Early-Delivery Dynamic Atomic Broadcast (Extended Abstract) *

Ziv Bar-Joseph[1], Idit Keidar[2], and Nancy Lynch[1]

[1] MIT Laboratory for Computer Science
zivbj@mit.edu, lynch@theory.lcs.mit.edu
[2] Technion Department of Electrical Engineering
idish@ee.technion.ac.il

Abstract. We consider a problem of atomic broadcast in a dynamic setting where processes may join, leave voluntarily, or fail (by stopping) during the course of computation. We provide a formal definition of the *Dynamic Atomic Broadcast* problem and present and analyze a new algorithm for its solution in a variant of a synchronous model, where processes have approximately synchronized clocks.

Our algorithm exhibits constant message delivery latency in the absence of failures, even during periods when participants join or leave. To the best of our knowledge, this is the first algorithm for totally ordered multicast in a dynamic setting to achieve constant latency bounds in the presence of joins and leaves. When failures occur, the latency bound is linear in the number of actual failures. Our algorithm uses a solution to a variation on the standard distributed consensus problem, in which participants do not know a priori who the other participants are. We define the new problem, which we call *Consensus with Uncertain Participants*, and give an early-deciding algorithm to solve it.

1 Introduction

We consider a problem of atomic broadcast in a *dynamic setting* where an unbounded number of participants may join, leave voluntarily, or fail (by stopping) during the course of computation. We formally define the *Dynamic Atomic Broadcast (DAB)* problem, which is an extension of the Atomic Broadcast problem [13] to a setting with infinitely many processes, any finite subset of which can participate at a given time. Just as Atomic Broadcast is a basic building block for state machine replication in a static setting, DAB can serve as a building block for state machine replication among a dynamic set of processes.

We present and analyze a new algorithm, which we call *Atom*, for solving the DAB problem in a variant of a synchronous crash failure model. Specifically, we assume that the processes solving DAB have access to approximately-synchronized local clocks and to a lower-level network that guarantees timely

* Full version appears as MIT Technical Report MIT-LCS-TR-840 [3]. This work supported by MURI AFOSR Award F49620-00-1-0327, AFOSR Contract F49620-00-1-0097, NTT Contract number MIT9904-12, and NSF Contract CCR-0121277.

message delivery among currently active processes. The challenge is to guarantee consistency among the sequences of messages delivered to different participants, while still achieving timely delivery, even in the presence of joins and leaves.

Atom exhibits *constant* message delivery latency in the absence of failures, *even during periods when participants join or leave*; this is in contrast to previous algorithms solving similar problems in the context of view-oriented group communication, e.g., [1, 9]. When failures occur, Atom's latency bound is linear in the number of failures that actually occur; it does not depend on the number of potential failures, nor on the number of joins and leaves that occur.

A key difficulty for an algorithm solving DAB is that when a process fails, the network does not guarantee that the surviving processes all receive the same messages from the failed process. But the strong consistency requirements of DAB dictate that processes agree on which messages they deliver to their clients. The processes carry out a protocol to coordinate message delivery, which works roughly as follows: Each Atom process divides time into *slots*, using its local clock, and assigns each message sent by its client to a slot. Each process delivers messages to its client in order of slots, and within each slot, in order of sender identifiers. Each process determines the *membership* of each slot, and delivers messages only from senders that it considers to be members of the slot. To ensure consistency, the processes must agree on the membership of each slot.

Processes joining (or voluntarily leaving) the service coordinate their own join (or leave) by selecting a join-slot (or leave-slot) and informing the other processes of this choice, without delaying the normal delivery of messages. When a process fails, Atom uses a novel *distributed consensus service* to agree upon the slot in which it fails. The consensus service required by Atom differs from the standard stopping-failure consensus services studied in the distributed algorithms literature (see, e.g., [16]) in that the processes implementing the consensus service do not know a priori who the other participants are. Atom tracks process joins and leaves, and uses this information to approximate the active set of processes that should participate in consensus. However, different processes running Atom may have somewhat different perceptions of the active set, e.g., when a participant joins or leaves Atom at roughly the time consensus is initiated.

In order to address such uncertainties, we define a new consensus service, *consensus with uncertain participants (CUP)*. When a process i initiates CUP, it submits to CUP a finite set W_i estimating the current world, in addition to i's proposed initial consensus value v_i. The worlds suggested by different participants do not have to be identical, but some restrictions are imposed on their consistency. Consider, e.g., the case that process k joins Atom at roughly the time CUP is initiated. One initiator, i, may think that k has joined in time to participate and include k in W_i, while another, j, may exclude k from W_j. Process k cannot participate in the CUP algorithm in the usual way, because j would not take its value into account. On the other hand, if k does not participate at all, i could block, waiting forever for a message from k. We address such situations by allowing k to explicitly *abstain* from an instance of CUP, i.e., to participate without providing an input. A service that uses CUP must ensure the following *world consistency* assumption: that for every i, (1) W_i includes all the processes that ever initiate this instance of CUP (unless they fail or leave prior to i's initiation); and (2) if $j \in W_i$, (and neither i nor j fail or leave), then j participates in CUP either by initiating or by abstaining. Thus, W_i sets can differ only in the inclusion of processes that abstain, leave, or fail. Note that once an instance of CUP has been started, no new processes (that are not included in W_i) can join the running instance. Nevertheless, CUP provides a good abstraction for solving DAB, because Atom can invoke multiple instances of CUP with different sets of participants.

We give an early-deciding algorithm to solve CUP in a fail-stop model, that is, in an asynchronous crash failure model with perfect failure detectors. The failure detector is external to CUP; it is implemented by Atom. CUP uses a strategy similar to previous early-deciding consensus algorithms [10], but it also tolerates uncertainty about the set of participants, and moreover, it allows processes to leave voluntarily without incurring additional delays. The time required to reach consensus is linear in the number of failures that actually occur during an execution, and does not depend on the number of potential failures.

We also analyze the message-delivery latency of Atom under different failure assumptions. We show a constant latency bound for periods when no failures occur, even if joins and leaves occur. When failures occur, the latency is proportional to the number of actual failures. This is inevitable: atomic broadcast requires a number of rounds that is linear in the number of failures.

We envision a service using Atom, or a variation of it, deployed in a large LAN, where latency is predictable and message loss is bounded. In such settings, a network with the properties we assume can be implemented using forward error correction (see [2]), or retransmissions (see [20]). The algorithm can be extended for use in environments with looser time guarantees, e.g., networks with differentiated services; we outline ideas for such an extension in Section 7.4.

In summary, this paper makes the following main contributions: (1) the definitions of two new services for dynamic networks: DAB and CUP; (2) an early-delivery DAB algorithm, Atom, which exhibits constant latency in the absence failures; (3) a new early-deciding algorithm for CUP; and (4) the analysis of Atom's message-delivery latency under various failure assumptions.

The rest of this paper is organized as follows: Section 2 discusses related work. In Section 3, we specify the DAB service. In Section 4 we specify CUP and in Section 5, we present the CUP algorithm. Section 6 specifies the environment assumptions for Atom, and Section 7 presents the Atom algorithm. Section 8 concludes the paper. In the full paper [3], we present the algorithms in pseudocode and prove their correctness.

2 Related Work

Atomic broadcast in a dynamic universe, where processes join and leave, was first considered in the context of view-oriented group communication systems (GCSs) [6], pioneered by Isis [4]. Our service resembles those provided by GCSs; although we do not export membership to the application, it is computed and would be easy to export.

GCSs, including those designed for synchronous systems and real-time applications (e.g., Cristian's [9], xAMp [18], and RTCAST [1]), generally run a group membership protocol every time a process joins or leaves, and therefore delay message delivery to all processes when joins or leaves occur. Cristian's service exhibits constant latency only in periods in which no joins or failures occur; latency during periods with multiple joins is not analyzed. xAMp is a GCS supporting a variety of communication primitives for real-time applications. The presentation of xAMp in [18] assumes that a membership service is given. The delays due to failures and joins are incurred in the membership part, which is not described or analyzed. The latency bound achieved by RTCAST is *linear* in the number of processes, even when no process fails, due to the use of a logical ring. Moreover, RTCAST makes stronger assumptions about its underlying network than we do – it uses an underlying reliable broadcast service that guarantees that correct processes deliver the same messages from faulty ones.

Light-weight group membership services [11] avoid running the full-scale membership for join and leaves by using atomic broadcast to disseminate join and leave messages in a consistent manner. Unlike our CUP service, the atomic broadcast services used by such systems do not tolerate uncertainty about the participants. Therefore, a race condition between a join and a concurrent failure can cause such light-weight group services (e.g., [11]) to violate consistency. Those light-weight group services that do preserve consist membership semantics (e.g., [19]), do incur extra delivery latencies whenever joins and leaves occur.

Other work on group membership in synchronous and real-time systems, e.g., [15, 14] has focused on membership maintenance in a static, fairly small, group of processes, where processes are subject to failures but no new processes can join the system. Likewise, work analyzing time bounds of synchronous atomic broadcast, e.g., [12, 8, 7], considered a static universe, where processes could fail but not join. Thus, this work did not consider the DAB problem.

In a previous paper [2], we considered a simpler problem of dynamic totally ordered broadcast without all or nothing semantics. For this problem, the linear lower bound does not apply, and we exhibited an algorithm that solves the problem in constant time even in the presence of failures.

Recent work [17, 5] considers different services, including (one shot) consensus, for infinitely many processes in asynchronous shared memory models. Chockler and Malkhi [5] present a consensus algorithm for infinitely many processes using a *static* set of active disks, a minority of which can fail. This differs from the model considered here, as in our model all system components may be ephemeral. Merritt and Taubenfeld [17] study consensus under different concurrency models, and show that if there is no bound on the number of participants, in an asynchronous shared memory model, solving consensus requires infinitely many bits. The algorithms they give tolerate only initial failures. To the best of our knowledge, atomic broadcast has not been considered in a similar context.

3 Dynamic Atomic Broadcast Service Specification

The universe is an infinite ordered set of endpoints, I; M is a message alphabet. Figure 1 presents DAB's signature. We assume that an application using DAB satisfies some basic well-formedness assumptions, (cf. [3]), e.g., that a process does not join / leave more than once, does not multicast messages before a join or after a leave, and does not send the same message more than once. We do not consider rejoining; instead, we consider the same client joining at new endpoints.

Input: $\texttt{join}_\texttt{i}$, $\texttt{leave}_\texttt{i}$, $\texttt{fail}_\texttt{i}$, i∈I	**Output:** $\texttt{join\_OK}_\texttt{i}$, $\texttt{leave\_OK}_\texttt{i}$, i∈I
$\texttt{mcast}_\texttt{i}$(m), m∈M, i∈I	$\texttt{rcv}_\texttt{i}$(m), m∈M, i∈I

Fig. 1. The signature of the DAB service.

We require that there be a total ordering $\mathcal{S}$ on all the messages received by any of the endpoints, such that for all $i \in I$, the following properties are satisfied.

- *Multicast order:* If $\texttt{mcast}_i\texttt{(m)}$ occurs before $\texttt{mcast}_i\texttt{(m')}$, then m precedes m' in $\mathcal{S}$.
- *Receive order:* If $\texttt{rcv}_i\texttt{(m)}$ occurs before $\texttt{rcv}_i\texttt{(m')}$ then m precedes m' in $\mathcal{S}$.
- *Multicast gap-freedom:* If $\texttt{mcast}_i\texttt{(m)}$, $\texttt{mcast}_i\texttt{(m')}$, and $\texttt{mcast}_i\texttt{(m'')}$ occur, in that order, and $\mathcal{S}$ contains m and m'', then $\mathcal{S}$ also contains m'.
- *Receive gap-freedom:* If $\mathcal{S}$ contains m, m', and m'', in that order, and $\texttt{rcv}_i\texttt{(m)}$ and $\texttt{rcv}_i\texttt{(m'')}$ occur, then $\texttt{rcv}_i\texttt{(m')}$ also occurs.
- *Multicast liveness:* If $\texttt{mcast}_i\texttt{(m)}$ occurs and no $\texttt{fail}_i$ occurs, then $m \in \mathcal{S}$.
- *Receive liveness:* If $m \in \mathcal{S}$, m is sent by i and i does not leave or fail, then $\texttt{rcv}_i\texttt{(m)}$ occurs, and for every m' that follows m in $\mathcal{S}$, $\texttt{rcv}_i\texttt{(m')}$ also occurs.

In addition to the above, DAB is required to satisfy basic integrity properties, e.g., that $\texttt{join\_OK}_i$ ($\texttt{leave\_OK}_i$) must be preceded by a $\texttt{join}_i$ ($\texttt{leave}_i$); that every $\texttt{join}_i$ ($\texttt{leave}_i$) is followed by a $\texttt{join\_OK}_i$ ($\texttt{leave\_OK}_i$); and that messages are not received more than once, and are not received unless they are multicast. The formal definitions of these appear in [3].

4 Consensus with Uncertain Participants – Specification

In order to solve DAB, we use the CUP service. CUP is an adaptation of the problem of fail-stop uniform consensus to a setting in which the set of participants is not known precisely ahead of time, and in which participants can leave the algorithm voluntarily after initiating it. Moreover, participants are not assumed to initiate at the same time. CUP assumes an underlying reliable network, and a perfect failure detector. The signature of the CUP service is presented in Figure 2; I is a universe as above; V is a totally ordered set of possible consensus *values*; and M_{CUP} is a message alphabet.

A process i may participate in CUP in two ways: it may *initiate* CUP using $\texttt{init}_i\texttt{(v,W)}$ and provide an initial value and an initial world, or it may *abstain* (using $\texttt{abstain}_i$). Informally speaking, a participant abstains when it does not need to participate in CUP, but because of uncertainty about CUP participants, some other participant may expect it to participate. CUP reports the consensus decision value to process i using the $\texttt{decide}_i\texttt{(v)}$ action. The environment provides a *leave detector* and a *failure detector*: $\texttt{leave\_detect}_i\texttt{(j)}$ notifies i that

Input: $\texttt{init}_\texttt{i}\texttt{(v,W)}$, $\texttt{v} \in \texttt{V}$, $\texttt{W} \subseteq \texttt{I}$, $\texttt{W finite}$, $\texttt{i} \in \texttt{I}$ // *i initiates*
$\texttt{abstain}_\texttt{i}$, $\texttt{i} \in \texttt{I}$ // *i abstains*
$\texttt{net\_rcv}_\texttt{i}\texttt{(m)}$, $\texttt{m} \in \texttt{M}_\texttt{CUP}$, $\texttt{i} \in \texttt{I}$ // *i receives message m*
$\texttt{leave}_\texttt{i}$, $\texttt{i} \in \texttt{I}$ // *i leaves*
$\texttt{leave\_detect}_\texttt{i}\texttt{(j)}$, $\texttt{j}, \texttt{i} \in \texttt{I}$ // *i detects that j has left*
$\texttt{fail}_\texttt{i}$, $\texttt{i} \in \texttt{I}$ // *i fails*
$\texttt{fail\_detect}_\texttt{i}\texttt{(j)}$, $\texttt{j}, \texttt{i} \in \texttt{I}$ // *i detects that j failed*

Output: $\texttt{decide}_\texttt{i}\texttt{(v)}$, $\texttt{v} \in \texttt{V}$, $\texttt{i} \in \texttt{I}$ // *i decides on value v*
$\texttt{net\_mcast}_\texttt{i}\texttt{(m)}$, $\texttt{m} \in \texttt{M}_\texttt{CUP}$, $\texttt{i} \in \texttt{I}$ // *i multicasts m*

Fig. 2. The signature of CUP.

j has left the algorithm voluntarily, and $\texttt{fail\_detect}_i\texttt{(j)}$ notifies i that j has failed. In Section 4.1 we specify assumptions about CUP's environment; assuming these hold, CUP satisfies the following properties:

- *Uniform Agreement:* For any $i, j \in I$, if $\texttt{decide}_i\texttt{(v)}$ and $\texttt{decide}_j\texttt{(v')}$ both occur then $v = v'$.
- *Validity:* For any $i \in I$, if $\texttt{decide}_i\texttt{(v)}$ occurs then (1) for some j, $\texttt{init}_j\texttt{(v,*)}$ occurs; and (2) if $\texttt{init}_i\texttt{(v',*)}$ occurs then $v \leq v'$.
- *Termination:* If $\texttt{init}_i$ occurs, then a $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs.

The validity condition (2) is not a standard property for consensus but is needed for our use in Atom. Another difference from standard consensus is that participants that abstain need not be informed of the decision value.

In addition to these properties, CUP satisfies a well-formedness condition saying that only participants that have initiated can decide, and each participant decides at most once (cf. [3]).

4.1 CUP Environment Assumptions

CUP requires some simple well-formedness conditions saying that each participant begins participating (by initiating or abstaining) at most once, leaves at most once, and fails at most once. CUP also makes standard integrity assumptions about the underlying network, namely that every message that is received was previously sent, and no message is received at the same location more than once. Moreover, the order of message receipt between particular senders and receivers is FIFO. We now specify the more interesting environment assumptions.

The following assumptions are related to the worlds `W` suggested by participants in their `init` events. The first is a safety assumption saying that each `W` set submitted by an initiating participant i must include all participants that ever initiate CUP and that do not leave or fail prior to the $\texttt{init}_i$ event. This implies that every participant must be included in its own estimated world. The next is a liveness assumption saying that, if any process i expects another process j to participate, then j will actually do so, unless either i or j leaves or fails.

– *World consistency:* If $\texttt{init}_i\texttt{(*, W)}$ and $\texttt{init}_j\texttt{(*,*)}$ events occur, then either $j \in W$, or a $\texttt{leave}_j$ or $\texttt{fail}_j$ event occurs before the $\texttt{init}_i\texttt{(*, W)}$ event.
– *Init occurrence:* If an $\texttt{init}_i\texttt{(*,W)}$ event occurs and $j \in \texttt{W}$, then an $\texttt{init}_j$, $\texttt{abstain}_j$, $\texttt{leave}_i$, $\texttt{fail}_i$, $\texttt{leave}_j$, or $\texttt{fail}_j$ occurs.

The next assumptions are related to leaves, leave detection, and failure detection. The second property says that leaves are handled gracefully, in the sense that the occurrence of a $\texttt{leave\_detect}_i\texttt{(j)}$ implies that i has already received any messages sent by j prior to leaving. Thus, a $\texttt{leave\_detect}_i\texttt{(j)}$ is an indication that i has not lost any messages from j. Note that we do not have a failure assumption analogous to the lossless leave property; thus, failures are different from leaves in that we allow the possibility that some messages from failed processes may be lost.

– *Accurate leave detector:* For any $i, j \in I$, at most one $\texttt{leave\_detect}_i\texttt{(j)}$ event occurs, and if it occurs, then it is preceded by a $\texttt{leave}_j$.
– *Lossless leave:* Assume $\texttt{net\_mcast}_j\texttt{(m)}$ occurs and is followed by a $\texttt{leave}_j$. Then if a $\texttt{leave\_detect}_i\texttt{(j)}$ occurs, it is preceded by $\texttt{net\_rcv}_i\texttt{(m)}$.
– *Accurate failure detector:* For any $i, j \in I$, at most one $\texttt{fail\_detect}_i\texttt{(j)}$ event occurs, and if it occurs, then it is preceded by a $\texttt{fail}_j$.
– *Complete leave and failure detector:* If $\texttt{init}_i\texttt{(*,W)}$ occurs, $j \in W$, and $\texttt{leave}_j$ or $\texttt{fail}_j$ occurs, then $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs.

The next liveness assumption describes reliability of message delivery. It says that any message that is multicast by a non-failing participant that belongs to any of the W sets submitted to CUP, is received by all the non-leaving, non-failing members of all those W sets.

– *Reliable delivery:* Define $\texttt{U} = \cup_{k \in I}\{\ \texttt{W} \mid \texttt{init}_k\texttt{(*, W)}\ \ \texttt{occurs}\}$. If $i, j \in \texttt{U}$ and $\texttt{net\_mcast}_i\texttt{(m)}$ occurs after an $\texttt{init}_i$ or $\texttt{abstain}_i$ event, then a $\texttt{net\_rcv}_j\texttt{(m)}$, $\texttt{leave}_j$, $\texttt{fail}_i$, or $\texttt{fail}_j$ occurs.

5 The CUP Algorithm

The algorithm proceeds in asynchronous rounds, $1, 2, \ldots$. In each round, a process sends its current estimates of the value and the world to the other processes. Each process maintains two-dimensional arrays, `value` and `world`, with the value and world information it receives from all processes in all rounds. It records, in a variable `out[r]`, the processes that it knows will not participate in round `r` because they have left, abstained, or decided, and in a variable `failed[r]`, the processes that it learned have failed in this round.

When $\texttt{init}_i\texttt{(v,W)}$ occurs, process i triggers `net_mcast(i,1,v,W)` to send its initial value `v` and estimated world `W` to all processes, including itself. Note that two separate processes, A and B can initiate CUP with different (overlapping) subsets of processes in their `W` parameter. For example, it could be that A has `W = {A,B,C}` while B has `W = {A,B,D}`. However, in this case we are guaranteed from the *World consistency* and *Init occurrence* assumptions that C either fails prior to $\texttt{init}_B\texttt{(v,W)}$, or abstains, and the same holds for A and D. The world W is determined to be the set of processes that i thinks are still active, i.e., the processes in i's previous world that i does not know to be out or to have failed in round r. Process i may perform this multicast only if its round is r-1, it has received round r-1 messages from all the processes in W, and it is not currently able to decide. The value v that is sent is the minimum value that i has recorded for round r-1 from a process in W. When a $\texttt{net\_rcv}_i\texttt{(j,r,v,W)}$ occurs, process i puts v and W into the appropriate places in the value and world arrays.

Process i can decide at a round r when it has received messages from all processes in its world[r,i] except those that are out at round r, such that all of these messages contain the same value and contain worlds that are subsets of world[r,i]. The subset requirement ensures that processes in world[r,i] will not consider values from processes outside of world[r,i] in determining their values for future rounds. When process i decides, it multicasts an OUT message and stops participating in the algorithm.

When $\texttt{abstain}_i$ occurs, process i also sends an OUT message, so that other processes will know not to wait for further messages from it, and stops participating in the algorithm. When a $\texttt{net\_rcv}_i\texttt{(j,OUT)}$ occurs, process i records that j is out of the algorithm starting from the first round for which i has not yet received a regular message from j.

When $\texttt{leave}_i$ occurs, i just stops participating in the algorithm. When a $\texttt{leave\_detect}_i\texttt{(j)}$ event occurs, i records that j is out from the first round after the round of the last message received from j. The lossless leave assumption ensures that i has already received all the messages j sent. Process i knows that process j has failed if $\texttt{fail\_detect}_i\texttt{(j)}$ occurs.

In [3] we prove that when CUP's environment satisfies CUP's safety assumptions, CUP satisfies its safety guarantees, and when CUP's environment satisfies CUP's safety and liveness assumptions, CUP satisfies its liveness guarantees.

5.1 Analysis

The algorithm is early-deciding in the sense that the number of rounds it executes is proportional to the number of actual failures that occur, and does not depend on the number of participants or on the number of processes that leave. In [3], we prove the following theorem, which says that the algorithm always terminates after it can run two rounds without failures.

Theorem 1. *Suppose that* r > 0. *Suppose that there is a point t in the execution such that every process is in* round $\leq$ r *at point t, and no* fail *events happen from t onward. Then every process always has* round $\leq$ r +2.

The proof is based on the observation that once failures stop, the values and worlds that processes send in their round messages stop changing; the value converges to the minimum value that a live process has, and the world converges to the set of live processes. After a round in which all processes in W send the same value and the world W, all the live processes can decide.

We next analyze CUP's running time assuming the following bounds on message latency, failure and leave detection times, and the difference between different processes' initiation times. Note: time bounds are not assumed as a condition for correctness; they are only assumed for the sake of the analysis.

1. δ_1 is an upper bound on *message latency* and on *failure and leave detection time*. Moreover, if a message is lost due to failure, then the failure is detected at most δ_1 after the lost message was sent. More precisely,
   (a) if `net_rcv(m)` occurs, the time since the corresponding `net_mcast(m)` is at most δ_1.
   (b) Assume $\texttt{init}_i\texttt{(*,W)}$ occurs with $j \in W$ and $\texttt{fail}_j$ or $\texttt{leave}_j$ occurs at time t. Then $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs by time $t + \delta_1$.
   (c) Let $U = \cup_{k \in I}\{W| \ \texttt{init}_k(*, W) \text{ occurs}\}$. Assume $i, j \in U$ and $\texttt{net\_mcast}_j\texttt{(m)}$ occurs at time t but no $\texttt{net\_rcv}_i\texttt{(m)}$ occurs. Then either $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs by time $t + \delta_1$.
2. δ_2 bounds the difference between the initiation time of different processes. More precisely:
   Assume a process initiates at time t and does not fail by time $t+\delta_1$, and that $\texttt{init}_i\texttt{(*, W)}$ occurs. Then, every process $j \in W$ initiates, abstains, leaves, or fails by time $t + \delta_2$.

Given these bounds, in [3], we prove the following theorem:

Theorem 2. *Suppose that there is a point t in the execution such that no* `fail` *events happen from t onward. Suppose also that some process initiates CUP by time t. Then every process that decides, decides by time* $t + 3\delta_1 + \delta_2$.

6 Environment and Model Assumptions for Atom

We model time using a continuous global variable `now`, which holds the real time. This is a real variable, initially 0. Each endpoint i is equipped with a local clock, $\texttt{clock}_i$. We assume a bound of Γ on clock skew, where Γ is a positive real number. Specifically, for each endpoint i, we assume that in any state of the system that is reachable $|clock_i - now| \leq \Gamma/2$. That is, the difference between each local clock and the real time is at most $\Gamma/2$. It follows that the clock skew between any pair of processes is Γ, formally: in any reachable state, and for any two endpoints i and j, $|clock_i - clock_j| \leq \Gamma$. We assume that local processing time is 0 and that actions are scheduled immediately when they are enabled.

We assume that we are given a low-level reliable network service Net, with a message alphabet, M'. The Net signature is defined in Figure 3. The actions are the same as those of DAB, except that they are prefixed with `net_`.

Input: $\texttt{net\_join}_\texttt{i}$, $\texttt{net\_leave}_\texttt{i}$, i∈I $\texttt{fail}_\texttt{i}$, i∈I $\texttt{net\_mcast}_\texttt{i}\texttt{(m)}$, m∈M′, i∈I,	**Output:** $\texttt{net\_join\_OK}_\texttt{i}$, i∈I, $\texttt{net\_leave\_OK}_\texttt{i}$, i∈I, $\texttt{net\_rcv}_\texttt{i}\texttt{(m)}$, m∈M′, i∈I

Fig. 3. The signature of the Net service.

Like DAB, Net assumes that its application satisfies the some basic integrity conditions. Assuming these, Net satisfies a number of safety and liveness properties. First, Net satisfies the basic integrity properties that DAB does. In addition, Net guarantees FIFO delivery of messages, and a simple liveness property:

- FIFO *delivery:* If $\texttt{net\_mcast}_i\texttt{(m)}$ occurs before $\texttt{net\_mcast}_i\texttt{(m')}$, and $\texttt{net\_rcv}_j\texttt{(m')}$ occurs, then $\texttt{net\_rcv}_j\texttt{(m)}$ occurs before $\texttt{net\_rcv}_j\texttt{(m')}$.
- *Eventual delivery:* Suppose $\texttt{net\_mcast}_i\texttt{(m)}$ occurs after $\texttt{net\_join\_OK}_j$, and no $\texttt{fail}_i$ occurs. Then either $\texttt{net\_leave}_j$ or $\texttt{fail}_j$ or $\texttt{net\_rcv}_j\texttt{(m)}$ occurs.

Additionally, the network latency is bounded by a constant nonnegative real number Δ. The maximum message latency of Δ guaranteed by Net is intended to include any pre-send delay at the network module of the sending process, and is independent of the message size. Since an implementation of Net cannot predict the future, it must deliver messages within time Δ as long as no failures occur. In particular, if a message is sent more than Δ time before its sender fails, it must be delivered.

7 The Atom Algorithm

The Atom algorithm uses Net and CUP services as building blocks. It uses multiple instances of CUP. As before, $\texttt{fail}_i$ causes process i to stop. $\texttt{fail}_i$ is an input to all the components, i.e., Net and all instances of CUP (including dormant ones), and causes all of them to stop; $\texttt{leave}_i$ also goes directly to all the local instances of CUP, including dormant ones.

Atom defines the constant Θ, a positive real number that represents the duration of a time slot. We assume that $\Theta > \Delta$. We define the message alphabet M' of Net in term of the alphabet M of DAB:

- M_1, the set of finite sequences of elements of M. These are the bulk messages processes send.
- $M_2 = M_1 \cup \{JOIN, LEAVE\} \cup \{CUP\_INIT \times I\}$
- $M' = I \times M_2 \times \mathsf{N}$.

Each message contains either a bulk message (sequence of client messages) for a particular slot, a request to join or leave a particular slot, or a report that process has initiated consensus on behalf of a particular endpoint. Each message is tagged with the sender and the slot. The algorithm divides time and messages into slots, each of duration Θ. Each process multicasts all of its messages for a given slot in one bulk message. This is an abstraction that we make in order to simplify the presentation. In practice, the bulk message does not have to be sent as one message; a standard packet assembly/disassembly layer can be used.

Message delivery is also done in order of slots. Before delivering messages of a certain slot `s`, each process has to determine the *membership* of `s`, i.e., the set of processes from which to deliver slot `s` messages. To ensure consistency, all the processes that deliver messages for a certain slot have to agree upon its membership. Within each slot, messages are delivered in order of process indices, and for each process, messages from its bulk message are delivered in FIFO order.

7.1 Atom$_i$ Signature and Variables

The signature of Atom$_i$ includes all the interaction with the client and underlying network. In addition, Atom$_i$ has input and output actions for interacting with CUP. Since Atom uses multiple instances of CUP, at most one for each process j, actions of CUP automata are prefixed with `CUP(j)`, where `CUP(j)` is the instance of CUP used to agree in which slot process j fails. E.g., process i uses the action `CUP(j).init`$_i$ to initiate the CUP automaton associated with process j. `CUP.fail` and `CUP.leave` are *not* output actions of Atom, since they are routed directly from the environment to all instances of CUP.

Atom$_i$ also has two internal actions, `end_slot`$_i$, and `members`$_i$, which play a role in determining the membership of each slot. `end_slot(s)`$_i$ occurs at a time by which slot `s` messages from all processes should have reached i. At this point, processes from which messages are expected but do not arrive are *suspected* to have failed in this slot; we are guaranteed that these processes indeed have failed, but we are uncertain about the slot in which they fail. For each suspected process j, CUP(j) is run to have the surviving processes agree upon j's failure slot. This is needed because failed processes can be suspected at different slots by different surviving processes. After CUP reaches decisions about all the suspected processes that could have failed at slot `s`, `members(P,s)` can occur, with `P` being the agreed membership for slot `s`. When `members(P,s)`$_i$ occurs, the messages included in bulk messages that i received for slot `s` from processes in `P` are delivered (their delivery is triggered) in order of process indices.

A variable `join-slot` holds the slot at which a process starts participating in the algorithm; this will be the value of `current-slot` when `join_OK` will be issued, and the first slot for which a bulk message will be sent. If a process explicitly leaves the algorithm, its `leave-slot` holds the slot immediately following the last slot in which the process sends a bulk message. Both `join-slot` and `leave-slot` are initially ∞.

The flags `did-join-OK` and `did-leave` ensure that `join_OK` and `net_leave` actions are not performed more than once. The set `mcast-slots` tracks the slots for which the process already multicast a message (JOIN, LEAVE, or bulk). Likewise, `ended-slots` and `reported-slots` track the slots for which the `end_slot` or `members` actions, resp., were performed.

`out-buf[s]` stores the message (bulk, JOIN, or LEAVE) that is multicast for slot `s`; it initially holds an empty sequence, and in an active slot, all application messages are appended into it. A JOIN message is inserted for the slot before the `join-slot`, and a LEAVE message for the `leave-slot`. Either way, there is no overlap with a bulk message. Variables `joiners[s]` and `leavers[s]` keep track of the processes j for which `join-slot`$_j$ `=s` (resp. `leave-slot`$_j$ `=s`). `suspects[s]` is the set of processes suspected in slot `s` as determined when `end_slot(s)` occurs. `in-buf[j,s]` holds the sequence of messages received in a slot `s` bulk message from j. Its data type supports assignment, extraction of the head of the queue, and testing for emptiness. `alive[s]` is a derived variable containing the set of processes from which slot `s` messages were received.

There are three variables for tracking the status and values of the different instances of CUP. `CUP-status[j]` is initially `idle`; when CUP(j) is initiated, it becomes `running`; if a CUP_INIT message for j arrives, it becomes `req`; and when there is a decision for CUP(j), or if the process abstains from CUP(j), it becomes `done`. `CUP-req-val[j]` holds the lowest slot value associated with a CUP_INIT message for j ($\perp$ if no such message has arrived). Finally, `CUP-dec-val[j]` holds the decision reached by CUP(j), and $\perp$ if there is none.

7.2 Algorithm flow

Upon an application `join`, Atom triggers `net_join`. Once the Net responds with a `net_join_OK`, Atom calculates the `join-slot` to be $2 + \lceil \Gamma/\Theta \rceil$ slots in the future. This will allow enough time for the JOIN message to reach the other processes. A JOIN message is then inserted into `out-buf[join-slot - 1]`. Once `current-slot` is `join-slot`, `join_OK` is issued to the application.

When the application issues a `leave`, the `leave-slot` is chosen to be the ensuing slot, and a LEAVE message is inserted into `out-buf[leave-slot]`. A `net_leave` is issued after the LEAVE message has been multicast, and the `net_leave_OK` triggers a `leave_OK` to the application.

Messages multicast by the application are appended to the bulk message for the current slot in `out-buf[current-slot]`. Once a slot `s` ends, the message pertaining to this slot is multicast using `net_mcast`. If `s = join-slot - 1`, a JOIN message is sent. If `s = leave-slot`, a LEAVE message is sent, and if `s` is between `join-slot` and `leave-slot - 1`, a bulk message is sent. A received bulk message is stored in the appropriate `in-buf`. When a (j, JOIN, s) (or (j, LEAVE, s)) message is received, j is added to `joiners[s]` (resp. `leavers[s]`). Additionally, when a LEAVE message is received, `CUP.leave_detect` is triggered for all running instances of CUP.

$\texttt{end\_slot}_i\texttt{(s)}$ occurs once i should have received all the slot `s` messages sent by non-failed processes. Since such messages are sent immediately when slot `s` ends, are delayed at most Δ time in Net, and the clock difference is at most Γ, i should have all the slot `s` messages $\Delta + \Gamma$ time after it began slot `s+1`. Process i expects to receive slot `s` messages from every process in `alive[s-1]` that does not leave in slot `s`. Any process from which a slot `s` message is expected but does not arrive becomes suspected at this point, and is included in `suspects[s]`.

For every suspected process, CUP is run in order to agree upon the slot at which the process failed. Note that CUP is only performed for failed processes since we implement a perfect failure detector. The slot `s` in which the process is suspected is used as the initial value for CUP. The estimated world for CUP is `alive[s]` $\cup$ `joiners[s+1]`. This way, if k joins in slot `s+1`, k is included in the estimated world. This is needed in order to satisfy the world consistency assumption of CUP, because k can detect the same failure at slot `s+1`, and therefore participate in CUP(j). When i initiates CUP(j), it also multicasts a (CUP_INIT, j) message. If a process k does not detect the failure and does not participate, the (CUP_INIT, j) message forces k to abstain. Since Atom implements the failure detector for CUP, the effect of $\texttt{end\_slot}_i\texttt{(s)}$ also triggers `CUP(k).fail_detect(j)` actions for every suspected process j, and for every currently running instance k of CUP.

Process i abstains from CUP(j) only if (1) a (CUP_INIT,j) message has previously arrived, setting `CUP-status[j]`$_i$ `= req`; and (2) `end_slot`$_i$ has already occurred for a slot value greater than `CUP-req-val[j]`$_i$. The latter condition ensures that i abstains only from instances of CUP that it will not initiate. There are two circumstances that can lead to a process i abstaining from CUP(j). First, if i is just joining, and the failure occurs before its `join-slot`, then i will not be affected by the decision because it does not deliver any messages for this slot. Second, if j has joined and immediately failed before i could see its JOIN message, then j did not send any bulk messages prior to its failure, and thus no process will deliver any messages from j.

The `members(P,s)` action triggers the delivery of all slot `s` messages from processes in `P`. It occurs once agreement is reached about the processes to be included in `P`. Recall that the slots at which a process k is suspected by two processes i and j can differ by at most one. Therefore, `members`$_i$`(P,s)` can occur after `end_slot(s+1)`, when the suspicions for slot `s+1` are determined, since all processes that i does not suspect at slot `s+1` could not have failed prior to ending slot `s`. Thus, after i gets decisions from all instances of CUP pertaining to processes suspected in slots up to `s+1` i can deliver all slot `s` messages. The set `P` includes every process j that is alive in slot `s` and for which there is either no CUP instance running, or the CUP decision value is greater than `s`.

In [3] we prove the following: (1) Atom satisfies CUP's safety assumptions independently of CUP; (2) assuming a service that satisfies the CUP safety guarantees, Atom satisfies CUP's liveness assumptions; and (3) assuming a service that satisfies the CUP safety and liveness guarantees, Atom satisfies the DAB service safety and liveness guarantees.

7.3 Latency Analysis

In failure free executions, Atom's message latency is bounded by $\Delta + 2\Theta + 2\Gamma$. We denote this bound by Δ_{Atom}. In executions with failures, the upper bound on message latency is linear in the number of failures. In [3], we prove the following:

Lemma 1. *Consider an execution in which no process fails. If the application at process j performs* `mcast`$_j$`(m)` *when* `current-slot`$_i$ `= s` *and if process i delivers m, then i delivers m immediately after* `end_slot`$_i$`(s+1)` *occurs.*

From this lemma, we derive the following theorem:

Theorem 3. *If the application at process j performs* `mcast`$_j$`(m)` *at time t, and if process i delivers m, then i delivers m by time $t + \Delta_{Atom} = t + \Delta + 2\Theta + 2\Gamma$.*

We then turn our attention to executions in which there is a long time period with no failures. We analyze the time it takes Atom to clear the backlog it has due to past failures, and reach a situation in which message latency is bounded by the same bound as in failure free executions, namely Δ_{Atom}, barring additional failures. The fact that once failures stop for a bounded time all messages are delivered within constant time implies that in periods with f failures, Atom's latency is at most linear in the number of failing processes.

In order to analyze how long it takes Atom to reach a stable point, we need to use our bounds on CUP's running time once failures stop. We first have to assign values to the constants that were used in the analysis of CUP in Section 5.1 (δ_1 and δ_2). Recall, δ_1 is an upper bound on message latency and on failure and leave detection time, and if a message is lost due to failure, then the failure is detected at most δ_1 after the lost message was sent; and δ_2 is an upper bound on the difference between different processes' initiation times. In [3], we prove the following bounds: $\delta_1 = \Delta + 3\Theta + 2\Gamma$; and $\delta_2 = \Gamma + \Theta$.

We then consider executions in which failures do occur but there are long time periods with no failures. We analyze the time it takes Atom to clear the backlog it has due to past failures, and again reach a situation in which message latency is bounded by Δ_{Atom}, barring additional failures.

Let $t_1 = \delta_2 + 4\delta_1$, where δ_2 and δ_1 are bounds as given above for the difference between process initiation times and failure detection time, resp. By the bounds above, we get that $t_1 = \Gamma + \Theta + 4(\Delta + 3\Theta + 2\Gamma) = 4\Delta + 9\Gamma + 13\Theta$.

Assume that from time t to time $t' = t + t_1$ there are no failures. We now show that if a message m is sent after time t', and there are no failures for a period of length Δ_{Atom} after m is sent, then m is delivered within Δ_{Atom} time of when it is sent. Since the delivery order preserves the FIFO order, this also implies that any message m' sent before time t' is delivered by time t' barring failures in the Δ_{Atom} time interval after m' is sent.

Theorem 4. *Assume no process fails between time t and $t' = t+t_1$. If* `mcast(m)`$_j$ *occurs at a time t'' such that $t + t_1 \leq t''$, and no failures occur from time t'' to time $t'' + \Delta_{Atom}$, and if i delivers m, then i delivers m by time $t'' + \Delta_{Atom}$.*

7.4 Future Direction: Extending Atom to Cope with Late Messages

In this paper, we assumed deterministic network latency bounds. Since the network latency, Δ is expected to be of a smaller order of magnitude than Θ, it would not significantly hurt time bounds if conservative assumptions are made in the choice of Δ. Future research may consider networks where latency bounds cannot be ensured. E.g., some networks may support differentiated services with probabilistic latency guarantees, and loss rates may exceed those assumed in the latency analysis of the underlying reliable network (see [2, 20]).

Although Atom cannot guarantee atomic broadcast semantics while network latency exceeds its bound, it would be useful to modify Atom as to allow it to recover from such situations, and to once more provide correct semantics after network guarantees are re-established. In addition, it would be desirable to inform the application when a violation of Atom semantics occurs. There are some strategies that can be used to make Atom recover from periods in which network guarantees are violated. For example, a lost or late message can cause inaccurate failure suspicions. With Atom, if a process k is falsely suspected, it will receive a (CUP_INIT, k) message for itself. We can have the process use this as a trigger to "commit suicide", i.e., inform the application of the failure and have the application re-join as a new process (similar to the Isis [4] algorithm).

8 Conclusions

We have defined two new problems, *Dynamic Atomic Broadcast* and *Consensus with Uncertain Participants*. We have presented new algorithms for both problems. The latency of both of our algorithms depends linearly on the number of failures that occur during a particular execution, but does not depend on an upper bound on the potential number of failures, nor on the numbers of joins and leaves that happen during the execution.

Acknowledgments

We thank Alan Fekete, Rui Fan, Rachid Guerraoui, and the referees for useful comments that helped improve the presentation.

References

1. T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Apps. Symp. (RTAS)*, Jun 1996.
2. Z. Bar-Joseph, I. Keidar, T. Anker, and N. Lynch. QoS preserving totally ordered multicast. In Franck Butelle, editor, *5th Int. Conf. On Prin. Of Dist. Sys. (OPODIS)*, pp. 143–162, Dec 2000. Special issue of *Studia Informatica Universalis*.
3. Z. Bar-Joseph, I. Keidar, and N. Lynch. Early-delivery dynamic atomic broadcast. Tech. Rep. MIT-LCS-TR-840, MIT Lab. for Comp. Sci., Apr 2002.
4. K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Comp. Soc. Press, 1994.
5. G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *21st ACM Symp. on Prin. of Dist. Comp. (PODC)*, July 2002. To appear.
6. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Comp. Surveys*, 33(4):1–43, Dec 2001.
7. F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Journal of Real-Time Systems*, 2:195–212, 1990.
8. F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Inf. Comp.*, 118:158–179, Apr 1995.
9. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Dist. Comp.*, 4(4):175–187, Apr 1991.
10. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, Oct 1990.
11. B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Dist. Sys. Eng.*, 1:29–36, 1993.
12. A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast. In *9th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 297–309, 1990.
13. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *chapter in: Distributed Systems*. ACM Press, 1993.
14. S. Katz, P. Lincoln, and J. Rushby. Low-overhead time-triggered group membership. In *11th Int. Wshop. on Dist. Algs. (WDAG)*, pp. 155–169, 1997.
15. H. Kopetz and G. Grunsteidl. TTP - a protocol for fault-tolerant real-time systems. *IEEE Computer*, pp. 14–23, January 1994.
16. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
17. M. Merritt and G. Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *14th Int. Symp. on DIStributed Comp. (DISC)*, Oct 2000.
18. L. Rodrigues and P. Verissimo. xAMp, A multi-primitive group communications service. In *IEEE Int. Symp. on Reliable Dist. Sys. (SRDS)*, pp. 112–121, Oct 1992.
19. L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. A dynamic light-weight group service. In *15th IEEE Int. Symp. on Reliable Dist. Sys. (SRDS)*, pp. 23–25, Oct 1996.
20. P. Verissimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of lan-based protocols. In *10th IFAC Wshop. on Dist. Comp. Control Systems*, Sep 1991.