when no process fail. This is superior to previous results (see [2]) which introduced linear latency regardless of the number of failures.

Unlike most group communication systems providing similar services (e.g., [10, 21, 6]), in our algorithm processes can join and leave without introducing delays to communication between active participants. It was challenging to design an algorithm that would avoid communication delay to active processes when joins occur without failure, while not compromising consistency if failures occur near the time of a join.

1.1 Related work

Dynamic atomic broadcast is provided by several group communication systems. Most of these do not address QoS issues. The only exception that we are aware of is RTCAST [2]. RTCAST achieves a latency bound which is *linear* in the number of processes, regardless of the number of failures. Moreover, the failure model assumed in RTCAST is weaker than the one we assume. There, it is assumed that if a process p fails, and a correct process q receives, from the network, some message m sent by q before its failure, then every other correct process will receive m as well. In contrast, we allow the network to deliver a message from a faulty process to some correct process and not to another. Much of our algorithm's complexity is dedicated to overcoming such situations.

In an earlier paper [4], we have presented a simpler algorithm that does not overcome such situations, and instead allows correct processes to deliver different message sequences in cases of failures.

1.2 Roadmap

The rest of this paper is organized as follows: Section 2 presents the model, Section 3 specifies the DAB service we implement, and Section 4 describes our assumptions of the underlying communication network. The following three sections describe our new DAB algorithm. In Section 8 we informally argue that the algorithm is correct (i.e., implements DAB). In Section 9 we study the algorithm's QoS guarantees; the Appendix presents a formal correctness proof. In Section 10 we explain how the algorithm can be extended to recover from situations in which the network QoS is violated. In Section 11 we prove the lower bound for DAB. Section 12 concludes the paper.

2 Model

We assume a static universe P of n processes, with distinct identifiers in $\{1, \cdots, n\}$. Processes communicate by exchanging multicast messages within a *multicast group*[1]. Processes can voluntarily join and leave the multicast group at any time. Each join request is associated with a unique, monotonically increasing *incarnation number*, so that a process that joins the group multiple times uses a larger incarnation number each time.

Processes use an underlying network communication service which allows for QoS reservation. Specifically, the network allows for reservation of variable bandwidth, specified by the average transmission rate and the maximum burst that can be sent during a time interval of length Θ. As long as messages are sent at the reserved rate, the network guarantees to deliver messages with a bounded delay, denoted by Δ. In Section 4 we specify the underlying network service interface and our assumptions about the network.

[1]For simplicity, we assume that a single multicast group exists.

The failure model. Processes can fail by crashing and may later recover. Formally, we model failures by special *fail* actions; we do not model recoveries explicitly, but we allow a previously failed process to later perform a *join*. Crashed processes lose their volatile memory; however, we assume that recovered processes use fresh (larger) incarnation numbers following their recovery and do not re-use old ones[2]. We do not consider Byzantine (malicious) failures. When a process j fails, messages that j sent during the last Θ time before its failure may be lost due to the failure. Such lost messages may be received by some correct processes and fail to reach others.

Clock synchronization. We assume each process i has an internal clock denoted by $clock_i$. We assume that the difference between $clock_i$ and the real time is bounded. We denote by *now* the real time that has passed from the beginning of the execution[3] (thus, each execution starts with $now = 0$). We assume that there is a constant Γ so that the maximum difference between $clock_i$ and *now* is bounded by $\Gamma/2$. Thus, for each process i: $now - \Gamma/2 \leq clock_i \leq now + \Gamma/2$. This implies that the maximum difference between two processes' internal clocks is at most Γ. We further assume that each process can precisely schedule events according to its local clock.

The mathematical framework. We model each process as a timed I/O automaton [15]. An automaton interacts with its environment by two sets of external actions: input actions and output actions. A *trace* of an I/O automaton is the sequence of external actions it takes in an execution. Executions are assumed to be sequential, that is, actions are atomic, and no two actions can occur simultaneously.

3 Dynamic Atomic Broadcast Service Specification

We present an algorithm that guarantees gap-free total ordering of messages and also preserves QoS. The algorithm is implemented by a *Dynamic Atomic Broadcast (DAB)* layer that resides between the application and the underlying network, as depicted in Figure 1.

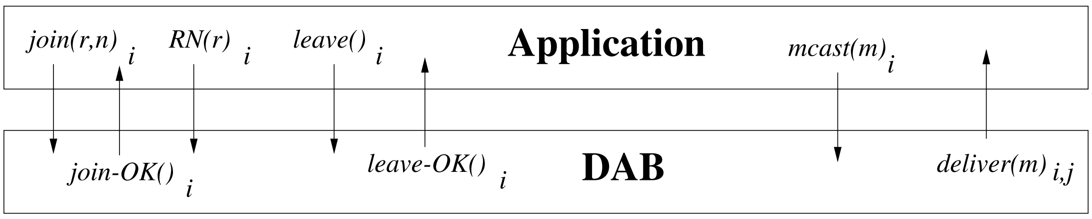


Figure 1: The dynamic atomic broadcast (DAB) service interface.

We now specify the dynamic atomic broadcast service. This service is composed of the DAB layer and the underlying network (cf. Figure 1). In this section we use the term *process* to refer to an application process running at a certain location. Processes use the service to send messages of a bounded size to the multicast group; the service delivers messages to all the processes in the same order.

3.1 Reservation model

Upon joining the multicast group, a process *reserves* the bandwidth required for its communication with all process in P, that is, the process asks the service to allocate a certain bandwidth. If

[2]This can be achieved by storing incarnation numbers on disk, or by using clocks
[3]The real time is used as an abstraction for the latency analysis.

a process subsequently wishes to change its reserved bandwidth, it *renegotiates* its reservation parameters according to its new transmission rate.

Our service works within the framework of *Variable Bit-Rate (VBR)* [3] flows, which allows applications to send bursty traffic. In this model, processes reserve an average transmission rate as well as a maximum burst size. Typically, the application declares its transmission rate in bytes per second. For simplicity, we assume that the rate is declared in units of messages per second. Since message size is bounded, these rates correspond closely.

Message sending is divided into time slots of a fixed length, Θ. Θ is the same for all the processes and is fixed throughout the execution. In addition, there exists a constant C which is the number of slots over which the average sending rate is computed. The application declares two rate parameters:

1. $AppAvgRate$ – the average message rate per Θ time. This means that $C * AppAvgRate$ is the maximum number of messages that may be sent during $C * \Theta$ time.

2. $AppMaxBurst$ – the maximum number of messages that may be sent during Θ time.

3.2 The service interface

The application interface of the service consists of the following types of actions:

- $join(r, n)_i$ is used by process i to join the multicast group with incarnation number n, and to reserve QoS. The structure r has two fields: $AppAvgRate$ and $AppMaxBurst$, as explained above. This action is called initially, to establish the transmission rate before any messages are sent.

- $join\text{-}OK_i$ reports to process i that its latest join was successful, and i can now start sending messages.

- $leave_i$ is used by process i to leave the multicast group.

- $leave\text{-}OK_i$ reports to process i that it may safely quit the application. The application is not allowed to perform a $join$ between a $leave$ and the corresponding $leave\text{-}OK$.

- $RN(r)_i$ is used by process i to renegotiate the QoS reserved from the network. The structure of r is as in $join$.

- $mcast(m)_i$ is used by process i to multicast message m to the group. We assume that messages are unique, that is, the same message is not sent more than once. In addition, the message is of a bounded size.

- $deliver(m)_{i,j}$ is used to deliver to process i a multicast message m that was previously multicast by process j.

We say that a message m is sent by a process i when $mcast(m)_i$ occurs, and that i delivers m when the dynamic atomic broadcast service at process i performs the $deliver(m)_{i,j}$ action. We say that process i's incarnation number is n at point t in an execution, if n is the incarnation number in i's latest $join_i$ up to t.

3.3 The service guarantees

The service totally orders message deliveries. In other words, there exists some sequence ordering of all the messages that ever get delivered, such that all processes deliver messages in an order consistent with this sequence.

Definition 1 *Dynamic Atomic Broadcast For each execution of the dynamic atomic broadcast service, there exists a* sequence, $\mathcal{S} = m_1, m_2, \ldots$, *including every message delivered by some process in that execution exactly once*[4]*, so that the following properties hold:*

- Integrity: *A message m is only delivered if it was previously sent, and it is delivered at most once to any particular process.*
- FIFO: *If a process i sends message m before sending m', and both messages are in $\mathcal{S}$, then m is before m' in $\mathcal{S}$.*
- Ordering: *(1) For every process i, and every incarnation number n, the sequence of messages delivered by i with incarnation number n is a contiguous subsequence of $\mathcal{S}$; (2) if i delivers message m with incarnation number n, and i delivers message m' with incarnation number n' where $n < n'$, then m is ordered before m' in $\mathcal{S}$.*
- Liveness
  1. Joining: *If i executes a $join_i$ at some point in the execution and does not subsequently fail or leave, then a $join\text{-}OK_i$ is eventually executed.*
  2. Leaving: *If i executes a $leave_i$ at some point in the execution and does not subsequently fail, then a $leave\text{-}OK_i$ is eventually executed.*
  3. Message Delivery: *If a $join_i$ occurs at some point t in the execution, and i does not subsequently fail or leave, then after point t, i delivers a suffix of $\mathcal{S}$ which includes all the messages that i sends after point t.*

Note that the Ordering property implies the following:

- *Total Order:* If processes i and j both deliver the same two messages m and m', then they deliver these messages in the same order.
- *Reliability:* If processes i and j both deliver the same two messages m and m', and if j delivers both with the same incarnation number n (i.e., a $join_j$ does not occur between $deliver(m)_j$ and $deliver(m')_j$), then j also delivers all the messages that i delivers between m and m'.

In addition to meeting the specification above, our dynamic atomic broadcast service meets QoS (or timeliness) guarantees, which are defined with four parameters - t_1, t_2, t_3, and *AppLatency*, as follows:

Definition 2 *Dynamic Atomic Broadcast QoS(t_1, t_2, t_3, AppLatency)*

- Joining: *If i executes a join at time t, then if i does not fail or leave, a $join\text{-}OK_i$ is executed by $t + t_1$.*
- Leaving: *If i executes a join at time t_0, and a leave at time $t > t_0$, then if i does not fail a $leave\text{-}OK_i$ is executed by $t + t_2$.*

[4]Note that for a finite execution the sequence is finite; otherwise, it may be finite or infinite.

- Message Delivery and Latency: *If a join-OK is executed at i at time t, and i does not subsequently fail or leave, then for every message m sent at time $t' > t + t_3$ by some process j (possibly $j = i$) that does not fail after sending m, i delivers m by time $t' + AppLatency$.*

The maximum latency of the dynamic atomic broadcast service is denoted *AppLatency*; this is the supremum over all executions, all messages m and all processes i of the time since the $mcast(m)_i$ action is performed in some execution until m is delivered by all processes that deliver it.

4 The reliable network

In this paper we build DAB over a *reliable network*, that is, a network that does not lose messages while no failures occur. In [4] we show how a reliable network can be built over an *unreliable network* that guarantees a bounded latency and a bounded loss rate, using a forward error correction (FEC) algorithm. We do not repeat this algorithm here. Rather, we present the semantics of the reliable network, and present its QoS guarantees in terms of the QoS of the unreliable network as proven in [4].

The reliable network interface and semantics. In this section, we use the term *process* to refer to an instance of a program that uses the reliable network at a certain location. The reliable network preserves the FIFO order on messages sent between every pair of processes[5]. The network does not duplicate, corrupt, or spontaneously generate messages. In addition, the network is *reliable*, that is, all messages sent through the network will reach their destination in the absence of failures.

The network supports the reservation of VBR traffic flows. In order to join the multicast group, a process makes a reservation of the bandwidth required for its communication. The interface of the underlying network consists of the following types of actions:

- The $net\text{-}reserve(r)_i$ action is used by process i to join the multicast group and to reserve QoS from the network. The structure r has two fields: *RelNetAvgRate* and *RelNetMaxBurst*, dual to the respective application QoS parameters described in the previous section. This action is called initially, to establish the transmission rate before any messages are sent, and can be subsequently called to renegotiate the QoS reservation.

- The $net\text{-}leave_i$ action is used by process i to leave the multicast group.

- The $net\text{-}rel\text{-}mcast(m, s)_i$ and $net\text{-}rel\text{-}recv(m, s)_{i,j}$ actions are used by process i to reliably multicast and receive messages from the network.

- The $net\text{-}flush(s)$ action is used by process i to tell the network to send all message submitted via $net\text{-}rel\text{-}mcast(m, s)_i$ that it has not yet sent. When $net\text{-}flush(s)_i$ is performed, the network appends a $(\perp, s)$ message to sequence of messages sent by i. This message is received by other processes, in the same way as any other message, via $net\text{-}rel\text{-}recv(m, s)_{i,j}$.

We say that a process i is *alive* if the $net\text{-}reserve(r)_i$ action has been performed and i has not subsequently failed or left.

QoS guarantees. The QoS guarantees of the reliable network are presented in terms of those of the underlying network, as is done in [4]. We therefore begin by stating our assumptions on the QoS of the unreliable network.

[5]Although messages sent over the Internet can sometimes arrive out of FIFO order, this is easy to fix using sequence numbers.

The bandwidth reservation parameters for the underlying network are $NetAvgRate$ and $NetMaxBurst$, and the maximum message latency is Δ. The unreliable network loss rate is bounded as follows [17]: The application specifies a *loss interval*, $x = k + l$, in terms of a number of consecutive messages from the same sender, and a bound, l, on the number of messages sent in the same interval that the network can lose. Specifically, a reservation of l out of $k+l$ guarantees that if a process i multicasts $k+l$ consecutive messages and does not subsequently fail, then every other live process receives at least k of these messages. We assume that the quantities k and l are the same for all processes.

In this paper we are only interested in studying cases in which QoS reservation and renegotiation are successful. Thus, for simplicity, we assume that all reservation requests made by a process are accepted by the network. Typically, QoS reservation and renegotiation take some time for the network to process. However, this time does not affect the message latency and for the sake of the analysis in this paper it is safe to ignore it. Therefore, we assume that once a reservation request is made, the bandwidth that was requested is immediately available to the reserving process.

The reliable network guarantees that if j is alive from $t - \Delta$ and i is alive from t and i performs the *net-rel-mcast*$(m, s)_i$ at time t and a *net-flush*$_i$ at time t' ($t' > t$) then by $t' + \Delta$, *net-rel-recv*$(m, s)_j$ will occur.

Denote by π the maximum time interval between successive *net-flush*$_i$ actions. The maximum latency of the reliable network is then $\Delta + \pi$. As we show in [4], the transmission rate parameters that need to be reserved from the unreliable network in order to meet the reliable network requirements depend on k and l as follows:

$$NetAvgRate = RelNetAvgRate + (\lceil RelNetAvgRate/k \rceil + \Theta/\pi) * l + \Theta/\pi$$

$$NetMaxBurst = RelNetMaxBurst + \lceil RelNetMaxBurst/k \rceil * l * \Theta/\pi$$

These bounds illustrate a tradeoff between the overhead needed to achieve the required bandwidth and the addition to the latency, based on different choices of π.

5 General overview of the DAB algorithm

Our DAB algorithm is composed of two parts: an *Ordering* algorithm and a *Dynamic Failure Manager (DFM).* The Ordering algorithm provides the DAB service to the application. It uses the reliable network to send and receive messages, and uses the DFM to handle failures; the DFM is used only when failures occur. The interfaces among the different parts of the algorithm are shown in Figure 2.

In this section, we present an overview of the DAB algorithm and discuss how the two parts of the algorithm interact. We first explain how the Ordering algorithm uses the DFM. We then present the DFM interface, requirements and guarantees. We discuss the Ordering algorithm in more detail in Section 6, and the DFM in Section 7.

The Ordering algorithm. The Ordering algorithm organizes the delivery of messages into *slots.* In each slot, it delivers messages from processes that are members of the group at that slot. When a process joins (leaves), the Ordering algorithm at the same location sends a special "join" ("leave") message to the other instances of the Ordering algorithm running at different locations. Such messages indicate in which slot the process should be added (removed) from the group. In the absence of failures, such messages are delivered by all processes, so all the processes add (remove) the process at the same slot.

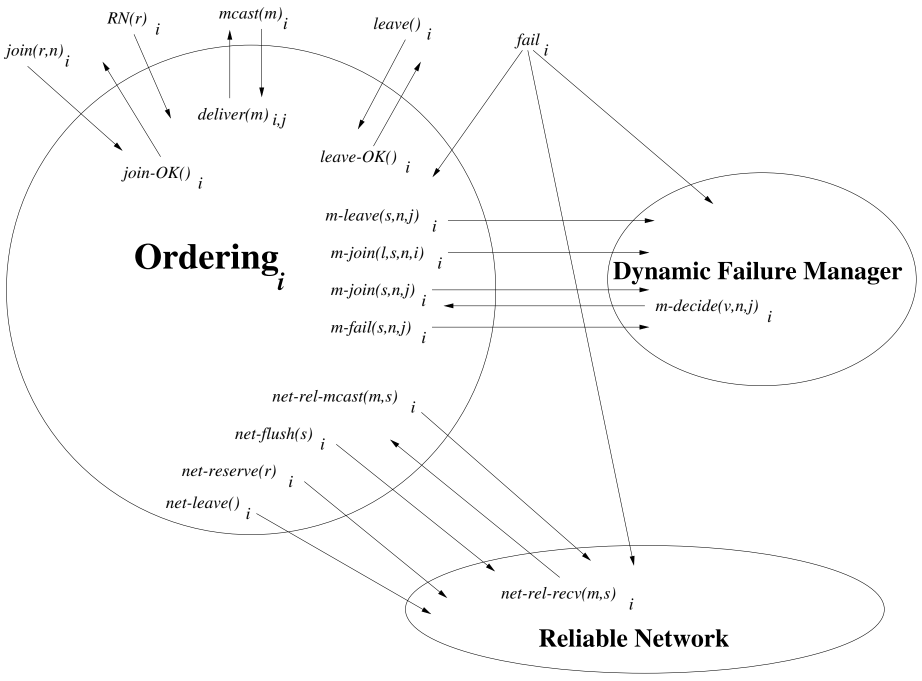


Figure 2: The TO service decomposition.

Failures are more complicated to handle. In our failure model, a subset of the messages sent by a process prior to its crash can reach some live processes while failing to reach others. Thus, although the Ordering algorithm can detect failures using time-outs, such failures are not detected at the same point in the message stream by different instances of the Ordering algorithm. However, all instances of the Ordering algorithm must deliver the same sequence of messages to live processes, and therefore, the algorithm has to make sure that the live processes *agree* on the point (or slot) at which to remove the failing process. This is precisely the role of the DFM: for each process that fails, the DFM allows the current group members to reach agreement upon the slot at which the failed process should be removed.

The DFM interface. The Ordering algorithm at each group member i uses the $m\text{-}fail(s,n,j)_i$ action to notify the DFM at i of the slot s in which i detected process j's failure (for j's incarnation number n). This notification invokes an algorithm to agree upon a slot in which j will be removed. When such agreement is reached, the DFM uses the $m\text{-}decide(v,n,j)_i$ action to notify i that j should be removed at slot v.

The set of processes among which such agreement has to be reached is constantly changing. Therefore, the DFM must keep track of the set of group members as this group evolves over time. To this end, the Ordering algorithm reports process joins and leaves to the DFM using the following actions:

- $m\text{-}join(l,s,n,i)_i$ is used to notify the DFM at i of the fact that the application process at i wishes to join the atomic broadcast algorithm; this action includes the current slot l, the slot s in which i will join, and the incarnation number n. Once this action occurs, we refer to s as $joinslot[i]$.

- $m\text{-}join(s,n,j)_i$ is used to notify the DFM at i of the slot s in which the application process j will join the algorithm; n is the incarnation number of j.

- $m\text{-}leave(n,j)_i$ is used to notify the DFM at i that j has left the algorithm.

The DFM requirements. Since different processes may detect the same failure at different times, different processes may also have different perceptions of the set of group members that have to participate in agreeing upon a failure. The DFM is able to cope with some uncertainty about this set, but not with arbitrary uncertainty: In order to function correctly, the DFM requires its environment to follow certain requirements. A formal specification of these requirements appears in Appendix A; here, we describe them only informally.

There are two major requirements: (1) For a joining process i, every other live process j (that has a smaller *joinslot* value than i) will get a $m\text{-}join(joinslot[i], n, i)_j$ notification before reaching slot $joinslot[i]$, and i will also get a $m\text{-}join(s, n, j)_i$ notification before slot $joinslot[i]$. (2) If two processes j and k suspect that i failed in slots s_j and s_k respectively, then $|s_k - s_j| \leq 1$. Note that this requirement does not restrict the difference in the actual detection time of i's failure by j and k.

These two requirements are reasonable for a dynamic system in which processes use clocks that are slightly skewed, and have access to some sort of failure detector. In particular, they are satisfied by our Ordering algorithm.

The DFM guarantees. Given an environment that meets the requirements above, the DFM service guarantees that all processes *m-decide* upon the same value – the slot in which a failed process should be removed – and that this value is the smallest proposed by any of the processes. These conditions are needed by the Ordering algorithm to ensure that the decision value does not require processes to deliver messages that they do not receive or to refrain from delivering messages they have already delivered. Formally, the DFM service satisfies the following specification:

- **Uniform Agreement:** If $m\text{-}decide(v, n, j)_i$ and $m\text{-}decide(v', n, j)_k$ are performed then $v = v'$.

- **Validity:** If i performs $m\text{-}decide(v, n, j)_i$ then some process k performed $m\text{-}fail(v, n, j)_k$.

- **Minimum Value:** If $m\text{-}decide(v, n, j)_i$ is performed following the $m\text{-}fail(s, n, j)_i$ action, then $v \leq s$.

- **Termination:** If the $m\text{-}fail(s, n, j)_i$ action is performed with $s \geq joinSlot[i]$, and i does not fail or leave, then $m\text{-}decide(v, n, j)_i$ will be performed.

6 The Ordering algorithm

In this section we present the Ordering algorithm. We describe the general operation of the Ordering algorithm in Sections 6.1 through 6.4. We present the algorithm formally as a timed I/O automaton in Figures 3, 4, and 5.

The algorithm divides the time into slots of length Θ. The Ordering algorithm keeps track of the set of processes that it thinks are active in every slot. Below, we explain how this set changes when there are joins, leaves and failures.

6.1 Sending and receiving messages

When the application performs the $mcast(m)_i$ action, the Ordering algorithm adds the slot number s to the message header and performs $net\text{-}rel\text{-}mcast(m, s)$. When a slot s ends (that is, Θ time has passed from the time s started) the Ordering algorithm performs the $net\text{-}flush(s)$ action.

When a message is received via the *net-rel-recv* action, it is stored in a buffer (per source). The algorithm delivers messages from these buffers according to slots, and within each slot, according to the process indices, i.e., it delivers all the messages for this slot sent by process 1, then all the messages sent by 2, etc.

The algorithm does not deliver messages immediately. Before delivering messages from some process j for slot s, it waits to receive all the messages for slot s from all the processes it currently thinks are active, and, in addition, all the messages for slot $s+1$ from processes with indices less than or equal to j. The algorithm identifies the last message sent by j for slot s upon receipt of the $(\perp, s)$ message that is added by the network when $\textit{net-flush}(s)_j$ is performed. This delay ensures that all messages delivered by any process will be received by all other active processes, since our failure model ensures that if a process is active at the end of slot $s+1$, all processes receive its messages for slot s.

6.2 Detecting failures

Processes use timeouts to detect failures. In particular, process i detects that process j has failed if $\Delta + \Gamma$ time after the beginning of slot s according to i's local clock, i has not received j's last message for slot $s-1$.

The reason this implies that j has failed is as follows: According to our algorithm, a *net-flush* occurs every time a slot ends. Also, when the $\textit{net-flush}(s-1)_j$ action occurs, at least one message, $(\perp, s-1)$, is sent. According to the reliable network guarantees, this $(\perp, s-1)$ message reaches i within real time at most Δ after the $\textit{net-flush}(s-1)_j$ occurs. Therefore, since the clock skew is bounded by Γ, if process i waits more than $\Delta + \Gamma$ clock time from the time it begins slot s for a message from process j for slot $s-1$, i knows that j has failed.

Thus, i detects j's failure at a real time which is at most $\Delta + \Gamma + \Theta + \Gamma$ after j had failed. This maximum time can occur if j fails at the beginning of slot s according to j's clock, and j's clock is ahead of i's by Γ.

When i detects the failure of j, i performs the $\textit{m-fail}(s-1, n, j)$ action. The delivery of all messages (starting with j's messages for slot $s-1$) is delayed until the corresponding $\textit{m-decide}(v, n, j)$ action occurs. When an agreement regarding j's failure is reached (the $\textit{m-decide}(v, n, j)$ action), the Ordering process at i resumes the delivery of messages. Process i delivers all of j's messages for slots $s \leq v$, and denotes j as failed from slot $v+1$ onward.

6.3 Joining the algorithm

Processes can join or re-join the algorithm at any time **without delaying messages sent by live processes**. Note that to achieve DAB, all the live processes should agree on the slot in which a joining process is added. This is done by having the Ordering process at the joining location notify all the other processes of the slot in which it will join. Note that the joining process may fail after sending this message. Therefore, the joining slot is chosen to be far enough in the future to allow correct processes to agree on whether or not the joining process will join at this slot.

The joining process. When requesting to join, the application process at i specifies its transmission rate r and incarnation number n. The Ordering process at i then reserves the rate r from the network via the *net-reserve* action. The Ordering algorithm at i computes the current slot which the algorithm is in as: $\textit{curSlot}_i := \textit{clock}/\Theta$ (rounded down to an integer).

Before attempting to join the algorithm, process i waits a predefined amount of time on its local clock until it is sure that all processes have completed any pending agreements for i's previous failures. We discuss the length of this time interval below. After this period, the algorithm computes the slot in which i will join the algorithm. This slot is chosen to be $s_i = curSlot_i + ((f+3)(\Delta+2\Theta)+\Gamma)/\Theta$ (rounded down to an integer). The Ordering algorithm informs the DFM of this slot using the $m\text{-}join(curSlot, s_i, n, i)_i$ action, and sends a ("join", i, s_i, n) message to all other processes. When i's slot number reaches s_i the Ordering algorithm performs the *join-OK* action, and the application can start sending messages.

From the time i wakes up until the ordering process i reaches slot s_i, i does not send or deliver messages. However, from the time i wakes up, it monitors incoming "join" messages from other processes and responds to them the same way active processes respond to such messages, as explained below. In addition, from slot $s_i - 3$ onward, i examines all incoming messages in order to determine which processes it will view as active during slot s_i. In particular, if i does not receive the last message for slot $s_i - 3$, $(\perp, s_i - 3)$, from some process j, then i regards j as failed. If i does receive the last message for slot $s_i - 3$ from j, then i performs an $m\text{-}join(s_i - 3, n, j)_i$ action.

If i receives j's last message for slot $s_i - 2$ but does not receive all of j's messages for slots $s_i - 2$ and $s_i - 1$, then i performs an $m\text{-}fail(s', n, j)_i$ action where s' is the last slot in which it received all of j's messages. In this case, i does not wait for an $m\text{-}decide(v, n, j)_i$ response from the DFM; i simply considers j to be failed when i begins participating in the algorithm in slot s_i. It is safe to do so, since in this case the decision value will be smaller than s_i: since i does not receive all of j's messages for slot $s_i - 1$, by our failure model, no process can receive all of j's messages for slot s_i. Starting from slot s_i, i behaves in the same way as any active process.

Active processes. Each active process j has a *Join* array, in which it keeps track of the slots in which joining processes are to be added. When j receives a ("join", i, s_i) message, it compares $Join[i]$ to s_i. If $Join[i] \neq s_i$, j sets $Join[i] := s_i$, informs the DFM of this join using the $m\text{-}join(s_i, n, i)$ action, and echoes this message by multicasting it to all other processes. If $Join[i] = s_i$, it does nothing. When an active process j reaches slot s_i, j adds i to the list of active processes; thereafter, it delivers messages from i until i fails or leaves.

The echoing of "join" messages ensures that if a process fails immediately after its join, and some live process learns of the join, then all live processes learn of the join. The joining slot s_i is selected so that in the presence of at most f failures, enough time remains prior to s_i for the echo mechanism to ensure that either all the live processes learn of the join or none do. If i fails before s_i, this failure is detected and the DFM agrees upon i's failure slot.

The delay for join. After the application process at i issues a *join*, the Ordering process at location i waits $2\Gamma + 3(\Delta + \Theta) + f(\Delta + \Gamma + 2\Theta)$ before sending a "join" message. The rationale for this delay is the following: As we show in Section 7, the DFM guarantees that at most $\Gamma + 3(\Delta + \Theta) + (f-1)(\Delta + \Gamma + 2\Theta)$ time after the $m\text{-}fail(s, n-1, i)_k$ action is performed, the corresponding $m\text{-}decide(v, n-1, i)_k$ action is performed. Since $m\text{-}fail(*, *, i)$ is performed by all processes at most $2\Gamma + \Delta + \Theta$ after i has failed, the maximum time between i's failure and the time in which the last $m\text{-}decide(v, *, i)$ action is performed is at most: $2\Gamma + 3(\Delta + \Theta) + f(\Delta + \Gamma + 2\Theta)$.

6.4 Leaving the algorithm

When the application at process i performs the *leave* action during slot s, the Ordering algorithm at i performs the *net-rel-mcast*("leave",s)$_i$ action, and notifies its DFM. It then waits until two $net\text{-}flush_i$ actions are executed (i.e., until the end of slot $s+1$ according to i's clock) and then performs the *net-leave*$_i$ and *leave-OK* actions. The delay in notifying the application that it is safe to leave the algorithm ensures that all processes will receive this "leave" message. When process i is *about to deliver* a ("leave", s) message from j, it does not deliver the message. Instead, i removes j from its set of active processes, suspends delivery of messages from j (until j is added again), and performs a $m\text{-}leave(n,j)_i$ action.

$Ordering_i$ **(Algorithm for process i)**

Signature

Input:
- $join(r,n)_i$, m a structure with two integer fields
- $RN(r)_i$, m a structure with two integer fields
- $leave()_i$,
- $mcast(m)_i$, $m \in M$
- $net\text{-}rel\text{-}recv(m,s)_{i,j}$, $m \in M$, s integer
- $m\text{-}decide(v,n,j)_i$ v,n integers, $j \in I$

Output:
- $join\text{-}OK()_i$
- $leave\text{-}OK()_i$
- $net\text{-}flush(s)_i$ s integer
- $net\text{-}rel\text{-}mcast(m,s)_i$ $m \in M$, s integer
- $deliver(m)_{i,j}$, $m \in M$
- $net\text{-}reserve(r)_i$ m a structure with two integer fields
- $net\text{-}leave()_i$
- $m\text{-}join(s,l,n,i)_i$ s,l,n integers
- $m\text{-}join(s,n,j)_i$ s,n integers, $j \in I$
- $m\text{-}leave(n,j)_i$ n integer, $j \in I$
- $m\text{-}fail(s,n,j)_i$ s,n integers, $j \in I$

Time-passing:
- $v(t), t \in R^+$

Internal:
- $send\text{-}join$
- $end\text{-}deliver$
- $failure\text{-}detector$
- $skip\text{-}failed$
- $process\text{-}leave$
- $end\text{-}recvSlot$
- $wait\text{-}start$

State

For all j, $Rqueue(j)$, a FIFO queue of messages, initially empty
$Squeue, mQueue$, FIFO queue of messages, initially empty
$current$, an integer initially 1, // current process to receive from
$myJoin$, an integer initially ∞
$finished$ unbounded array of reals, initially ∞ in all places // the time i finished sendingSlot s
$Join, inc$ array of size n of integers, initially 0 in all places
$maxDeliver$ array of size n of integers, initially -1 in all places
$Sfailed$ array of size n of integers, initially ∞ in all places
$Failed$ a group of process indices, initially empty
$sendingSlot, recvSlot$, integers initially 1,
$rate$, pair of integers initially $\perp$
$changeRate, Pleave, Nleave$, boolean initially FALSE
$lSlot$ integer, initially 0
$state \in \{idle, preJoin, preActive\}$ initially $idle$
$clock \in R^{\geq 0}$, initially 0
$last \in R^+ \cup \{\infty\}$, initially ∞

Figure 3: The Ordering automaton for process i: signature and variables.

Transitions

```
Input $join(r,n)_i$
  Eff: $last = clock + \Gamma + 3(\Delta + \Theta)$
          $(f-1)(\Delta + \Gamma + 2\Theta)$
       $state = preJoin$
       $rate = r$
       $inc[i] = n$

Internal $send\text{-}join$
  Pre: $state = preJoin$
       $clock = last$
  Eff: $changeRate = TRUE$
       $myJoin = (clock + (f+3)(\Delta + 2\Theta) + \Gamma)/\Theta$
       add (“join”,$i, myJoin, inc[i]$) to $Squeue$
       add (“join”,$i, myJoin$) to $mQueue$
       $recvSlot = myJoin$
       for all $j$
          $maxDeliver[j] = myJoin - 4$
       $sendingSlot = \lceil clock/\Theta \rceil$
       $last = (sendingSlot + 1)\Theta$
       $state = preActive$

Output $join\text{-}OK()_i$
  Pre: $sendingSlot = myJoin$
  Eff:

Input $leave()_i$
  Eff: $Pleave$ = TRUE
       add (“leave”,$sendingSlot$) to $Squeue$
       $lSlot = sendingSlot + 2$

Output $net\text{-}leave()_i$
  Pre: $Pleave$ = TRUE
       $lSlot \geq sendingSlot$
  Eff: $NLeave$ = TRUE

Output $leave\text{-}OK()_i$
  Pre: $Nleave$ = TRUE
  Eff:

Input $RN(m)_i$
  Eff: $rate = r$
       $changeRate$ = TRUE

Output $net\text{-}reserve(r)_i$
  Pre: $clock = last$
       $changeRate$ = TRUE
       $r = rate$
  Eff: $changeRate$ = FALSE

Output $m\text{-}join(s,l,n,i)_i$
  Pre: (“join”,$i, l$) is first on $mQueue$
       $inc[i] = n$
       $s = sendingSlot$
  Eff: discard first element of $mQueue$

Output $m\text{-}join(s,n,j)_i$
  Pre: (“join”,$j, s$) is first on $mQueue$
       $inc[j] = n$
  Eff: discard first element of $mQueue$

Output $m\text{-}leave(n,j)_i$
  Pre: (“leave”,$j, s$) is first on $mQueue$
       $inc[j] = n$
  Eff: discard first element of $mQueue$

Output $m\text{-}fail(s,n,j)_i$
  Pre: (“fail”,$j, s$) is first on $mQueue$
       $inc[j] = n$
  Eff: discard first element of $mQueue$

Input $m\text{-}decide(v,n,j)_i$
  Eff: $maxDeliver[j] = v$
       $Sfailed[j] = v + 1$
```

Figure 4: The Ordering algorithm automaton for process i: transition definitions part 1.

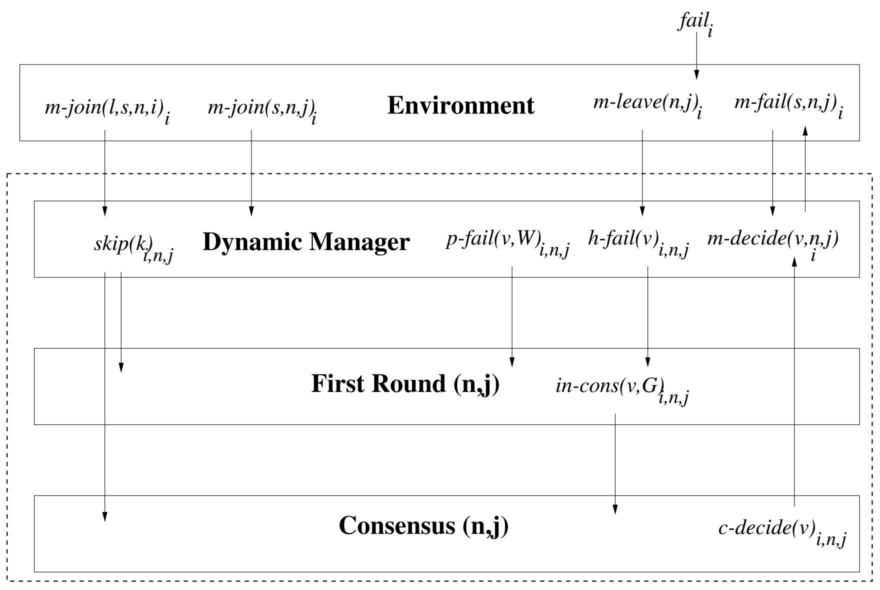
Transitions

```
Input $mcast(m)_i$
    Eff: add $m$ to $Squeue$

Output $net\text{-}rel\text{-}mcast(m, s)_i$
    Pre: $(m)$ is first on $Squeue$
         $s = sendingSlot$
    Eff: discard first element of $Squeue(j)$

Internal $wait\text{-}start$
    Pre: $state = preActive$
         $sendingSlot < myJoin$
         $clock = last$
    Eff: $finished[sendingSlot] = clock$
         $last := clock + \Theta$
         $sendingSlot++$

Output $net\text{-}flush(s)_i$
    Pre: $sendingSlot \geq myJoin$
         $s = sendingSlot$
         $clock = last$
         $changeRate = FALSE$
         $Squeue$ is empty
    Eff: $last := clock + \Theta$
         $finished[sendingSlot] = clock$
         $sendingSlot++$

Input $net\text{-}rel\text{-}recv(m, s)_{i,j}$
    Eff: if $(m = (\text{“join”}, j, s, n))$ then
             if $(Join[j] \neq s)$ then
                 $Join[j] := s$
                 $inc[j] := n$
                 add $(\text{“join”}, j, s)$ to $Squeue$
                 add $(\text{“join”}, j, s)$ to $mQueue$
         elseif $(s \geq myJoin)$ then
             add $(m)$ to $Rqueue(j)$
         elseif $(s = myJoin - 3$ && $m = \perp)$ then
             add $(\text{“join”}, j, s)$ to $mQueue$
             $inc[j] = m.inc$
         if $(s \geq myJoin - 2$ && $m = \perp)$ then
             $maxDeliver[j]++$

Output $deliver(m)_{i,j}$
    Pre: $current > 0$
         $j = current$
         $maxDeliver[j] \geq recvSlot$
         $(m)$ is first on $Rqueue(j)$
         $m \neq \perp$ && $m \neq$ “leave”
    Eff: discard first element of $Rqueue(j)$

Internal $process\text{-}leave$
    Pre: $current > 0$
         $j = current$
         $maxDeliver[j] \geq recvSlot$
         $(m)$ is first on $Rqueue(j)$
         $m =$ “leave”
    Eff: $Failed := Failed \cup j$
         add $(\text{“leave”}, j)$ to $mQueue$
         discard all elements of $Rqueue(j)$
         $current = current + 1 \bmod (n + 1)$

Internal $end\text{-}deliver$
    Pre: $current > 0$
         $j = current$
         $maxDeliver[j] \geq recvSlot$
         $\perp$ is first on $Rqueue(j)$
    Eff: discard first element of $Rqueue(j)$
         $current := (current + 1) \bmod (n + 1)$

Internal $end\text{-}recvSlot$
    Pre: $current = 0$
    Eff: $current := 1$
         $recvSlot++$
         for all $j$ s.t. $Join[j] = recvSlot$ {
             $Failed := Failed \setminus \{j\}$
             $Sfailed[j] = \infty$
         }
         for all $j$ s.t. $Sfailed \geq recvSlot$
             $Failed := Failed \cup \{j\}$

Internal $failure\text{-}detector$
    Pre: $clock \geq finished[maxDeliver[j] + 2] + \Delta + \Gamma$
    Eff: add $(\text{“fail”}, j, (maxDeliver[j] + 1))$ to $mQueue$
         if $(maxDeliver[j] + 1 < myJoin - 1)$ then
             $Failed := Failed \cup j$

Internal $skip\text{-}failed$
    Pre: $current \in Failed$
    Eff: $current := (current + 1) \bmod (n + 1)$

TimePassage $v(t)$
choose $p \geq 0$
    Pre: $now + t - \Gamma/2 \leq clock + p \leq now + t + \Gamma/2$
         $clock + p \leq last$
    Eff: $now := now + t$
         $clock := clock + p$
```

Figure 5: The Ordering algorithm automaton for process i: transition definitions part 2.

7 Constructing the DFM

The DFM's task is to decide upon the slot in which a failed process should be removed from the algorithm. The DFM is composed of three algorithms: the *Dynamic Manager (DM)* which interacts with the DFM's environment; the *First Round* algorithm; and a *Consensus* module which is implemented by any standard uniform consensus algorithm for the fail-stop model. Figure 6 depicts the interaction between the DFM and its environment, as well as its decomposition.



$fail_i$
Environment
$m\text{-}join(l,s,n,i)_i$
$m\text{-}join(s,n,j)_i$
$m\text{-}leave(n,j)_i$
$m\text{-}fail(s,n,j)_i$
Dynamic Manager
$skip(k)_{l,n,j}$
$p\text{-}fail(v,W)_{i,n,j}$
$h\text{-}fail(v)_{i,n,j}$
$m\text{-}decide(v,n,j)_i$
First Round (n,j)
$in\text{-}cons(v,G)_{i,n,j}$
Consensus (n,j)
$c\text{-}decide(v)_{i,n,j}$

Figure 6: The DFM decomposition and interface; the dashed rectangle surrounds the DFM service.

The three components of the DFM. We now explain the roles of the three components of the DFM, and the interaction among them. The DM at location i uses $m\text{-}join_i$ and $m\text{-}leave_i$ inputs from its environment (the Ordering algorithm) in order to keep track of the set of processes that are active at each point in the execution. Whenever a process failure is reported (via $m\text{-}fail(s,n,j)_i$), the DM invokes an instance of the First Round and an instance of Consensus (both are parameterized by n and j). Consensus is used in order to reach agreement on the slot in which j will be removed; it is invoked after the First Round as we now explain.

The difficulty with using a standard uniform consensus algorithm is that although such algorithms can tolerate failures, they cannot cope with processes joining an in-progress execution of the algorithm. When the Consensus module is invoked, it must be informed of a set of processes, G, that will participate in this invocation of Consensus unless they fail. We now explain how the DM and First Round overcome this difficulty.

The DM tracks joins and leaves in order to be able to suggest an initial estimate, W, of the set of processes that will participate in each invocation of Consensus unless they fail. This set includes the processes that join before or at the slot following the slot in which the failure was detected. However, since instances of the DM running at different locations can detect the same failure as happening at different slots (otherwise there would be no need for Consensus), they can have different values for W. For example, if some process i_1 fails, and its failure is detected as happening in slot 7 by process i_2, the same failure can be detected as happening in slot 6 by process i_3. If a fourth process, i_4 joins the algorithm at slot 8, process i_2 includes i_4 in W (since from i_2's point of view, i_4 could have detected i_1's failure as happening in slot 8, in which case i_4

would have to participate in the Consensus), whereas process i_3 does not.

One purpose of the First Round is to eliminate such uncertainties, and to provide the Consensus module at the local process with a set of processes, G, that will participate in this invocation of Consensus unless they fail. In order to overcome such uncertainties, processes like i_4 in the example above have a special role in the First Round, they *help* processes that like i_2 expect them to participate in the First Round (to avoid blocking), but they do not subsequently participate in the Consensus algorithm (i.e., they are not included in G). The First Round therefore has two types of input actions: $p\text{-}fail$ for participants, and $h\text{-}fail$ for helping processes.

A second purpose of the First Round is to modify the input value for Consensus in order to have the decision value meet the Minimum Value guarantee of the DFM. The output of the First Round is the input for Consensus.

Both the First Round and the Consensus need to detect failures in order to avoid waiting for messages from a failed process. They do not implement such failure detection internally. Instead, the DM notifies in-progress executions of these algorithms of a failure (or leave) of a process k via the $skip(k)$ action.

The Dynamic Manager.

The DM algorithm is presented as an I/O automaton in Figure 7. We now describe its operation. The DM uses *m-join* and *m-leave* actions to keep track of the active processes, as well as of the slot, $joinSlot[j]$, in which each process j joins.

When $m\text{-}fail(s,n,j)_i$ occurs, the DM at process i compares $joinSlot[i]$ with s and acts as follows: If $s \geq joinSlot[i]$, then it performs $p\text{-}fail(s,W)_i$, where W is the set of all processes k that i thinks are active, and for which $joinSlot[k]$ is less than or equal to $s+1$. If $s < joinSlot[i]$, then the DM at i performs the $h\text{-}fail(s)_i$ action.

Whenever an $m\text{-}fail(*,*,k)$ or an $m\text{-}leave(*,k)$ action is performed, the DM performs the $skip(k)$ for all ongoing consensus and First Round algorithms. When the consensus algorithm reaches an agreement, it notifies the DM using the *decide* action. The DM, in turn, notifies its environment of this decision using the *m-decide* action.

The following lemma (proven in the Appendix) discusses the relationship between the input sets of different processes that perform the $p\text{-}fail$ action. It is used to prove the correctness of the DFM.

Lemma 7.1 *Let $P[n,j]$ be the set of processes i that perform $p\text{-}fail(s^i,W^i)_{i,n,j}$. For all $i \in P[n,j]$:*
1. $m\text{-}fail(*,*)_{i,n,j}$ is performed exactly once.
2. If $k \in P[n,j]$ and $k \notin W^i$, then k either fails or leaves after performing the $p\text{-}fail(s^k,W^k)_{k,n,j}$ action, and before i performs the $p\text{-}fail(s^i,W^i)_{i,n,j}$ action.
3. For all $k \in W^i$, either k performs $p\text{-}fail(s^k,W^k)_{k,n,j}$ or $h\text{-}fail(s^k)_{k,n,j}$, or $skip(k)_i$ occurs, or i fails or leaves.

$DynamicManager_i$ **(Algorithm for process i)**

Signature

```
Input:
    $m\text{-}join(l,s,n,i)_i$  $l,s,n$ integers
    $m\text{-}join(s,n,j)_i$  $s,n$ integers, $j \in I$
    $m\text{-}leave(n,i)_i$  $n$ integer
    $m\text{-}leave(n,j)_i$  $n$ integer, $j \in I$
    $m\text{-}fail(s,n,j)_i$  $s,n$ integers, $j \in I$
    $c\text{-}decide(s,n,j)_i$  $s,n$ integers, $j \in I$
Output:
    $decide(s,n,j)_i$  $s,n$ integers, $j \in I$
    $skip(k)_{i,n,j}$  $k \in I$
    $p\text{-}fail(v,W)_{i,n,j}$  $v$ integer, $W \subseteq I$
    $h\text{-}fail(v)_{i,n,j}$  $v$ integer
Internal:
```

State

joinSlot array of size N of integers, initially 0 in all places

status, array of size $N * N$, each element is $\in \{inactive, active\}$ initially *inactive* in all places.

newCons, *curCons* a set of pairs (n,j) where n is an integer and $j \in I$ initially empty // The set of pending and current consensus algorithms respectively.

res a set of triplets (v,n,j) where v,n are integers and $j \in I$, initially empty

consQueue an unbounded two dimensional array, each element in this array is a queue, initially empty in all places

part, *help* an unbounded two dimensional arrays of booleans, initially FALSE in all places.

R an unbounded two dimensional array, each element of this array is a subset of I, initially empty in all places

V an unbounded two dimensional array, each element of this array is an integer

Transitions

```
Input $m\text{-}join(l,s,n,i)_i$
    Eff:  $status[i][i] := active$
          $joinSlot[i] := s$

Input $m\text{-}join(s,n,j)_i$, $j \neq i$
    Eff:  $status[i][j] := active$
          $joinSlot[j] := s$

Input $m\text{-}leave(n,i)_i$
    Eff:  exit

Input $m\text{-}leave(n,j)_i$, $j \neq i$
    Eff:  $status[i][j] := inactive$
          for all $(n',j') \in curCons$
              add $skip(j)$ to $consQueue[n'][j']$

Output $skip(k)_{i,n,j}$
    Pre:  $skip(k)$ is first on $consQueue[n][j]$
    Eff:  discard first element of $consQueue[n][j]$

Input $c\text{-}decide(v,n,j)_i$
    Eff:  $curCons = curCons \setminus (n,j)$
          $res = res \cup (v,n,j)$

Output $decide(v,n,j)_i$
    Pre:  $(v,n,j) \in res$
    Eff:  $res = res \setminus (v,n,j)$

Input $m\text{-}fail(s,n,j)_i$, $j \neq i$
    Eff:  $status[i][j] := inactive$
          for all $(n',j') \in curCons$
              add $skip(j)$ to $consQueue[n'][j']$
          for all $k$ s.t. $(joinSlot[k] \leq s+1$
                  && $status[i][k] = active)$
              $R[n][j] = R[n][j] \cup k$
          $V[n][j] = s$
          if $(s \geq joinSlot[i])$ then
              $part[n][j] = TRUE$
              $newCons = newCons \cup (n,j)$
          else
              $help[n][j] = TRUE$

Output $p\text{-}fail(v,W)_{i,n,j}$
    Pre:  $(n,j) \in newCons$
          $W = R[n][j]$
          $v = V[n][j]$
          $part[n][j] = TRUE$
    Eff:  $part[n][j] = FALSE$
          $newCons = newCons \setminus (n,j)$
          $curCons = curCons \cup (n,j)$

Output $h\text{-}fail(v)_{i,n,j}$
    Pre:  $v = V[n][j]$
          $help[n][j] = TRUE$
    Eff:  $help[n][j] = FALSE$
```

Figure 7: The Dynamic manager algorithm

The First Round algorithm. We describe the operation of the first round algorithm. The algorithm is presented as an I/O automaton in Figure 8. The First Round algorithm is invoked every time a process is detected as failed, by one of two actions: *h-fail* or *p-fail*. When the algorithm is initiated by $h\text{-}fail(v)_i$, the DFM at location i multicasts a ("help",v) message to all processes and terminates. If it is initiated by $p\text{-}fail(v, W)_i$, it multicasts a "first round" message containing v and W and waits for messages from every other process in W.

When the First Round at i receives messages from all processes $k \in W$ for which no $skip(k)$ action was performed, it performs $inCons(v, G)$, where v is the lowest value in any of the received messages (including i's initial value), and G is the set of processes that a "first round" message was received from, including i. The $inCons_i$ actions is performed no sooner than $\Delta + \Theta$ time after i sends its First Round message. This ensures that the message it sent had reached all processes before the uniform consensus is triggered.

In the Appendix, we prove formally that the First Round achieves its two purposes (eliminating uncertainties in the participants group and Minimum Value). In addition, we prove that combined with the DM it guarantees termination.

Thus, all the processes that do not fail or leave start the consensus algorithm. In addition, all the processes in each process' set G participate in this invocation of Consensus unless they leave or fail, and all the processes that participate are in each process's set G. Thus, any standard uniform consensus algorithm can be used, and the combined service achieves the DFM guarantees, as we now argue.

Correctness and latency. We prove that the DFM meets its specification (stated in Section 5). In addition, we present lemmas that establish its latency bounds. The formal proofs may be found in the Appendix.

Theorem 1 *The DFM achieves the following guarantees:*
1. Uniform Agreement.
2. Validity.
3. Minimum Value.
4. Termination.

Proof: 1. As shown above, the First round algorithm achieves the spec of any consensus algorithm, and thus the uniform agreement is guaranteed by combining this property of the first round algorithm with the uniform agreement guarantee of the consensus algorithm.
2. From the validity of the consensus algorithm we know that the output of such a consensus must be the input of one of the processes that participated in it. But inputs to the consensus algorithm can only come from processes that performed the $m\text{-}fail(s, *, *)$ action or from messages from such processes in which the value in the message is s. Thus, if i performs the $m\text{-}decide(v, n, j)_i$ action, there must be some process k that performed the $m\text{-}fail(v, n, j)_k$ action.
3. Combining the validity condition of the consensus algorithm with the Minimum Value guarantee of the First round algorithm guarantees the minimum value requirement (note that only processes k such that $k \in C[n, j]$ can perform the $m\text{-}decide(v, n, j)_k$ action according to the DFM algorithm).
4. By combining the termination guarantee of the First round with the termination guarantee of any consensus algorithm the termination guarantee of the DFM is proved.
■

We now analyze the time it takes from when a $m\text{-}fail(s, n, j)_i$ action is performed until the corresponding $m\text{-}decide(v, n, j)_i$ is performed. Note that this is exactly the time it takes to perform the First round and Consensus algorithm since when an *m-fail* action occurs the DFM starts a First round algorithm, and when the consensus algorithm terminates the *m-decide* action is performed. The following lemma discusses this time as a function of the number of processes f that can fail during the execution of the First round and Consensus algorithms.

Lemma 7.2 *If f processes fail during the execution of the First round and Consensus algorithms, then the algorithm takes at most:*

$$\Gamma + \Theta + 3(\Delta + \Theta) + f(\Delta + \Gamma + 2\Theta)$$

We now move to prove the second time guarantee of the DFM in the following lemma.

Lemma 7.3 *If $m\text{-}fail(s, n, j)_i$ is performed at time t_0, and starting at time $t > t_0$ no $m\text{-}fail$ action was performed by any process for $3(\Delta + 2\Theta + \Gamma)$ time, m-decide$(v, n, j)_i$ will be performed by $t + 3(\Delta + 2\Theta + \Gamma)$.*

$FirstRound_i$

Signature

```
Input:
    $p\text{-}fail(v, W)_{i,n,j}$ $v$ integer, $W \subseteq I$
    $h\text{-}fail(v)_{i,n,j}$ $v$ integer
    $skip(k)_{i,n,j}$ $k \in I$
Output:
    $inCons(v, G)_{i,n,j}$ $v$ integer, $W \subseteq I$
Internal:
    $send(v, str)_{i,k,n,j}$ $v$ an integer
    $receive(v, str)_{k,i,n,j}$ $v$ an integer
```

State

S, C a set of processes indices, initially empty.

$mode \in \{idle, active\}$ initially $idle$

$curVal$ integer

for each $j \in I$ $queue(j)$, a queue of messages initially empty.

Transitions

```
Input $p\text{-}fail(v, W)_{i,n,j}$
    Eff: $S = W$
         $C = W$
         for all $j \in I$
             add $(v,$"part"$)$ to $queue(j)$
         $mode = active$

Input $h\text{-}fail(v)_{i,n,j}$
    Eff: for all $j \in I$
             add $(v,$"help"$)$ to $queue(j)$

Input $skip(k)_{i,n,j}$
    Eff: if$(mode = active)$ then
             $S = S \setminus k$
             $C = C \setminus k$

Output $inCons(v, G)_{i,n,j}$
    Pre: $mode = active$
         $S$ is empty
         $v = curVal$
         $G = C$
    Eff: $mode = idle$

Internal $send(v, str)_{i,k,n,j}$
    Pre: $(v, str)$ is first on $queue(k)$
    Eff: discard first element of $queue(k)$

Internal $receive(v, str)_{k,i,n,j}$
    Pre:
    Eff: if$(curVal > v)$ then
             $curVal = v$
         $S = S \setminus k$
         if$(str =$ "help"$)$ then
             $C = C \setminus k$
```

Figure 8: The First round algorithm

8 Correctness

This section has two parts. We first give an informal proof showing that our algorithm achieves our service specification. Next we briefly show how the Ordering algorithm achieves the DFM environment requirements. The full details of the proofs appear in the Appendix.

Achieving the service specification First, we need the following lemma that shows that all active processes add a joining process j in the same slot.

Lemma 8.1 *If some process i that is active at slot s adds j in that slot, then all active processes add j at slot s.*

In addition, we also use the following lemma:

Lemma 8.2 *If a m-decide(v,n,j) action occurred following a $m\text{-}fail(s,n,j)$ action then $0 \leq (v-s) \leq 1$, and this bound is tight.*

We are now ready to prove that our algorithm achieves the service specification. We first describe the ordering of $\mathcal{S}$. We order $\mathcal{S}$ in the following way - For every message m that was delivered by some process, the triplet (m_s, m_i, m_p) consists of the slot m_s in which m has been sent, the sender process index (m_i) and the place of m within the messages i sent for slot s (m_p). We order all messages using these triplets (that is, m^1 is ordered before m^2 if $m_s^1 < m_s^2$, and if $m_s^1 = m_s^2$ we compare m_i^1 and m_i^2 and so on). Note that this is a complete ordering on the messages.

Integrity is trivially satisfied from our assumption that the network does not duplicate, corrupt or spontaneously generate messages. The fifo of two messages m and m' sent by the same process is guaranteed from the way we construct $\mathcal{S}$.

When no process fails, messages are delivered in each slot according to process indices by all processes. When a process leaves all processes stop delivering its messages after the last message it sent before the "leave" message. As shown above, when a process i joins, all processes add i in the same slot. Thus, all processes order the delivery exactly as in $\mathcal{S}$. Since there are no process failures, using the reliable network guarantees that all processes receive all messages sent, and so each process delivers a contiguous subsequence of $\mathcal{S}$. Since $\mathcal{S}$ is ordered by the slot number, and each time a process joins the slot number from which it starts delivering messages ($joinSlot_i$) increases, it must be that any message m, i delivers while i has incarnation number n, is ordered ahead in $\mathcal{S}$ of any message i delivered while i had incarnation number n' where $n' < n$.

When a process j fails, then using the DFM all processes agree on the slot S_D in which j failed. It follows from the DFM guarantees that $S_D \leq s_i$ for all processes i that performed $m\text{-}fail(s_i, n, j)_i$. Thus, all correct processes have received all of j's messages up to S_D, and no process had delivered messages from j for slots greater than S_D (this results from Lemma 8.2 and the one slot delay we enforce on message delivery). Since messages are still delivered in each process according to their order in $\mathcal{S}$, the total ordering is guaranteed.

The Joining and Leaving liveness requirements are guaranteed since our algorithm performs the *join-OK* a fixed time after the *join* action is performed, and the same thing holds for *leave* and *leave-OK*.

As for Message Delivery and Latency, If m is sent by some process j more than Γ time after the *join-OK*$_i$ was performed, m must have a slot number greater than or equal to $joinSlot_i$ since the clock skew between all processes is at most Γ. Thus, if i and j do not fail m will be delivered by i, because i delivers all messages it receives with slot number greater or equal to $joinSlot_i$ (except for some of the messages from faulty processes). The maximal latency is also guaranteed, and we discuss it in Section 9.

Achieving the DFM requirements The two DFM requirements are achieved by the Ordering algorithm. First, using Lemma 8.1 we know that when j joins, all active processes are aware of this join before they reach $joinSlot_j$. In addition, according to our joining algorithm, j knows of all active processes when it reaches $joinSlot_j$, since j monitors messages prior to its actual join.

As for the second requirement, our failure model guarantees that if j fails there will be at most one slot difference between the the last message i received from j and the last message k received from j. Thus, i and k will detect j's failure in at most one slot difference.

9 QoS guarantees

If no process fails, then the maximum delay caused by this algorithm is the following:

Lemma 9.1 *If no process fails during the execution of the algorithm then*

$$AppLatency = \Delta + \Gamma + 2\Theta$$

Proof: Assume that process i sends a message m in slot s, and the delivery of m is delayed until a message m' from another process, j, will be received. Since message delivery is done per slot, m' must be a message from j for slot at most $s+1$ (only messages sent for slots $s' \leq s+1$ can delay the delivery of m). Since the difference between the two processes' internal clock is at most Γ, we know that i sent m at most $\Gamma + 2\Theta$ time before j sent its last message for $s+1$. Since j's messages for slot $s+1$ arrives at all the processes at most Δ time after j ends slot $s+1$, all the processes receive m' at most Δ time after j ended slot $s+1$. Thus, after at most $\Delta + \Gamma + 2\Theta$ time from the time i sent m, it will be enabled for delivery by all processes. ■

We now turn to analyze the effect of process failures on message delivery time. The following lemma discusses the maximum delay caused by a the algorithm as a function of the number of processes that fail during the execution of the algorithm.

Lemma 9.2 *Denote by $f-1$ the number of processes that fail between a $m\text{-}fail(s,n,j)_i$ and a $m\text{-}decide(v,n,j)_i$ actions, then*

$$AppLatency = 4\Theta + 3\Gamma + \Delta + 3(\Delta + \Theta) + (f-1)(\Delta + \Gamma + 2\Theta)$$

Proof: If s_j is the slot j failed in, then the the failure of j can delay the delivery of messages that where sent at slot $s_j - 2$ and up. This is so because all processes saw js messages for slot $s_j - 2$ and thus delivered all messages for slot $s_j - 3$. However, it could be that some process did not receive all of js messages for slot $s_j - 1$, delaying the delivery of all messages sent for slot $s_j - 2$ from processes with indices $k \geq j$.

As explained in Section 6.2, at most $\Gamma + \Theta + \Delta + \Gamma$ after the first process started slot s_j, all processes detect that j failed, and perform the $m\text{-}fail(s,n,j)$ action. According to the DFM guarantees, at most $\Gamma+\Theta+3(\Delta+\Theta)+(f-1)(\Delta+\Gamma+2\Theta)$ time after a $m\text{-}fail(s,n,j)_i$ is performed, the corresponding $m\text{-}decide(v,n,j)_i$ is performed. Thus the total time the algorithm can delay a message is:
$2\Theta+\Gamma+\Theta+\Delta+\Gamma+\Gamma+\Theta+3(\Theta+\Delta)+(f-1)(\Delta+\Gamma+2\Theta) = 4\Theta+3\Gamma+\Delta+3(\Delta+\Theta)+(f-1)(\Delta+\Gamma+2\Theta)$.
■

Note that although the maximum delay time scales with the total number of processes that can fail, in practice we would not expect this delay to occur. The maximum delay will occur only if at least one process will fail every $\Delta + \Theta + \Gamma$ time. Such scheduling of process failures is very rare in practice, and so the expected maximum delay of our algorithm is much smaller than the one we just described. The following lemma discusses the latency suffered by a message m for which no processes fails during a certain time before and after it is sent.

Lemma 9.3 *If message m is sent by i at time t, and no process fails between $t - (\Delta + \Theta + 2\Gamma + 3(\Delta + 2\Theta + \Gamma))$ and $t + (\Delta + 2\Theta + \Gamma)$, then all processes deliver m by $t + \Delta + 2\Theta + \Gamma$.*

Proof: According to our algorithm (Section 6.2), if a process k fails, i will perform the $m\text{-}fail(*, *, k)_i$ action at most $\Delta + \Theta + 2\Gamma$ after ks failure. Since no process failed between $t - (\Delta + \Theta + 2\Gamma + 3(\Delta + 2\Theta + \Gamma))$ and t, we know that no $m\text{-}fail$ action was performed by i between $t - (3(\Delta + 2\Theta + \Gamma)$ and t. According to the DFM guarantees, if $m\text{-}fail(s, n, j)_i$ is performed at time t_0, and starting at time $t' > t_0$ no $m\text{-}fail$ action was performed by any process for $3(\Delta + 2\Theta + \Gamma)$ time, all processes will perform the *m-decide* action by $t' + 3(\Delta + 2\Theta + \Gamma)$. Since no $m\text{-}fail$ action was performed by any process between $t - (3(\Delta + 2\Theta + \Gamma)$ and t, a corresponding $m\text{-}decide(*, *, j)_k$ action was performed at all processes for any process j for which a $m\text{-}fail(*, *, j)_k$ was performed. Thus, when m was sent all messages that where delayed until a decision regarding a process failure was made, have been delivered.
According to our reliable network guarantees m will be received by all processes at most Δ time after it was sent. Assume m was sent for slot s. Then, after receiving m all processes must receive all messages for slot $s + 1$ in order to deliver m. Since m was sent for slot s, at most $2\Theta + \Gamma$ time after m was sent, all processes finished slot $s + 1$, and since no process has failed between t and $t + \Delta + 2\Theta + \Gamma$ all these messages (for slot $s + 1$) will arrive at all processes at most Δ time after they where sent. So m will be delivered by all processes at most at time $t + \Delta + 2\Theta + \Gamma$. ■

Reserved rates. The messages our ordering algorithm adds over the messages sent by the application are the messages added by the reliable network layer and messages added by the DFM. The DFM sends at most one message every slot (all of the DFM messages are aggregated and are sent once a slot). In addition, our algorithm issues a *net-flush* every Θ time, thus π of Section 4 is set to Θ. We thus get the following upper bounds on the average and maximum rates used by our algorithm:

$$NetAvgRate = AppAvgRate + 1 + (\lceil (AppAvgRate + 1)/k \rceil + 1) * l$$

$$NetMaxBurst = AppMaxBurst + 1 + (\lceil (AppMaxBurst + 1)/k \rceil) * l$$

10 When the network QoS guarantees are violated

So far, we assumed that the QoS guarantees provided by the network (namely, bounded delay and bounded message loss) are deterministic. However, some networks only provide probabilistic QoS guarantees. In such networks, there may be periods of time during which the QoS guarantees are violated. Although our algorithm cannot guarantee atomic broadcast while the QoS guarantees are violated, it is important for the algorithm to be able to recover from such violation. In other words, a certain time after the QoS guarantees are restored, the algorithm should again be able to provide the DAB service. In addition, it would be desirable to inform the application when a violation of DAB semantics occurs, and when the correct semantics are resumed (following the failure awareness approach of [9]).

We now discuss how QoS violations can affect our algorithm, and how the algorithm can be modified in order to recover from them. We discuss these ideas informally, a more careful study of these ideas remains for future work.

Violation of QoS guarantees may lead one correct process, i, to detect a second correct process, j, as faulty (either due to message delay or due to loss). Process i will then invoke the DFM to agree upon j's failure slot. In order to recover from this situation, we modify the algorithm to have j fail itself when it gets a consensus or a first round message related to its own failure. The algorithm at j will then notify the application of the failure, and the application would have to re-attempt to join (a similar approach was taken in [2]).

Not every case of excessive message loss must lead to incorrect failure detection. In other cases, the loss can be detected when the reliable network delivers messages with gaps. In such cases, the application can be informed of the fact that loss occurred. Our algorithm, without modification, recovers from such situations.

The loss or delay of a "join" message may lead to a process joining before all the other processes know of the join. To recover from this situation, we modify the algorithm to have processes monitor all incoming messages. If process j sees a message from process i when j thinks that i is not active, j adds i to the list of active processes and delivers messages from i from now on. It also informs the application of the potential loss of messages. Once the network QoS guarantees are restored, all such message will arrive at j, and DAB semantics will be resumed.

The loss of a DFM message (consensus or first round) may lead to blocking. To overcome this, we have processes periodically re-send their latest DFM message. Once the network guarantees are restored, these messages will reach their destinations.

11 A lower bound for DAB with process failures

The following theorem shows that in the model studied in this paper, any algorithm implementing DAB (see Definition 1 in Section 3) can guarantee, at best, a latency bound which is proportional to the number of failures it can tolerate.

Theorem 2 *A Dynamic Atomic Broadcast (DAB) algorithm that can tolerate f process failures cannot guarantee a latency bound smaller than $(f+1)\Delta$.*

Proof: Assume that a DAB algorithm $\mathcal{A}$ can tolerate f process failures and guarantees a latency bound of δ. We now show that $\delta \geq (f+1)\Delta$. As shown in [12], the processes may use $\mathcal{A}$ to solve the Consensus problem by sending their initial values as their first message and agreeing upon the value in the first delivered message. By our assumption on $\mathcal{A}$, this message is delivered at most δ time after the algorithm is initiated, and thus, Consensus is solved in δ time.

Since $f+1$ rounds is a well known lower bound for synchronous Consensus tolerating f stopping failures (see [14], Ch. 6.7), and from our assumption that messages can be delayed up to Δ time by the network, we conclude that the algorithm cannot guarantee that Consensus be solved in less than $(f+1)\Delta$ time, and hence $\delta \geq (f+1)\Delta$. ■

12 Conclusions

We have designed a *Dynamic Atomic Broadcast (DAB)* algorithm that preserves QoS guarantees. We have conducted a detailed theoretical study of the QoS guarantees of our algorithm under different circumstances. In particular, we have shown that in periods with no failures, the latency for the ordered multicast is within a constant of the latency of the underlying network (independently of the number of participants). This is an improvement over the latency exhibited by previous algorithms (e.g., [2]). When failures do occur, the latency is linear in the number of processes that fail within a bounded time interval, as dictated by a lower bound.

We have discussed possible ways of extending our algorithm to recover from situations in which the network QoS is violated. Future work will further develop these ideas, and present a careful study of the time it takes the algorithm to recover from such situations.

Our algorithm uses a Dynamic Failure Manager to achieve atomic semantics in the presence of failures: the DFM reaches consensus regarding the point at which each failed process should be removed from the algorithm. Achieving such consensus is difficult because of the dynamic model: Processes can join and leave the algorithm at any time. Furthermore, different processes can detect the same failure while having different perceptions of the set of processes participating in the algorithm. The DFM service resembles virtually synchronous group membership; its use by the DAB service resembles the use of group membership in totally ordered group communication services[6]. However, unlike most group communication systems providing similar services, (e.g., [10, 21, 6]), using our DFM processes can join and leave the algorithm without introducing delays to the communication among active processes. We believe that our DFM may be useful for additional applications, beyond DAB. Future work will explore this possibility.

Acknowledgements

We are grateful to Alan Fekete and Alex Shvartsman for numerous helpful discussions and suggestions. We thank Tal Anker for suggesting the study the QoS of totally ordered multicast as a research topic.

References

[1] *10Six - A massive online team play over the Internet.* URL: http://www.tensix.com/ , http://www.heat.net/10sixchannel/index.html.

[2] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1996.

[3] The ATM Forum Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification Version 4.0, af-sig-0061.000*, July 1996.

[4] Z. Bar-Joseph, I. Keidar, T. Anker, and N. Lynch. QoS preserving totally ordered multicast. In *5th International Conference On Principles Of DIstributed Systems (OPODIS)*, December 2000. To appear.

[6]Although our DAB algorithm does not export group membership views to the application, it uses the DFM to manage such views internally. The algorithm could therefore easily be modified to provide applications with such views.

[5] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3), 1984.

[6] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246, June 1998.

[7] J. Crowcroft, M. Handley, and I. Wakeman. *Internetworking Multimedia.* UCL Press, September 1999. Available from: http://www.cs.ucl.ac.uk/staff/j.crowcroft/mmbook/book/book.html.

[8] X. Défago, A. Schiper, and P. Urbán. Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey. Technical Report DSC/2000/036, Swiss Federal Institute of Technology, Lausanne, Switzerland, September 2000.

[9] C. Fetzer and C. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *27th Annual International Fault-Tolerant Computing Symposium*, pages 282–291, 1997.

[10] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE International Symposium on High Performance Distributed Computing*, 1997. Also Technical Report 95-1527, Department of Computer Science, Cornell University.

[11] L. Gautier and C. Diot. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In *Proceedings of IEEE Multimedia Systems*, June 28 - July 1 1998. URL http://www-sop.inria.fr/rodeo/MiMaze/.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *chapter in: Distributed Systems.* ACM Press, 1993.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 78.

[14] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[15] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[16] S. McCanne. *A Distributed Whiteboard for Network Conferencing.* UC Berkeley CS Dept., May 1992. Unpublished Report. Software available from ftp://ftp.ee.lbl.gov/conferencing/wb.

[17] C. Partridge. *A Proposed Flow Specification, RFC 1363*, September 1992. Internet Engineering Task Force, Network Working Group.

[18] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[19] S. Shenker, C. Partridge, and R. Guerin. *Specification of Guaranteed Quality of Service, RFC 2212*, September 1997. Internet Engineering Task Force, Network Working Group.

[20] The UCL Networked Multimedia Research Group. *NTE: Network Text Editor.* URL: http://www-mice.cs.ucl.ac.uk/multimedia/software/nte/.

[21] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer Verlag, 1995. LNCS 938.

We now present a formal correctness proof of our service. We begin in Section A by presenting a formal specification of the DFM environment. In Section B we then show that given a DFM meeting the DFM guarantee presented in Section 5, the Ordering algorithm provides the DAB service specification of Section 3. We also show that the Ordering algorithm meets the DFM environment specification of Section A. Finally, in Section C we show that the DFM meets its specification.

A DFM environment specification

The DFM environment specification is composed of two parts, safety and liveness. The safety requirements are presented as an automaton in Figure 9.

We now explain briefly each of the preconditions in the environment specification. A precondition for the $m\text{-}join(l, s, n, i)_i$ action should guarantee that when i joins, all processes k have enough time to perform the $m\text{-}join(s, n, i)_k$ before they reach slot s. Thus, we require that s will be bigger than the current slot of all active processes. We also require that all processes know that i was not active prior to performing the *m-join* action.

The $m\text{-}join(s, n, j)_i$ preconditions should guarantee that if i is about to be joined when s arrives, i will add j before i reaches slot s (that is, i performs this action when $slot[i] < s$).

The *m-leave* action preconditions guarantee that a process will not be reported as left unless it actually left. In addition, a process can be reported to leave only if it has been seen as active prior to this report.

The *m-fail* action precondition guarantees that a process will not be reported as fail unless it actually failed. In addition, the precondition limits the difference (in slots) between the slots in which different processes detect the failure of process j. This achieves the second intuitive requirement stated in Section 5.

The preconditions for the *inc-slot* action guarantees that when i actually joins, all processes that where supposed to join before i know of is joining, and i knows of them. This achieves the first intuitive requirement of Section 5.

The DFM liveness requirements are:

Liveness:

- For all processes k such that $status[k][k]$ is *active* when the $fail_j$ action is performed for j with incarnation number n, and $joinSlot[k] \leq failSlot[j] + 2$, k either $m\text{-}fail(*, n, j)_k$, or $fail_k$ or $m\text{-}leave(*, inc[k], k)_k$ is performed.
- All processes that have $status[i][k] = active$ when k performs the $m\text{-}leave(n, k)_k$ action, will set $status[i][k]$ to *inactive*.

The first liveness item guarantees that if a process k was active when a process j failed, and k was about to join the algorithm or have already joined, k will either report the failure of j or will fail or leave itself. The second guarantees that when an active process j leaves, all other active processes will note that fact.

$DFMEnv$

Signature

Input:

- $fail_i$
- $m\text{-}decide(v,n,j)_i$

Output:

- $m\text{-}join(l,s,n,i)_i$ l,s,n integers
- $m\text{-}join(s,n,j)_i$ $j \neq i$, s,n integers, $j \in I$
- $m\text{-}leave(n,j)_i$ n integer, $j \in I$
- $m\text{-}fail(s,n,j)_i$ s,n integers, $j \in I$

Internal:

- $inc\text{-}slot()_i$

State

$slot$ array of size N of integers, initially 0 in all places // $slot[i]$ is the current slot i is in.

$joinSlot$ array of size N of integers, initially 0 in all places // $joinSlot[i]$ holds the last slot in which i joined (l).

$failSlot$ array of size N of integers, initially 0 in all places // $failSlot[i]$ is the last slot in which i failed.

inc array of size N of integers, initially 0 in all places // $inc[i]$ holds is current incarnation number (n) as determined by the $m\text{-}join(*,l,n,i)_i$ action.

$status$, array of size $N*N$, each element is $\in \{inactive, active\}$ initially $inactive$ in all places. // $status[i][k]$ is the status i sees for k.

$Leave, Failed$ arrays of size N of booleans, initially FALSE in all places. // $Leave[i]$ is TRUE if i is inactive, and the last leave or fail action i performed was a $m\text{-}leave(*,i)_i$. $Failed[i]$ is the same for failures.

Transitions

```
Output $m\text{-}join(l,s,n,i)_i$
    Pre: $n = inc[i] + 1$
         $l > slot[i]$
         for all $k \in I$ {
            $s > slot[k] + 2$
            $status[k][i] = inactive$
         }
         $s > l + 2$
    Eff: $status[i][i] := active$
         $slot[i] := l$
         $joinSlot[i] := s$
         $Failed[i] := FALSE$
         $Leave[i] := FALSE$
         $inc[i] = n$

Output $m\text{-}join(s,n,j)_i$, $j \neq i$
    Pre: $status[i][j] = inactive$
         $n = inc[j]$
         $s \geq joinSlot[j]$
         if($s > joinSlot[i] - 2$) then
            $slot[i] < s$
    Eff: $status[i][j] := active$

Internal $inc\text{-}slot()_i$
    Pre: if($slot[i] = joinSlot[i] - 1$) then
            for all $k$ s.t.($joinSlot[k] \leq joinSlot[i]$
                 && $status[k][k] = active$) {
              $status[i][k] = active$
              $status[k][i] = active$
            }
    Eff: $slot[i] = slot[i] + 1$

Output $m\text{-}leave(n,i)_i$
    Pre: $status[i][i] = active$
         $n = inc[i]$
    Eff: for all $j \in I$
            $status[i][j] = inactive$
         $Leave[i] = TRUE$

Output $m\text{-}leave(n,j)_i$, $j \neq i$
    Pre: $status[i][j] = active$
         $n = inc[j]$
         $Leave[i] = TRUE$
    Eff: $status[i][j] := inactive$

Input $fail_i$
    Eff: if ($status[i][i] = active$) then
              $Failed[i] = TRUE$
              $failSlot[i] = slot[i]$
              for all $j \in I$
                  $status[i][j] = inactive$

Output $m\text{-}fail(s,n,j)_i$, $j \neq i$
    Pre: $status[i][j] = active$
         $Failed[j] = TRUE$
         $n = inc[j]$
         $s < slot[i]$
         $s \geq joinSlot[i] - 2$
         $failSlot[j] - 1 \leq s \leq failSlot[j]$
    Eff: $status[i][j] := inactive$

Input $m\text{-}decide(v,n,j)_i$
    Eff: none
```

Figure 9: The environment specification automaton

B Correctness proof of the DAB service

In this section we give the formal proofs for the two parts of Section 8.

B.1 Achieving the service spec

Lemma B.1 *If some process i that is active at slot s adds j in that slot, then all active processes add j at slot s.*

Proof: If i adds j at slot s then it must be that in some slot prior to s i received a join message for j for slot s. If i received the message from j, then i received it at most Δ time after it was sent, and echoed it at most Θ after that. Since i is active when s starts, and since j chooses an s that starts $((f+3)(\Delta+\Theta)+\Gamma)$ time after it sends the join message, all processes must have received is join message for j before starting to receive message for s, and so all processes will also add j at slot s.
If i did not receive the message directly from j, it means that at least one process k had received js original join message, and echoed it. Using an inductive argument it is easy to see that if i received the join message for j only after $x(\Delta+\Theta)$ time from the time j sent it, then it must be that at least $x-1$ processes have failed (including j). This is so because for each $\Delta+\Theta$ time after j sent its join message, at least one processes had echoed it, otherwise i would not have received it at all. On the other hand, since i did not receive the join message prior to $x(\Delta+\Theta)$ time after it was sent, all processes that sent the join message before $(x-1)(\Delta+\Theta)$ time must have failed. Since the number of processes that can fail is at most f, $x \leq f+1$. So i had received js join message at most $(f+1)(\Delta+\Theta)$ time after j sent its original join message. Since i echoed the join message at most Θ time after it received it, all processes must have received the join message at most $(f+2)(\Delta+\Theta)$ time after it was sent. This happens at least $(\Delta+\Theta)$ time before all processes start slot s, and so when s starts all active processes add j. ∎

Lemma B.2 *If a m-decide(v,n,j) action occurred following a m-fail(s,n,j) action then $0 \leq (v-s) \leq 1$, and this bound is tight.*

Proof: The fact that $0 \geq (v-s)$ comes from the minimum value guarantee in the failure manager spec. As for the other side of the inequality, since the difference between the detection of a processes failure is at most one slot, all actions $m\text{-}fail(s',n,j)_k$ performed by another process k must have $s'+1 \geq s$, and so, from the validity requirement it must be that $v+1 \geq s$, and so $0 \leq (v-s) \leq 1$. ∎

Theorem 3 *Our TO algorithm achieves the following requirements:*

- Integrity
- Ordering
- Joining
- Leaving
- Message Delivery and Latency

This requirements are fully specified in Definition 1 in Section 3.

Proof: We prove this theorem by proving that each of the items above is guaranteed by our algorithm.

- *Integrity:* This is trivially satisfied from our assumption that the network does not duplicate, corrupt or spontaneously generate messages.

- *Ordering:* When no process fails, messages are delivered in each slot according to process indices by all processes. When a process leaves all processes stop delivering its messages after the last message it sent before the "leave" message. As shown above, when a process i joins, all processes add i in the same slot. Thus, all processes order the delivery exactly as in $\mathcal{S}$. Since there are no process failures, using the reliable network guarantees that all processes receive all messages sent, and so each process delivers a contiguous subsequence of $\mathcal{S}$. Since $\mathcal{S}$ is ordered by the slot number, and each time a process joins its slot number increases, it must be that any message m, i delivers while i has incarnation number n, is ordered ahead in $\mathcal{S}$ of any message i delivered while i had incarnation number n' where $n' < n$.
  When a process j fails, then using the DFM all correct processes will have the same value, S_D, as their decision value. Given S_D, we know that all correct processes have received all of js messages for slot S_D (this comes from the minimum value guarantee of the DFM). In addition, no process had delivered js messages for slot $S_D + 1$ (because for every process i, $s_i - 1 \leq S_D$, from lemma 8.2, and according to the algorithm a process does not deliver messages for slot $s - 1$ until it receives all messages for slot s). So all processes act the same when the DFM reaches a decision, delivering js messages up to slot s, and skipping j for slot $s+1$ on. Since messages are still delivered in each process according to their order in $\mathcal{S}$, and all processes act in the same way for js failure, the total ordering is guaranteed.

- *Joining:* According to our algorithm (Section 6.3), if i performs a *join* at time t, and i does not fail or leave, a *join-OK*$_i$ will be performed at time $t + t_1$ where $t_1 = 2\Gamma + 3(\Delta + \Theta) + f(\Delta + \Gamma + 2\Theta) + (f+3)(\Delta + 2\Theta) + \Gamma)$.

- *Leaving:* According to our algorithm (Section 6.4), at most 2Θ time after a *leave* is performed, the corresponding *leave-OK*$_i$ is executed. Thus setting $t_2 = 2\Theta$ guarantees this condition.

- *Message Delivery and Latency:* If m is sent by some process j after time $t + \Theta$, m must have a slot number bigger or equal to $joinSlot_i$ since the clock skew between all processes is at most Θ. Thus, if i and j do not fail m will be delivered by i, because i delivers all messages it receives with slot number greater or equal to $joinSlo_i$ (except for some of the messages from faulty processes), and according to the reliable network guarantees m will be received by i. So setting $t_3 = \Theta$ guarantees that m will be delivered. As for the maximal latency, we prove in Lemma 9.2, that the maximal latency of any message delivered in our algorithm is:

$$AppLatency = 4\Theta + 3\Gamma + \Delta + 3(\Delta + \Theta) + (f-1)(\Delta + \Gamma + 2\Theta)$$

■

B.2 Implementing the DFM environment specification

We show that for each of the actions in the DFM environment spec, and each of the liveness requirements of that spec, the Ordering algorithm achieves its preconditions.

inc-slot() - The fact that all active processes add i in slot $joinSlot_i$ is proved in Lemma 8.1. The requirement that i knows of all active processes that have a join slot less than $joinSlot_i$ is guaranteed by the following observation. For process j that is active, and did not send a join message from the time i woke up, it must be that j has already joined the algorithm (since j sent its join message before i woke up, i will not even send its join message before j actually joins) and so according to the join algorithm (Section 6.3), i must have added j to the list of active processes and so must have performed the *m-join* action for j. For a process j that sent its join message after i woke up, if $joinSlot_j \leq joinSlot_i$ i must have received this join message before $joinSlot_i$ (Lemma 8.1) and so i informed the DFM of j's join.
m-join(s,l,n,i) - The requirement that all processes view i as inactive when i performs this action is guaranteed by the fact that i does not perform this action until i is sure that all consensus algorithms that were performed to decide on its failure have concluded (Section 6.3).
m-join(s,n,j) - The requirement that i has the right incarnation number for j is guaranteed because i uses the number that appears in js join message or in the header of one of js messages. Since i only adds j after j sends a join message or after seeing a message from j we know that $s \geq joinSlot[j]$ (j only sends messages for slots $s' \geq joinSlot[j]$). If $joinSlot[j] > joinSlot[i] - 2$, then the fact that i adds j before i reaches $joinSlot_j$ is guaranteed by Lemma 8.1.
m-leave(n,j),fail - This is trivial.
m-fail(s,n,j) - Since we use a failure detector (Section 6.2) we are guaranteed that this action will only be performed if j actually failed. The fact that $s < slot[i]$ is again guaranteed by our failure detector since it waits more than Θ time before it declares a process as failed. The fact that all processes report a failure slot within 1 of each other is guaranteed by our failure model (Section 2).

The two liveness conditions are also guaranteed by our Ordering algorithm. If $status[k][k]$ was *active* when the j failed, and $joinSlot[k] \leq failSlot[j] + 2$, then according to Section 6.3, k would try to receive js messages for slot $failSlot[j]$. Thus, if j does not fail or leave k will notice js failure and so it will perform the $m\text{-}fail(*, n, j)_k$ action.
If $status[i][k]$ was active when k performs the $m\text{-}leave(n, k)_k$ action, i will either receive ks leave message and thus will perform the $m\text{-}leave(n, k)$ action, or i will not receive this message in which case i would suspect that k failed and so it will perform the $m\text{-}fail(*, n, k)_i$ action, again setting $status[i][k] = inactive$. The only other possibility is that i will fail or leave before it will notice ks message or failure. However, in this case, according to the automaton specification $status[i][k]$ is automatically set to *inactive*.

C Correctness of the DFM

In this section we prove all the lemmas that where presented in Section 7.

C.1 The Dynamic manager algorithm

Lemma C.1 *For all processes i, if the* $p\text{-}fail(s^i, W^i)_{i,n,j}$ *action is performed then:*
1. The $m\text{-}fail(*, n, j)_i$ *action cannot be performed twice by the same process i.*
2. If $k \in P[n, j]$ *and* $k \notin W^i$, *k failed or left after performing the* $p\text{-}fail(*, *)_{k,n,j}$ *action, and before i performed the* $p\text{-}fail(*, *)_{i,n,j}$ *action.*
3. For all $k \in W^i$, *k either performs the* $p\text{-}fail(v', W')_{k,n,j}$ *action or the* $h\text{-}fail(v')_{k,n,j}$ *action or* $status[i][k]$ *becomes inactive.*

Proof: 1. As can be seen (from the spec) performing the $m\text{-}fail(*, n, j)_i$ action results in setting $status[i][j] = inactive$. A precondition of the $m\text{-}fail(*, n, j)_i$ action is that $status[i][j] = active$. Thus, $status[i][j]$ should change to $active$ between two consecutive $m\text{-}fail(*, n, j)_i$ actions, and this could only be done in the $m\text{-}join(*, *, j)_i$ action. When i performs the $m\text{-}fail(*, n, j)_i$ action we have $status[j][j] = inactive$ with $inc[j] = n$ since i can only perform this action when $Failed[j] = TRUE$ (meaning that the last action performed for j was the $fail_j$ action), and so in order to set $status[i][j]$ to active again, j must perform the $join(*, *, j)_j$ action after i performs the first $m\text{-}fail(*, n, j)_i$ action. However, when j performs the $join(*, *, j)_j$ action, j must set $inc[j] = n+1$ according to the spec, and since a precondition of the $m\text{-}fail(*, n', j)_i$ action is that $n' = inc[j]$, i cannot perform the $m\text{-}fail(*, n, j)_i$ action again (since $inc[j] > n$ from now on).
2. Assume for contradiction that k did not fail after performing the $p\text{-}fail(*, *)_{k,n,j}$ action.
If $status[i][k]$ was $active$ when i performed the $p\text{-}fail(s^i, W^i)_{i,n,j}$ action, then since $k \notin W^i$ it means that $joinSlot[k] \geq s^i + 2$, and since $s^i \geq failSlot[j] - 1$ we can conclude that $joinSlot[k] \geq failSlot[j] + 1$. Since k did not fail or leave after performing the $p\text{-}fail(s^k, *)_{k,n,j}$ action, this must be less or equal to the $joinSlot[k]$ that k had when performing the $m\text{-}fail(s^k, n, j)_k$ action (since according to part 1 of this lemma the $m\text{-}fail(*, n, j)_k$ action can only be performed once). We now arrive at a contradiction since for this $joinSlot[k]$ value we have $joinSlot[k] \geq failSlot[j] + 1 > s^k$, but according to the $m\text{-}fail$ action, in order to perform the $p\text{-}fail(*, *)_{k,n,j}$ action k must have $s^k \leq joinSlot[k]$. So in this case it must be that k failed or left after performing the $p\text{-}fail(*, *)_{k,n,j}$ action.
Assume $status[i][k]$ was $inactive$ when i performed the $p\text{-}fail(s^i, W^i)_{i,n,j}$ action, and k did fail or left after performing the $p\text{-}fail(s^k, *)_{k,n,j}$ action. Then it must be that when k performed the $p\text{-}fail(s^k, *)_{k,n,j}$ action it had $slot[k] > s^k \geq joinSlot[k]$. So k must have performed the $inc\text{-}slot()_k$ action that sets $slot[k] = joinSlot[k]$. A precondition of such an action is that for all active processes m with $joinSlot[m] \leq joinSlot[k]$, $status[m][k] = active$. It is easy to see that i must have been one of these processes when this action took place, otherwise i could not have had $status[i][k] = inactive$ since k did not fail or leave after this action. So it must be that $status[i][k]$ was set to active after i performed the $p\text{-}fail(s^i, W^i)_{i,n,j}$ action. However, since $joinSlot[k] \leq failSlot[j]$ we know that when i performed the $p\text{-}fail(s^i, W^i)_{i,n,j}$ action i had $slot[i] \geq s^i + 1 \geq failSlot[j] \geq joinSlot[k]$, and this contradicts the fact that i joined k after performing this action since the precondition for joining a process (in the $m\text{-}join(s, n, k)_i$ action) is that $slot[i] < s$ and since $s \leq joinSlot[j]$ this implies that $slot[i] < joinSlot[j]$ when i performed the $m\text{-}join$ action. Thus it must be that k failed or left after performing the $p\text{-}fail$ action, and since $status[i][k]$ was $inactive$ when i performed the $p\text{-}fail$ action, this must have been before i performed the $p\text{-}fail$ action.
3. If $k \in W^i$ then $s^i \geq joinSlot[k] - 1$, and so $failSlot[j] \geq s^i \geq joinSlot[k] - 1$. Since $s^k + 1 \geq failedSlot[j]$, we have $s^k + 1 \geq joinSlot[k] - 1 \rightarrow s^k \geq joinSlot[k] - 2$. If $status[k][k]$ was $active$ when $fail_j$ occurred then according to the liveness guarantee of the environment specification k will either perform the $m\text{-}fail(s^k, n, j)_k$ action or will fail or leave (which will result in setting $status[i][k]$ to inactive according to the liveness guarantees). If k performs the $m\text{-}fail(s^k, n, j)_k$ action k will either perform the $h\text{-}fail(*)_{k,n,j}$ or the $p\text{-}fail(*, *)_{k,n,j}$ actions.
If $status[k][k]$ was inactive when the $fail_j$ action occurred, then it must be that k did not rejoin the algorithm before i performed the $m\text{-}fail(*, n, j)_i$ action. This is so because in order to rejoin, k must set $joinSlot[k] > failSlot[j] + 2$. However, since $k \in W^i$ it must be that $joinSlot[k] \leq s^i + 1 \leq failSlot[j] + 1$ and so k could not have rejoined the algorithm after j has failed. Note that k cannot rejoin the algorithm before i sets $status[i][k]$ to inactive according to the $m\text{-}join(*, *, k)_k$ precondition. Thus, $status[k][k]$ was $inactive$ when i performed the $m\text{-}fail(*, *, j)_i$ action, and so i will set $status[i][k]$ to $inactive$ at some point according to the liveness guarantees. ■

C.2 The First Round algorithm

Lemma C.2 *If $i \in P[n,j]$ then the combined service guarantees the following with respect to $F^{n,j}$:*
1. Termination.
2. $C[n,j] \subseteq G^i$, and if $k \in G^i$ and $k \notin C[n,j]$ then k failed or left after performing the $p\text{-}fail(*,*)_{k,n,j}$ action.
3. For all processes $k \in C[n,j]$, $l^k \leq u^i$.

Proof: 1. In order to terminate $F^{n,j}$ i has to perform the $inCons(*,*)_{i,n,j}$ action. In order to perform this action i needs to receive messages from all processes in W^i or wait until they failed or left. However, according to lemma 7.1 part 3 all such processes will either perform the $h\text{-}fail(*)_{*,n,j}$ or $p\text{-}fail(*,*)_{*,n,j}$ actions, resulting in sending a message to i or will fail or leave. Thus, all processes that i waits for will either send a message or fail or leave and termination is guaranteed.
2. If $k \in C[n,j]$ then $k \in P[n,j]$ according to the construction of $C[n,j]$ and in addition, k did not fail before finishing the first round algorithm. If $k \notin W^i$ we know that k had failed or left before i performed the $p\text{-}fail(*,*)_{i,n,j}$ action (lemma 7.1 part 2), and thus $k \notin C[n,j]$ since in order to finish the first round k needs to receive a message from i (since $i \in P[n,j]$ and did not fail or left so $i \in W^k$). But since k failed or left before i sent any message, k did not finish the first round algorithm.
So if $k \in C[n,j]$ it must be that $k \in W^i$ and since $k \in C[n,j]$ we know that ks message for the first round have reached i, and so according to the *receive* action effect, $k \in G^i$ (since k must have sent a "part" message since $k \in C[n,j]$).
If $k \in G^i$ and $k \notin C[n,j]$ then it must be that k started $F^{n,j}$, and that $k \in P[n,j]$ (otherwise i would not add it to G^i, since k would send a "help" message). Since $k \notin C[n,j]$ we know that k must have failed or left (since according to part 1 of this lemma, if this is not the case k would have finished $F^{n,j}$).
3. If $i \in C[n,j]$ it must be that for all processes k such that $k \in C[n,j]$, $i \in G^k$ according to part 2 of this lemma. But this means that k received is message for $F^{n,j}$, and thus according to the $receive(v)_{k,i,n,j}$ action effect, $l^k \leq u^i$. ■

C.3 Time analysis

Lemma C.3 *If f processes fail during the execution of the First round and Consensus algorithms, then the algorithm takes at most:*

$$\Gamma + \Theta + 3(\Delta + \Theta) + f(\Delta + \Gamma + 2\Theta)$$

Proof: If two processes i and j participate in the First round algorithm, then the difference between the time i started the algorithm and the time j started the algorithm is at at most $\Theta + \Gamma$. This is so because both start the algorithm when the failure of k is detected. If i detects the failure of k in slot s, j would detect it at most in slot $s+1$, and since the clock skew is at most Γ the total difference is at most $\Theta + \Gamma$.
Assume j is the last process to start the First round algorithm. Then the First round ends at all processes at most $2\Theta + \Delta + \Gamma$ time after it started at j. This is so because if no process fails, then all First round messages are sent at most Θ after j started the algorithm, and arrive at most Δ time after that at all processes, finishing the first round. If the First round message from process p would not arrive at process i $\Theta + \Delta$ time after j detected ks failure, it means that p has failed, and this failure will be detected by all processes at most $\Theta + \Delta + \Gamma$ time after p have failed, which is at

most $2\Theta + \Delta + \Gamma$ time after j started the First round algorithm. Thus, $2\Theta + \Delta + \Gamma$ time after the last process detected ks failure all processes would be able to finish the First round (either receiving all messages sent for this round, or detecting processes from which no messages where received as failed).

The same analysis shows that this is the case for all rounds in the Consensus algorithm too. If a process fails during a round then it will take at most $2\Theta + \Delta + \Gamma$ time to conclude the round by all other processes. Thus, if f processes fail during the execution of the algorithm we need at most three more rounds in addition to the f rounds in order to reach decision (the first round and at most two rounds at the end). In these three rounds no process fails, so they take at most $\Delta + \Theta$ time each. Thus the algorithm takes at most: $\Theta + \Gamma + 3(\Theta + \Delta) + f(\Delta + \Gamma + 2\Theta)$. ■

Lemma C.4 *If $m\text{-}fail(s,n,j)_i$ is performed at time t_0, and starting at time $t > t_0$ no $m\text{-}fail$ action was performed by any process for $3(\Delta + 2\Theta + \Gamma)$ time, m-decide$(v,n,j)_i$ will be performed by $t + 3(\Delta + 2\Theta + \Gamma)$.*

Proof: According to the previous lemma, the First round algorithm takes at most $\Delta+2\Theta+\Gamma$ time to finish by all processes, and so does any of the consensus rounds. Thus, since no process performs a $m\text{-}fail$ action from time t for $3(\Delta + 2\Theta + \Gamma)$ time, all processes will be able to finish at least two consecutive consensus rounds without reporting any failure. However, if all processes see the same set of processes for two consecutive rounds, the consensus algorithm ends at all processes. So by $t + 3(\Delta + 2\Theta + \Gamma)$, the consensus algorithm for j will end, and i will perform the *m-decide*$(v,n,j)_i$ action. ■