Early-Delivery Dynamic Atomic Broadcast

Ziv Bar-Joseph Idit Keidar
Nancy Lynch
MIT Laboratory for Computer Science
zivbj@mit.edu, idish@theory.lcs.mit.edu, lynch@theory.lcs.mit.edu

April 17, 2002

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 synchronous system, 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 Unknown 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 [17] 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 [20, 27], 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 synchronous crash failure model. Specifically, we assume that the processes solving DAB have access to approximately-synchronized local clocks and to a lower-level dynamic network that guarantees timely message delivery between 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., [21]) 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 unknown 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 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 [26], that is, in a time-free crash failure model where processes are equipped with *perfect failure detectors* [5]. The failure detector is external to CUP; it is implemented by Atom. CUP uses a strategy similar to previous early-deciding algorithms for consensus with a predetermined set of participants [13], 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 an upper bound on the number of potential failures, nor on the number of processes that leave.

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 or rounds that is linear in the number of failures (see [2]).

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 [3]), or retransmissions (see [28]). 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 7.7.

In summary, this paper makes the following main contributions: (1) the definitions of two new problems for dynamic networks, expressed by the DAB and CUP services; (2) an early-delivery DAB algorithm, Atom, which exhibits constant latency in the absence failures; (3) a new early-deciding algorithm for solving CUP in a fail-stop model; 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 and its analysis. We then turn to the presentation of Atom: Section 6 specifies the environment and model assumptions for Atom, and Section 7 contains a detailed presentation of the Atom algorithm and its analysis. Section 8 concludes the paper. The Appendix contains rigorous correctness proofs for both CUP and Atom.

2 Related Work

A dynamic universe, where processes join and leave, was first considered in the context of view-oriented group communication work [7], pioneered by the Isis [4] system. The first analysis of time bounds of message delivery in synchronous group communication systems was performed by Cristian [9]. Our service resembles the services provided by group communication systems; although we do not export membership to the application, it is computed, and would be easy to export.

View-oriented group communication systems, including systems designed for synchronous systems and real-time applications (e.g., Cristian's [9], xAMp [25], 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 system uses an atomic broadcast primitive to agree upon group membership. Since, unlike CUP, the atomic broadcast service works with a static universe, a process join has to be agreed upon before any new membership change is handled (voluntary leaves are not considered). Therefore, 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 group communication system supporting a variety of communication primitives for real-time applications. The presentation of xAMp in [25] focuses on the various communication primitives and 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. RTCAST is a real-time group communication system, for which a detailed analysis of membership latency was conducted [1]. 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; the cost of this primitive is not considered in the analysis.

Some group membership services avoid running the full-scale membership for join and leaves by using light-weight group membership [15] services; they use an atomic broadcast service to disseminate join and leave messages in a consistent manner, without running the full-scale group membership algorithm. However, unlike our CUP service, the atomic broadcast service such systems use do not tolerate uncertainty about the set of participants. Therefore, a race condition between a join and a concurrent failure can cause such light-weight group services (e.g., [23, 12, 15]) to violate the semantics of the underlying heavy-weight membership services. Those light-weight group services that do preserve the underlying heavy-weight membership semantics (e.g. [24]), do incur extra delivery latencies whenever joins and leaves occur.

Other work on group membership in synchronous and real-time systems, e.g., [19, 18] 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. [16, 10, 8], considered a static universe, where processes could fail but not join. Thus, this work did not consider the DAB problem.

In a previous paper [3], 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 [22, 6] considers different services, including (one shot) consensus, for infinitely many processes in asynchronous shared memory models. Chockler and Malkhi [6] 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 [22] study consensus under different concurrency models; in their terminology, our model assumes *unbounded* congruency and $[1, \infty]$-participation, which means that at least one process must participate and there is no bound on the number of participants. They show that with these assumptions, in an asynchronous shared memory model, infinitely many bits are required in order to solve consensus. The algorithms they give are not fault tolerant (they tolerate only initial failures). To the best of our knowledge, atomic broadcast has not been considered in a similar context. Moreover, these problems were not considered in message-passing models, and it is not clear that a canonical transformation from the shared memory model the message-passing model applies to a setting with infinitely many processes.

3 Dynamic Atomic Broadcast Service Specification

We now present the DAB service specification. Our universe consists of an infinite ordered set of endpoints, I. The specification of DAB is parameterized by a message alphabet, M. The signature of the DAB(M) service is presented in Figure 1.

Input:

```
join_i, leave_i, fail_i, i∈I
mcast_i(m), m∈M, i∈I
```

Output:

```
join_OK_i, leave_OK_i, i∈I
rcv_i(m), m∈M, i∈I
```

Figure 1: The signature of the DAB(M) service.

We do not consider recoveries from failure or rejoining after leaving. In other words, there cannot be multiple "incarnations" at a single endpoint. Instead of new incarnations, consider the same client joining at new endpoints.

Assumptions about the application: DAB(M) assumes that its application satisfies the following safety conditions:

- For each $i \in I$:
  - At most one $\texttt{join}_i$ and at most one $\texttt{leave}_i$ occur.
  - If $\texttt{leave}_i$ occurs, then it is preceded by $\texttt{join\_OK}_i$.
  - Any $\texttt{mcast}_i\texttt{(m)}$ has a preceding $\texttt{join\_OK}_i$ but no preceding $\texttt{leave}_i$ or $\texttt{fail}_i$.
- At most one `mcast(m)` occurs for each particular m.

DAB guarantees: Given an application that satisfies the above constraints, DAB(M) satisfies the properties we now specify.

We first specify some basic integrity properties, both safety and liveness. We later specify the properties related to the ordering and reliability of messages.

Basic safety properties:

- *Join/leave integrity:* For each i:
  - At most one $\texttt{join\_OK}_i$ and at most one $\texttt{leave\_OK}_i$ occur.
  - If $\texttt{join\_OK}_i$ occurs then it is preceded by $\texttt{join}_i$.
  - If $\texttt{leave\_OK}_i$ occurs then it is preceded by $\texttt{leave}_i$.
- *Message integrity:*
  - No two $\texttt{rcv}_j\texttt{(m)}$ actions occur for the same m and j.
  - If $\texttt{rcv}_j\texttt{(m)}$ occurs for some j then it is preceded by $\texttt{mcast}_i\texttt{(m)}$ for some i.

Basic liveness properties:

- *Eventual join:* If $\texttt{join}_i$ occurs then either $\texttt{fail}_i$ or $\texttt{join\_OK}_i$ occurs.
- *Eventual leave:* If $\texttt{leave}_i$ occurs then either $\texttt{fail}_i$ or $\texttt{leave\_OK}_i$ occurs.

To specify the ordering and reliability guarantees of DAB, 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.

Safety properties:

- *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 S contains m and m'', then S also contains m'.
- *Receive gap-freedom:* If 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.

Liveness property:

- *Multicast liveness:* If $\texttt{mcast}_i\texttt{(m)}$ occurs and no $\texttt{fail}_i$ occurs, then S contains m.
- *Receive liveness:* If S contains m, 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 S, $\texttt{rcv}_i\texttt{(m')}$ also occurs.

4 Consensus with Unknown Participants – Specification

In this section we define the problem of Consensus with Unknown Participants (CUP). CUP is an adaptation of the problem of fail-stop uniform consensus to a dynamic setting in which the set of participants is not known 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 uses an underlying reliable network, and a perfect failure detector.

We begin with a description of CUP's external signature (interface). We then specify the assumptions that CUP makes about its environment, including the application, the underlying network, and the external failure detector. We separate these into *safety* and *liveness* assumptions. Finally, we specify CUP's safety and liveness guarantees. CUP's safety guarantees depend on only the safety assumptions, that is, they are not allowed to be violated even if the liveness assumptions do not hold. On the other hand, CUP's liveness guarantees depend on both the safety and liveness assumptions.

4.1 External Signature

The CUP specification uses the following data types:

- I, an infinite ordered set of *endpoints*. Each endpoint in I corresponds to a potential participant in CUP.
- V, a totally ordered set of *values*. Initial values and decision values are elements of V.

Input:

```
initᵢ(v,W), v ∈ V, W ⊆ I, W finite, i ∈ I      // i initiates with value v,world W
abstainᵢ, i ∈ I                                 // i abstains
net_rcvᵢ(m), m ∈ M_CUP, i ∈ I                   // i receives message m
leaveᵢ, i ∈ I                                   // i leaves
leave_detectᵢ(j), j, i ∈ I                      // i detects that j has left
failᵢ, i ∈ I                                    // i fails
fail_detectᵢ(j), j, i ∈ I                       // i detects that j has failed
```

Output:

```
decideᵢ(v), v ∈ V, i ∈ I                        // i decides on value v
net_mcastᵢ(m),  m ∈ M_CUP, i ∈ I                // i multicasts m
```

Figure 2: The signature of CUP.

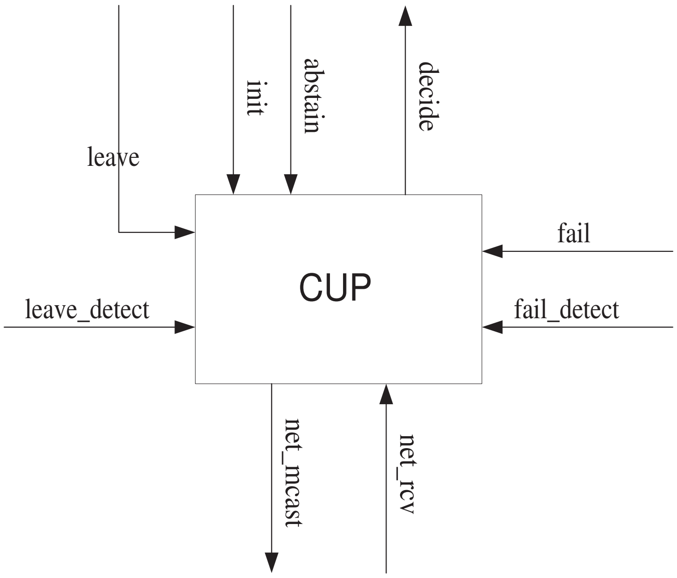

Figure 3: Interface diagram for CUP.

- M_{CUP}, a message alphabet.

The external signature of CUP is presented in Figure 2, and depicted in Figure 3.

The interface describes four kinds of interaction: "normal" interaction with clients of the CUP service, interaction with a multicast network, communication involving leaves and leave detection, and communication involving failures and failure detection.

Normal interaction with clients: A process may participate in the CUP service in two ways: it may provide an initial value, in which case we say that the process *initiates* CUP, or it may decline to provide an initial value, in which case we say that it *abstains*. Participant $i \in I$ initiates CUP using the $\texttt{init}_i\texttt{(v,W)}$ action. Here, `v` is i's initial value, and `W` is its initial *world*, that is, the set of processes that i expects to participate in CUP. Participant i abstains using the $\texttt{abstain}_i$ action. 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.

An environment assumption ensures that, if any process expects i to participate in CUP, i will in fact participate, unless it leaves or fails. CUP reports the consensus decision value to process i using the $\texttt{decide}_i\texttt{(v)}$ action.

Multicast network: The network interface consists of the `net_mcast` and `net_rcv` actions.

Leaves: A participant can leave the CUP service voluntarily using the $\texttt{leave}_i$ action. We assume that the environment provides a *leave detector*: the $\texttt{leave\_detect}_i\texttt{(j)}$ action is used to notify i that j has left the algorithm voluntarily.

Failures: The $\texttt{fail}_i$ action represents the failure of endpoint i. We assume that the environment provides a failure detector, which uses the $\texttt{fail\_detect}_i\texttt{(j)}$ action to notify i that j has failed.

4.2 Environment Assumptions

Here we list and explain the assumptions that CUP makes about its environment. We classify these as safety and liveness assumptions. Formally, each of the properties given here is a *trace property* ([21, Ch. 8]).

4.2.1 Safety assumptions

The first assumption expresses 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.

- *Well-formedness:* For any $i \in I$,

  1. At most one $\texttt{init}_i$ or $\texttt{abstain}_i$ event[1] occurs.
  2. At most one $\texttt{leave}_i$ event occurs.
  3. At most one $\texttt{fail}_i$ event occurs.
  4. No $\texttt{leave}_i$ or $\texttt{fail}_i$ precedes an $\texttt{init}_i$.

The next assumption says that, while the worlds `W` suggested by different participants in their `init` events do not have to be identical, CUP's environment must guarantee that they have a certain kind of consistency. Namely, 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.

- *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.

The next property describes the correctness of the message deliveries: 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.

[1] An "event" is an occurrence of an action in a sequence.

- *Message integrity:* There is a mapping from `net_rcv` events to preceding `net_mcast` events, such that the same message in M_{CUP} appears in both events, and such that no two $\texttt{net\_rcv}_i$ events for the same i map to the same `net_mcast` event. Moreover, two $\texttt{net\_rcv}_i$ events that map to `net_mcast` events of the same sender occur in the same order as the `net_mcast` events.

The next two properties describe assumptions about leaves and leave detection. The first says that leave detection is "accurate", in the sense that the occurrence of a $\texttt{leave\_detect}_i\texttt{(j)}$ implies that j has really left; it also includes a simple well-formedness condition. 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 network 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.

- *Accurate leave detector:* For any $i, j \in I$, at most one $\texttt{leave\_detect}_i\texttt{(j)}$ event occurs, and if $\texttt{leave\_detect}_i\texttt{(j)}$ 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 $\texttt{net\_rcv}_i\texttt{(m)}$.

The final safety assumption says that failure detection is accurate.

- *Accurate failure detector:* For any $i, j \in I$, at most one $\texttt{fail\_detect}_i\texttt{(j)}$ event occurs, and if $\texttt{fail\_detect}_i\texttt{(j)}$ occurs, then it is preceded by a $\texttt{fail}_j$.

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.

4.2.2 Liveness assumptions

The first liveness assumption says 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.

- *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 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.

The final liveness assumption says that the leaving or failure of any process that belongs to an initiator's `W` set is detected by that initiator, unless it finishes by deciding, leaving, or failing.

- *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.

4.3 CUP Service Guarantees

Now we list CUP's service guarantees. Again, we classify these as safety and liveness properties. As we noted earlier, CUP's safety guarantees depend only on its safety assumptions, whereas CUP's liveness guarantees depend on both its safety and liveness assumptions.

Formally, each individual property is a trace property. The complete specification consists of two general trace properties whose respective sets of traces are defined by the following predicates:

1. The conjunction of all the CUP safety assumptions implies all the CUP safety guarantees.

2. The conjunction of all the CUP safety and liveness assumptions implies all the CUP liveness guarantees.

4.3.1 Safety guarantees

The first guarantee expresses well-formedness conditions saying that only participants that have initiated can decide, and each participant decides at most once.

- *Well-formedness:* For any $i \in I$,

    1. If $\texttt{decide}_i$ occurs then it is preceded by an $\texttt{init}_i$.
    2. At most one $\texttt{decide}_i$ occurs.

The next two guarantees are the main agreement and validity guarantees for consensus. The uniform agreement property says that everyone who decides agrees. The validity property has two parts: it says that any decision value is some participant's initial value, and moreover, that any participant's decision is no greater than its initial value. The latter is not a "standard" property for consensus but is needed for our use in Atom.

- *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.
    2. If $\texttt{init}_i\texttt{(v',*)}$ occurs then $v \leq v'$.

4.3.2 Liveness guarantees

CUP provides one liveness guarantee, which says that any participant that initiates and neither leaves nor fails must eventually decide. We do not make such a guarantee for a participant that abstains, that is, participants that abstain need not be informed of the decision value.

- *Termination:* If an $\texttt{init}_i$ event occurs then a $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs.

5 The CUP Algorithm

In this section, we present our implementation of CUP.

5.1 Modeling Assumptions and Conventions

We use the I/O automaton model of Lynch and Tuttle (see, e.g., [21, Ch. 8]), using standard precondition/effect (guarded command) pseudo-code, augmented with one new construct: effects may include statements of the form `trigger(a)`, where `a` is an output action. Formally, we assume the automaton's state contains a special FIFO buffer `trigger-buffer`. The `trigger(a)` statement adds `a` to the end of `trigger-buffer`. The action at the head of `trigger-buffer` is always enabled, and gets removed from `trigger-buffer` when it is performed. No other state changes are associated with action `a`.

The $\texttt{fail}_i$ action described in the CUP interface represents the failure of endpoint i. In terms of the algorithm, we interpret this to mean that once $\texttt{fail}_i$ occurs, i performs no more locally controlled actions, and input actions have no effect on the state. We treat this as a general convention, and do not include event handlers for $\texttt{fail}_i$ actions in our pseudo-code.

5.2 The Algorithm

Figures 4 and 5 contain the CUP implementation for a particular endpoint $i \in I$. The algorithm includes no internal actions. Therefore, the signature consists of the actions indexed by this particular i in the external signature of CUP (see Section 4). The message alphabet M_{CUP} is specialized to the set of messages of the following forms:

- (i,r,v,W), where $i \in$ I, r $\in$ N, v $\in$ V, and W is a finite subset of I.
- (i,OUT,r), where $i \in$ I and r $\in$ N,

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

```
mode ∈ { ⊥, running, done}, initially ⊥
round ∈ N, initially 0
for each r ∈ N+, j ∈ I:
   value[r,j] ∈ V ∪ { ⊥ }, initially ⊥
   world[r,j], a finite subset of I or ⊥, initially ⊥
for each r ∈ N+
   out[r], a finite subset of I, initially {}
   failed[r], a finite subset of I, initially {}

Derived variables:

for each r ∈ N+
   out-by[r], a finite subset of I, defined as ∪_{r' ≤ r} out[r']
   failed-by[r], a finite subset of I, defined as ∪_{r' ≤ r} failed[r']
```

Figure 4: CUP_i state.

```
init_i(v,W)
Eff: if mode = ⊥ then
        mode ← running
        round ← 1
        trigger(net_mcast_i(i,1,v,W))

net_mcast_i(i,r,v,W) where r ≥ 2
Pre: mode = running
     r = round + 1
     W = world[round,i] \ out[round] \ failed[round]
     // All messages for the previous round have been received.
     ∀ j ∈ W: value[round,j] ≠ ⊥
     W ≠ { } ∧ v = min{value[round,j] | j ∈ W}
     // No decision can be made.
     ¬ ∀ j ∈ world[round,i] \ out[round]:
        value[round,j] = value[round,i] ∧ world[round,j] ⊆ world[round,i]
Eff: round ← r

net_rcv_i(j,r,v,W)
Eff: if mode ≠ done ∧ j ∉ failed-by[r] then
        value[r,j] ← v
        world[r,j] ← W

abstain_i
Eff: if mode = ⊥ then
        mode ← done
        trigger(net_mcast_i(i,OUT))

decide_i(v)
Pre: mode = running
     value[round,i] ≠ ⊥
     ∀ j ∈ world[round,i] \ out[round]:
        value[round,j] = v ∧ world[round,j] ⊆ world[round,i]
Eff: mode ← done
     trigger(net_mcast_i(i,OUT))

net_rcv_i(j,OUT)
Eff: if mode ≠ done then
        let r = min {r′ ∈ N+ | value[r′,j] = ⊥}
        out[r] ← out[r] ∪ {j}

leave_i
Eff: mode ← done

leave_detect_i(j)
Eff: if mode ≠ done then
        let r = min {r′ ∈ N+ | value[r′,j] = ⊥}
        out[r] ← out[r] ∪ {j}

fail_detect_i(j)
Eff: if mode ≠ done then
        let r = min {r′ ∈ N+ | value[r′,j] = ⊥}
        failed[r] ← failed[r] ∪ {j}
```

Figure 5: CUP_i transitions.

The code works as follows. When an $\texttt{init}_i\texttt{(v,W)}$ input occurs, process i triggers a `net_mcast(i,1,v,W)` to send its initial value `v` and estimated world `W` to all processes, including itself.

For each round $\texttt{r} \geq 2$, process i performs an explicit $\texttt{net\_mcast}_i\texttt{(i,r,v,W)}$ to multicast its round `r` value `v` and world `W`. The world `W` is determined to be the set of processes that i thinks are still active, that is, 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.

When an $\texttt{abstain}_i$ input occurs, process i sends an `OUT` message, so that other processes will know not to wait for further messages from it, and stops participating in the algorithm.

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 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, process i just stops participating in the algorithm. When $\texttt{leave\_detect}_i\texttt{(j)}$ occurs, process i records that j is out; when this occurs, the lossless leave assumption ensures that i has already received all the messages j sent. The round that is recorded for the leave is the first round after the round of the last message received from j.

Process i knows that another process has failed if it learns about the failure via a `fail_detect` event.

In the next section, we prove the algorithm's correctness. In Section 5.3, we show that 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.

5.3 The Early-Deciding Property

We now show that 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.

We start with some more lemmas.

Lemma 5.1 *If* $\texttt{init}_i\texttt{(*,W)}$ *occurs prior to* $\texttt{init}_j$*, then* $j \in \texttt{W}$*.*

Proof: The environment well-formedness assumption implies that j does not leave or fail before it initiates, and hence does not leave or fail before i initiates. Therefore, by world consistency, $j \in \texttt{W}$. ∎

Invariant 5.1 *If* `(i,1,*,W)` *and* `(j,2,*,*)` *are in the Net then* $j \in \texttt{W}$*.*

Proof: By strong induction. For the inductive step, assume that, in the final state of the execution, `(i,1,*,W)` and `(j,2,*,*)` are in the Net. Then both $\texttt{init}_i$ and $\texttt{init}_j$ events appear in the execution. If $\texttt{init}_i$ precedes $\texttt{init}_j$, then Lemma 5.1 implies the result, so assume that $\texttt{init}_j$ precedes $\texttt{init}_i$.

Since a round 2 message from j is in the Net, a round 1 message (j,1,*,W') is also. Then Lemma 5.1 implies that $i \in$ W'.

We claim that j does not leave or fail before the $\texttt{init}_i$. Suppose for the sake of contradiction that it does. Then the net_mcast(j,2,*,*) event precedes the $\texttt{init}_i$. Then environment well-formedness implies that i does not fail or leave prior to the net_mcast(j,2,*,*) event, because it initiates after this event. Also, i does not abstain, because it initiates. And i does not decide prior to the net_mcast(j,2,*,*) event, because that precedes the $\texttt{init}_i$. Therefore, $i \notin \texttt{failed[1]}_j \cup \texttt{out[1]}_j$ in the pre-state of the net_mcast(j,2,*,*) event, so $i \in \texttt{world[1,j]}_j \setminus \texttt{failed[1]}_j \cup \texttt{out[1]}_j$ in that state. The precondition of net_mcast implies that $\texttt{value[1,i]}_j \neq \perp$ in the pre-state, that is, j has received a round 1 message from i before the net_mcast(j,2,*,*). But this cannot happen, because $\texttt{init}_i$ happens after the net_mcast(j,2,*,*). This contradiction implies that j does not leave or fail before the $\texttt{init}_i$. Then world consistency implies that $j \in$ W, as needed. ■

In the rest of this section, we consider a situation where no failures happen from some point onward in an execution, and where the rounds of all processes are at most r at the point where failures cease. The following lemma says that all round r+2 messages that are ever sent have the same world component.

Lemma 5.2 *Suppose that* $\texttt{r} > 0$. *Suppose that there is a point t in an execution such that every process has* round $\leq$ r *at point t, and no* fail *events happen from t onward.*
If net_mcast(i,r+2,v,W) *and* net_mcast(j,r+2,v',W') *both occur in the execution, then* W = W'.

Proof: We show that W $\subseteq$ W'. The other direction is analogous.

The two sets are determined in the precondition of net_mcast, as follows:
$\texttt{W} = \texttt{world[r+1,i]}_i \setminus \texttt{out[r+1]}_i \setminus \texttt{failed[r+1]}_i$, where the values of the last two terms are taken from the pre-state of net_mcast(i,r+2,v,W), and
$\texttt{W'} = \texttt{world[r+1,j]}_j \setminus \texttt{out[r+1]}_j \setminus \texttt{failed[r+1]}_j$, where the values of the last two terms are taken from the pre-state of net_mcast(j,r+2,v',W'). Invariant A.7 implies that $\texttt{W} = \texttt{world[1,i]}_i \setminus \texttt{out-by[r+1]}_i \setminus \texttt{failed-by[r+1]}_i$, where the values of the last two terms are taken from the pre-state of net_mcast(i,r+2,v,W), and
$\texttt{W'} = \texttt{world[1,j]}_j \setminus \texttt{out-by[r+1]}_j \setminus \texttt{failed-by[r+1]}_j$, where the values of the last two terms are taken from the pre-state of net_mcast(j,r+2,v',W').

Consider some $k \in$ W. The precondition of net_mcast(i,r+2,v,W) implies that in the pre-state, $\texttt{value[r+1,k]}_i \neq \perp$, that is, i has received a round r+1 message from k. This means that k has previously sent a round r+1 message. Since (by assumption) $\texttt{r} > 0$, it follows that $\texttt{r}+1 \geq 2$, which means that k has sent a round 2 message. Invariant 5.1, applied to any state after both net_mcast(j,1,*,*) and net_mcast(k,2,*,*) have occurred, implies that k is in the world component of j's round 1 message, and so k is put into $\texttt{world[1,j]}_j$ when that is defined. To prove that $k \in$ W', it suffices to show that k is never placed into either of the sets $\texttt{out-by[r+1]}_j$ or $\texttt{failed-by[r+1]}_j$.

First, we show that k is never placed into $\texttt{out-by[r+1]}_j$. Suppose for the sake of contradiction that k is put into $\texttt{out-by[r+1]}_j$ at some point in the execution. Then consider some state that occurs after this has happened, and that is not before the pre-state of net_mcast(i,r+2,v,W). In this state, we have both `value[r+1,k]`$_i \neq \perp$ and $k \in$ `out-by[r+1]`$_j$. This contradicts Invariant A.4. Therefore, k is never placed into `out-by[r+1]`$_j$.

Second, we show that k is never placed into `failed-by[r+1]`$_j$. Suppose for the sake of contradiction that k is put into `failed-by[r+1]`$_j$ at some point in the execution. Then k fails in the execution, which implies that it fails before point t. But we have already noted that k sends a round `r` $+1$ message during the execution. It does not send this before point t, because that would mean that it would reach round `r` $+1$ before point t, contradiction our assumptions. So k sends the round `r` $+1$ message after point t, and so it cannot fail before point t, a contradiction. Therefore, k is never placed into `failed-by[r+1]`$_j$. ■

The next lemma says that, under the same assumptions as for the previous lemma, all the round `r+2` messages have the same value component.

Lemma 5.3 *Suppose that* `r` $>$ `0`*. Suppose that there is a point t in an execution such that every process has* `round` $\leq$ `r` *at point t, and no* `fail` *events happen from t onward.*
If `net_mcast(i,r+2,v,W)` *and* `net_mcast(j,r+2,v',W')` *both occur in the execution, then* `v` $=$ `v'`*.*

Proof: Process i determines v as the minimum of all values `value[r+1,k]`$_i$ for all $k \in$ `W`, and process i determines v' as the minimum of all values `value[r+1,k]`$_j$ for all $k \in$ `W'`. Lemma 5.2 implies that `W` $=$ `W'`. Since values are consistent (by Invariant A.2), the sets of values over which the two minima are taken are identical. Therefore, `v` $=$ `v'`. ■

Finally, we prove the main early-deciding theorem. It says that, if no failures happen from some point onward and the rounds of all processes are at most `r` when failures cease, then no CUP participant ever advances beyond round `r` $+2$. Since we have already proved termination, this implies that all active CUP participants decide by round `r` $+2$.

Theorem 5.4 *Suppose that* `r` $>$ `0`*. Suppose that there is a point t in the execution such that every process has* `round` $\leq$ `r` *at point t, and no* `fail` *events happen from t onward.*
Then every process always has `round` $\leq$ `r` $+2$*.*

Proof: Lemmas 5.2 and 5.3 yield a common value and world for round `r+2` messages. Fix `v'` and `W'` to be the common value and world, respectively.

We show that the precondition of `net_mcast(i,r+3,*,*)` can never be true, which implies that such an event can never happen. This implies that every process always has `round` $\leq$ `r` $+2$. Suppose for the sake of contradiction that the precondition of `net_mcast(i,r+3,v,W)` is true in some reachable state s, for some fixed i.

Since the precondition holds in s, `world[r+2,i]`$_i \neq \perp$ in s, and so Invariant A.1 implies that some `(i,r+2,v'',W'')` message is in the Net in s, where `v''` $=$ `value[r+2,i]`$_i$ and `W''` = `world[r+2,i]`$_i$. Since `v'` and `W'` are the common value and world for round `r+2` messages, this implies that `value[r+2,i]`$_i =$ `v'` and `world[r+2,i]`$_i =$ `W'`.

We show that for all $j \in$ `world[r+2,i]`$_i$ $\setminus$ `out[r+2]`$_i$, `value[r+2,j]`$_i =$ `value[r+2,i]`$_i$ and `world[r+2]`$_i \subseteq$ `world[r+2,i]`$_i$. This suffices to show that the final precondition fails, which yields a contradiction.

Fix $j \in$ `world[r+2,i]`$_i$ $\setminus$ `out[r+2]`$_i$. Since `failed[r+2]`$_i = \{\,\}$, if follows that $j \in$ `world[r+2,i]`$_i$ $\setminus$ `out[r+2]`$_i$ $\setminus$ `failed[r+2]`$_i$. The precondition of the `net_mcast` then implies that `value[r+2,j]`$_i \neq \perp$ in state s. Invariant A.1 then implies that some `(i,r+2,v''',W''')` message is in the Net in s, where `v'''` $=$ `value[r+2,j]`$_i$ and `W'''` $=$ `world[r+2,i]`$_i$. Since `v'` and `W'` are the only value and world for round 2 messages, this implies that `value[r+2,j]`$_i$ $=$ `v'` and `world[r+2,j]`$_i$ $=$ `W'` in state s. Thus, `value[r+2,j]`$_i$ $=$ `value[r+2,i]`$_i$ and `world[r+2]`$_i$ $\subseteq$ `world[r+2,i]`$_i$, as needed. ■

Note that this proof does not work for the case where `r=0`, because of potential differences in the initial worlds of correct processes. Consider, for example, an execution in which no process ever fails, and some process, k, leaves after sending a round 1 message. Process k may be included in the initial world of process i but not in the initial world of another process j, if j initiates CUP after k leaves. In this case, i takes k's round 1 message into account when choosing its round 2 message, while j does not (because k is not in j's initial world). This scenario can only occur in round 1, because no process can send a round 2 message before j initiates.

For the case where `r = 0`, the best we can state is:

Corollary 5.5 *Suppose there is a point t in the execution such that every process has* `round` $= 0$ *at point t, and no* `fail` *events happen from t onward.*
Then every process always has `round` ≤ 3.

Proof: This is immediate from Theorem 5.4, using `r` $= 1$. ■

5.4 Timing Assumptions

For the sake of analyzing the performance of the CUP algorithm, we use timed I/O automata [21, Ch. 23]. We can regard an ordinary I/O automaton as a special case of the timed model, in which arbitrary amounts of time can pass between events. All the safety results carry over to this model.

For this analysis, we add an extra assumption: we assume that any action that is enabled either gets performed or gets disabled by another action, before any time passes.

5.5 Latency Analysis

We now analyze the algorithm's latency in executions in which there are time bounds on certain environment actions. We assume the following bounds:

1. δ_1 is an upper bound on *message latency*. That is, if a `net_rcv(m)` event occurs, the time since the corresponding `net_mcast(m)` is at most δ_1.

2. δ_2 is an upper bound on *failure and leave detection time*. Moreover, if a message is lost due to failure, then the failure is detected at most δ_2 after the lost message was sent. More precisely,

   (a) Assume `init`$_i$`(*,W)` occurs with $j \in W$ and `fail`$_j$ or `leave`$_j$ occurs at time t. Then `fail_detect`$_i$`(j)`, `leave_detect`$_i$`(j)`, `decide`$_i$, `leave`$_i$, or `fail`$_i$ occurs by time $t + \delta_2$.

   (b) Define $U = \cup_{k \in I}\{W|$ `init`$_k(*, W)$ occurs$\}$. Assume $i, j \in U$ and `net_mcast`$_j$`(m)` occurs at time t but no `net_rcv`$_i$`(m)` occurs. Then `fail_detect`$_i$`(j)`, `leave_detect`$_i$`(j)`, `decide`$_i$, `leave`$_i$, or `fail`$_i$ occurs by time $t + \delta_2$.

3. δ_3 is an upper bound on the time difference between the initiation time of different processes. More precisely:
   Assume some process initiates at time t and does not fail by time $t + \delta_1$. Assume further that `init`$_i$`(*, W)` occurs. Then, every process $j \in W$ initiates, abstains, leaves, or fails by time $t + \delta_3$.

In practice, the failure detection time would be at least as large as the message latency. We therefore assume that $\delta_2 \geq \delta_1$.

We now use the above bounds on the environment to establish bounds on CUP's running times. The next lemma bounds the time it takes from when some process initiates CUP until all processes terminate round 1.

Lemma 5.6 *Assume that some process initiates CUP at time t and does not fail by time $t + \delta_1$. Then by time $t + \delta_2 + \delta_3$, every process that initiates either terminates round 1, or leaves, or fails.*

Proof: Let i be a process that initiates and does not leave or fail by time $t + \delta_2 + \delta_3$. We now show that i terminates round 1 by time $t + \delta_2 + \delta_3$. If i decides by time $t + \delta_2 + \delta_3$, then we are done. We therefore assume that i does not decide by this time.

In order to terminate round 1, i has to have a round 1 message from every process $j \in$ `world[1,i]`$_i$ $\setminus$ `out[1]`$_i$ $\setminus$ `failed[1]`$_i$. That is, for every process $j \in$ `world[1,i]`$_i$, i has to receive a round 1 message or an OUT message from j, or a `fail_detect`$_i$`(j)` or a `leave_detect`$_i$`(j)` event.

Fix a process $j \in$`world[1,i]`$_i$, i.e., j is in i's initial world. Since some process initiates at time t, by our assumption on initiation times, j initiates, abstains, leaves, or fails by time $t + \delta_3$.

If j fails or leaves by time $t + \delta_3$, then by our assumption on failure and leave detection times, `fail_detect`$_i$`(j)` or `leave_detect`$_i$`(j)` occurs by time $t + \delta_2 + \delta_3$ (since we assume that i does not decide, leave, or fail by this time), and we are done.

Assume now that j does not fail or leave by time $t + \delta_3$. Since j is in i's initial world, j either initiates or abstains by this time, at which point j sends a round 1 message or an OUT message (resp.). If i receives this message, i receives it by time $t + \delta_3 + \delta_1$. If i does not receive this message, `fail_detect(j)`$_i$ or `leave_detect(j)`$_i$ occurs by time $t + \delta_3 + \delta_2$.

Since $\delta_2 \geq \delta_1$, we get that for every $j \in$`world[1,i]`$_i$, by time $t + \delta_3 + \delta_2$, i either receives a round 1 message or an OUT message from j or a `fail_detect(j)`$_i$ or `leave_detect(j)`$_i$ event occurs. ■

The following lemma bounds the duration of subsequent rounds.

Lemma 5.7 *Assume that by time t, every process that initiates CUP either terminates round* `r > 0`*, or decides, or leaves, or fails. Then, by time $t + \delta_2$, every process that initiates CUP either terminates round* `r+1`*, or decides, or leaves, or fails.*

Proof: Consider a process i that initiates CUP and does not leave or fail or decide by time $t + \delta_2$. We now show that i terminates round `r+1` by time $t + \delta_2$.

In order to terminate round `r+1`, i has to have a round `r+1` message from every process $j \in$ `world[r+1,i]`$_i$ $\setminus$ `out[r+1]`$_i$ $\setminus$ `failed[r+1]`$_i$. That is, for every process $j \in$ `world[r+1,i]`$_i$, i has to either receive a round `r+1` message or an OUT message from j, or a `fail_detect`$_i$`(j)` or a `leave_detect`$_i$`(j)` event has to occur.

Fix a process $j \in$`world[r+1,i]`$_i$. Process j must have initiated. By time t, j terminates round `r+1`, or decides, or leaves, or fails. If j leaves or fails by time t, then `fail_detect(j)`$_i$ or `leave_detect(j)`$_i$ occurs by time $t + \delta_2$. Otherwise, j sends a round `r+1` message or an OUT message (in case it decides) by time t. If i receives this message, i receives it by time $t + \delta_1$. Otherwise, `fail_detect(j)`$_i$ or `leave_detect(j)`$_i$ occurs by time $t + \delta_2$. Since $\delta_2 \geq \delta_1$, we get that i terminates round `r+1` by time $t + \delta_2$. ■

Using the two lemmas above, we get the following bound on the running time of an execution of CUP with `r` rounds.

Lemma 5.8 *Assume that some process initiates CUP at time t and does not fail by time $t + \delta_1$. If i decides at round* `r > 0`*, it does so by time $t + \delta_3 + \mathtt{r}\delta_2$.*

Proof: By Lemma 5.6, by time $t + \delta_3 + \delta_2$, every process that initiates CUP either terminates round 1, or leaves, or fails. By iterative application of Lemma 5.7, we get that by time $t + \delta_3 + \delta_2 + (\mathtt{r} - \mathtt{1})\delta_2 = t + \delta_3 + \mathtt{r}\delta_2$, every process that initiates CUP either terminates round `r`, or decides, or leaves, or fails. ■

As a consequence of the above lemmas and the early-deciding theorem of the previous section we get the following theorem:

Theorem 5.9 *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 $t + \delta_3 + 3\delta_2$.*

Proof: Let `r` be the highest value of `round` of any process at time t. Since some process initiated CUP by time t, `r` $>$ `0`. By Theorem 5.4, every process that decides, decides at the end of round `r+2` at the latest.

We consider two cases. First, if `r > 1`, then by Invariant A.12, every process that initiated CUP has either terminated round `r-1` or left or failed by time t. By applying Lemma 5.7 three times, we get that every process that initiates CUP either terminates round `r+2` or leaves or fails by time $t + 3\delta_2$. Therefore, in this case, every process that decides, decides by time $t + 3\delta_2$.

Next, assume that `r = 1`. Since some process initiates CUP by time t and does not fail, by Lemma 5.6, by time $t + \delta_3 + \delta_2$, every process that initiates CUP either terminates round 1, or leaves, or fails. By applying Lemma 5.7 twice, we get that every process that initiates CUP either terminates round `r+2` or leaves or fails by time $t + \delta_3 + 3\delta_2$. Therefore, in this case, every process that decides, decides by time $t + \delta_3 + 3\delta_2$. ■

6 Environment and Model Assumptions for Atom

6.1 Timing Assumptions

We model time using a continuous global variable `now`, which holds the real time. This is a real variable, initially 0. We assume that it increases with derivative 1. Each endpoint i is equipped with a local clock, $\mathtt{clock}_i$, modeled by a continuous, bijective, monotonically increasing function from the nonnegative R to the nonnegative R.

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. Formally, when any locally controlled action of any process that is part of our local algorithm is enabled, then before any time passes, the action is either performed or becomes disabled.

6.2 Reliable Network Assumptions

We assume that we are given a low-level reliable network service Net. Like DAB, Net is parameterized by a message alphabet, M.

The Net(M) signature is defined in Figure 6. The actions are the same as those of DAB, except that they are prefixed with `net_`.

Input:

$\texttt{net\_join}_{\texttt{i}}$, $\texttt{net\_leave}_{\texttt{i}}$, $\texttt{fail}_{\texttt{i}}$, i∈I,
$\texttt{net\_mcast}_{\texttt{i}}\texttt{(m)}$, m∈M, i∈I

Output:

$\texttt{net\_join\_OK}_{\texttt{i}}$, $\texttt{net\_leave\_OK}_{\texttt{i}}$, i∈I,
$\texttt{net\_rcv}_{\texttt{i}}\texttt{(m)}$, m∈M, i∈I

Figure 6: The signature of the Net service.

Net(M) assumes that its application satisfies the same basic safety conditions as those specified above for DAB(M), except that action names are preceded with `net_`. Assuming the application satisfies these conditions, Net(M) satisfies a number of safety and liveness properties.

First, Net satisfies the basic properties specified above for DAB: join/leave integrity, message integrity, eventual join, and eventual leave. All of these properties are the same as for DAB, except that action names are prefixed with `net_`.

In addition, Net guarantees FIFO delivery of messages:

- 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 $\texttt{net\_rcv}_j\texttt{(m')}$.

Net(M) also satisfies the following liveness property:

- *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. Formally, Net(M) guarantees:

- *Message latency:* If $\texttt{net\_rcv}_j\texttt{(m)}$ occurs, then the real time elapsed since the corresponding $\texttt{net\_mcast}_i\texttt{(m)}$ is at most 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.

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 consists of a collection of processes corresponding to the different endpoints in I. It uses Net and CUP services as building blocks. It uses multiple instances of CUP.

7.1 Data Types

Atom defines the constant Θ, a positive real number. This will represent a time slot. We assume that $\Theta > \Delta$.

Recall that M represents the message alphabet of DAB. We will use M' to represent the message alphabet of Net. We define the message alphabet of Net in term of the alphabet of Atom:

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

M' is the complete message alphabet of Net. 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.

7.2 Using the Net and CUP

The Net service alphabet is instantiated with M'. That is, Atom uses a service Net(M') to implement the service DAB(M). Atom uses multiple instances of CUP, at most one for each process j.

As before, a $\mathtt{fail}_i$ action causes process i to stop. $\mathtt{fail}_i$ actions go to all the components, i.e., Net and all instances of CUP (including dormant ones), and cause all of them to stop taking any locally controlled actions. Since $\mathtt{fail}_i$ actions cannot be intercepted, we do not include them in the code.

$\mathtt{leave}_i$ actions also go directly to all the local instances of CUP, including dormant ones.

7.3 Atom Algorithm Overview

The algorithm divides time, and respectively, messages, into slots. As time advances, each process advances through slot. The duration of a slot is Θ.

Each process multicasts all of its messages for a given slot in one bulk message. This is a useful abstraction that we make in order to simplify the presentation and analysis of the Atom algorithm. In practice, the bulk message does not have to be sent as one message; a standard packet assembly/disassembly layer can be used to provide all-or-nothing behavior.

Message delivery is also done in order of slots. Before delivering messages of a certain slot $\mathtt{s}$, each process has to determine the *membership* of this slot, that is, the set of processes from which to deliver messages in this slot. To ensure total order, all the processes that deliver messages for a certain slot have to agree upon the membership of each slot. For each slot, messages are delivered in the order of process indices, and for each process, the messages are unpacked from its bulk message and delivered in FIFO order.

7.4 Signature

The signature of Atom at process i, Atom$_i$, is presented in Figure 7. It includes all the interaction with the client and all the interaction with the underlying network. The implementation of Atom uses CUP as a building block. Hence Atom$_i$ has additional 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)`. For example, process i uses the action $\texttt{CUP(j).init}_i$ 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.

The signature of Atom_i also includes two internal actions, `end_slot`, and `members`. These two actions play a role in determining the membership for each slot. $\texttt{end\_slot(s)}_i$ occurs at a time by which slot `s` messages from all processes should have reached process i. At this point, processes from which messages are expected but do not arrive are *suspected* to have failed. 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 process i performs $\texttt{members(P, s)}_i$, all 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.

Input:

$\texttt{join}_\texttt{i}$, $\texttt{leave}_\texttt{i}$
$\texttt{net\_join\_OK}_\texttt{i}$, $\texttt{net\_leave\_OK}_\texttt{i}$
$\texttt{mcast}_\texttt{i}\texttt{(m)}$, $\texttt{m}\in\texttt{M}$
$\texttt{net\_rcv}_\texttt{i}\texttt{(m)}$, $\texttt{m}\in\texttt{M}'$
$\texttt{fail}_\texttt{i}$
$\texttt{CUP(j).decide}_\texttt{i}\texttt{(v)}$, $\texttt{v}\in\mathbb{N}$

Output:

$\texttt{join\_OK}_\texttt{i}$, $\texttt{leave\_OK}_\texttt{i}$
$\texttt{net\_join}_\texttt{i}$, $\texttt{net\_leave}_\texttt{i}$
$\texttt{net\_mcast}_\texttt{i}\texttt{(i, m, s)}$, $\texttt{m}\in\texttt{M}_2$, $\texttt{s}\in\mathbb{N}$
$\texttt{rcv}_\texttt{i}\texttt{(m)}$, $\texttt{m}\in\texttt{M}$
$\texttt{CUP(j).init}_\texttt{i}\texttt{(v, W)}$, $\texttt{v}\in\mathbb{N}$
$\texttt{CUP(j).abstain}_\texttt{i}$
$\texttt{CUP(j).leave\_detect}_\texttt{i}\texttt{(j)}$, $\texttt{j}\in\texttt{I}$
$\texttt{CUP(j).fail\_detect}_\texttt{i}\texttt{(j)}$, $\texttt{j}\in\texttt{I}$

Internal:

$\texttt{end\_slot}_\texttt{i}\texttt{(s)}$, $\texttt{s}\in\mathbb{N}$
$\texttt{members}_\texttt{i}\texttt{(P, s)}$, P set of I, $\texttt{s}\in\mathbb{N}$

Figure 7: Atom_i: Signature.

7.5 Pseudo-code

The Atom_i code is presented in Figures 8–10. The state components are presented in Figure 8.

Recall that we do not assume that processes execute the algorithm from the beginning of time. Rather, the application issues an explicit `join` event, and waits for a `join_OK`. The 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 ∞, so as to be larger than any actual slot number they are compared with.

```
clock∈R, initially∈[0, Γ/2]; dynamic type: continuous functions

join-slot ∈ N ∪ ∞, initially ∞
leave-slot ∈ N ∪ ∞, initially ∞
did-join-OK, boolean, initially false
did-leave, boolean, initially false

mcast-slots ⊆ N, initially { }
ended-slots ⊆ N, initially { }
reported-slots ⊆ N, initially { }

for every s ∈ N
  out-buf[s] ∈ M2, initially empty sequence of M
  joiners[s] ⊆ I, initially { }
  leavers[s] ⊆ I, initially { }
  suspects[s] ⊆ I, initially { }

for every s ∈ N, j ∈ I
  in-buf[j,s], j ∈ I, s ∈ N, finite sequence of M or ⊥, initially ⊥

for every j ∈ I\{ i }
  CUP-status[j] ∈ { idle, req, running, done }, initially idle
  CUP-req-val[j]  ∈ N ∪ { ⊥ }, initially ⊥
  CUP-dec-val[j] ∈ N ∪ { ⊥ }, initially ⊥

derived variables:

current-slot ∈ N = ⌊ clock / Θ ⌋

for every s ∈ N
  alive[s] ⊆ I = { j |  in-buf[s,j] ≠ ⊥ }
```

Figure 8: Atom_i: State.

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

`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.

The variables `joiners[s]` and `leavers[s]` keep track of the processes j for which $\texttt{join-slot}_j$ =s (resp. $\texttt{leave-slot}_j$ =s). `suspects[s]` is the set of processes suspected in slot `s` as determined when `end_slot(s)` occurs.

The variable `in-buf[j,s]` is a finite sequence of messages received in a slot `s` bulk message from process j. The data type finite sequence supports assignment, extraction of the head of the queue, and testing for emptiness.

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.

`alive[s]` is a derived variable, storing the set of processes from which slot `s` bulk messages were received.

```
join_i
Eff: trigger(net_join_i)

net_join_OK_i
Eff: join-slot ← current-slot + 2 + ⌈ Γ/Θ ⌉
     out-buf[join-slot - 1] ← JOIN

join_OK_i
Pre: did-join-OK = false
     current-slot = join-slot
Eff: did-join-OK ← true

leave_i
Eff: if (join-slot ∈ N) then
        leave-slot ← max(current-slot, join-slot) + 1
        out-buf[leave-slot] ← LEAVE

net_leave_i
Pre: did-leave = false
     leave-slot ∈ mcast-slots
Eff: did-leave ← true

net_leave_OK
Eff: trigger(leave_OK_i)

mcast_i(m)
Eff: if (join-slot ≤ current-slot < leave-slot) then
        append m to out-buf[current-slot]

net_mcast_i(i, m, s)
Pre: join-slot ∈ N
     join-slot - 1 ≤ s ≤ leave-slot
     current-slot = s+1
     s ∉ mcast-slots
     m = out-buf[s]
Eff: mcast-slots ← mcast-slots ∪ { s }

net_rcv_i(j, JOIN, s)
Eff: joiners[s+1] ← joiners[s+1] ∪ { j }

net_rcv_i(j, LEAVE, s)
Eff: leavers[s] ← leavers[s] ∪ { j }
     foreach (k such that CUP-status[k] = running) do
        trigger(CUP(k).leave_detect_i(j))

net_rcv_i(j, m, s), m sequence of M
Eff: in-buf[j,s] ← m
```

Figure 9: Atom_i: Transitions related to multicast, join, and leave.

In Figure 9 we present the first part of Atom's transitions, including transitions related to joining, leaving, multicasting messages, and receiving messages from the network. Transitions related to membership and totally ordered delivery are presented in Figure 10.

When the application issues a `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 the `current-slot` reaches `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 to the other processes 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.

When a bulk message is received, it is stored in the appropriate `in-buf`. When a JOIN (LEAVE) message is received, the sender is added to the `joiners` (resp. `leavers`) set for the appropriate slot. Additionally, when a LEAVE message is received, `CUP.leave_detect` is triggered for all running instances of CUP.

Process i performs $\texttt{end\_slot}_i\texttt{(s)}$ once it should have received all the slot `s` messages sent by other non-failed processes. Since slot `s` messages are sent immediately when slot `s` ends, messages are delayed at most Δ time in Net, and the clock difference is at most Γ, process i should have all the non-failed processes' slot `s` messages $\Delta + \Gamma$ time after slot `s+1` began. At this time, `clock` $> (s+1)\Theta + \Delta + \Gamma$. Process i expects to receive slot `s` bulk messages from all the processes that are in `alive[s-1]`, except for those that are leaving in slot `s`. Any process from which a slot `s` bulk 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. 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 a (CUP-INIT,j) message has previously arrived, setting $\texttt{CUP-status[j]}_i$ `= req`, and only if $\texttt{end\_slot}_i$ has already occurred for a slot value greater than $\texttt{CUP-req-val[j]}_i$. The latter condition ensures that i abstains only from instances of CUP that it will not initiate. This is because the network guarantees that when a process fails, at most one slot bulk message from this process is lost (since we assume that $Delta \leq \Theta$). This implies that the detection of j's failure by two non-failed processes can occur at most one slot apart. Therefore, if $\texttt{end\_slot}_i$ has already occurred for a slot value greater than $\texttt{CUP-req-val[j]}_i$, i will never suspect j.

The `members(P, s)` action triggers the delivery of all slot `s` messages from processes in `P`. It can only occur once agreement has been reached about the processes to be included in `P`. Since the slot at which a process k is suspected by two processes i and j can differ by at most one, $\texttt{members}_i\texttt{(P, s)}$ can occur after i receives decision from all instances of CUP pertaining to processes suspected in

```
end_slot_i(s)
Pre: join-slot ≤ s
     leave-slot = ∞
     s ∉ ended-slots
     clock ⟩ (s+1)Θ + Δ + Γ
Eff: ended-slots ← ended-slots ∪ { s }
     suspects[s] ← (alive[s-1] ∪ joiners[s] \ leavers[s]) \ alive[s]
     foreach (j ∈ suspects[s]) do
        trigger(CUP(j).init_i(s, alive[s] ∪ joiners[s+1]))
        net_mcast_i(i, (CUP-INIT, j), s)
        CUP-status[j] ← running
        foreach (k such that CUP-status[k] = running) do
           trigger(CUP(k).fail_detect_i(j))

net_rcv_i(j, (CUP-INIT, k), s)
Eff: if (CUP-status[k] = idle ∨ CUP-req-val[k] ⟩ s) then
        CUP-status[k] ← req
        CUP-req-val[k] ← s

CUP(j).abstain_i
Pre: CUP-status[j] = req
     ∃s ∈ ended-slots : s > CUP-req-val[j]
Eff: CUP-status[j] ← done

CUP(j).decide_i(s)
Eff: CUP-status[j] ← done
     CUP-dec-val[j] ← s

members_i(P, s)
Pre: s = min{ ended-slots \ reported-slots }
     s + 1 ∈ ended-slots
     ∀j ∈ (suspects[s] ∪ suspects[s+1]) : CUP-status[j] = done
     P = { j ∈ alive[s] | CUP-dec-val[j] = ⊥ ∨ CUP-dec-val[j] > s }
Eff: reported-slots ← reported-slots ∪ { s }
     foreach j ∈ P, in order of indices do
         while in-buf[j,s] not empty do
            trigger(rcv_i(head(in-buf[i,s])))
```

Figure 10: Atom_i: Transitions related to membership and message delivery.

slots up to `s+1`. Therefore, $\texttt{members}_i$`(P, s)` must occurs after `end_slot(s+1)`, when the suspicions for slot `s+1` are determined. The set `P` includes every process j that is alive in slot `s` and for which there is either no CUP instance running (in which case j was not suspected), or the CUP decision value is greater than `s`.

7.6 Latency Analysis

In this section we analyze the latency guarantees of Atom. In Section 7.6.1 we show that in failure free executions, Atom's message latency is bounded by $\Delta + 2\Theta + 2\Gamma$. We denote this bound by Δ_{Atom}. In Section 7.6.2, we assign values to the constants that were used in the analysis of CUP in Section 5.5 (δ_1, δ_2, and δ_3). Then, in Section 7.6.3, we consider executions in which failures do occur but 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 failure, Atom's latency is at most linear in the number of failing processes.

7.6.1 Failure free executions

Lemma 7.1 *The time from when process j starts slot* `s` *(i.e.,* `current-slot`$_j$ *becomes* `s`*) until process i performs* `end_slot`$_i$`(s+1)` *is at most* $\Delta + 2\Theta + 2\Gamma$.

Proof: According to its preconditions, `end_slot`$_i$`(s)` occurs $2\Theta + \Delta + \Gamma$ time after i starts slot `s`. Since the difference between two processes' clocks is at most Γ, i starts slot `s` at most Γ time after j starts this slot. ■

Lemma 7.2 *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.*

Proof: If i delivers m, `rcv`$_i$`(m)` is triggered during the `Members`$_i$`(P,s)` action. Since no process fails, `suspects[s]`$_i \cup$ `suspects[s+1]`$_i$ is an empty set, and thus the only precondition that needs to be satisfied in order to perform `Members`$_i$`(P,s)` is `s+1` $\in$ `ended-slots`$_i$, which is true immediately after `end_slot`$_i$`(s+1)` occurs. ■

As a direct result of these two lemmas, we get the following theorem:

Theorem 7.3 *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* $t + \Delta_{Atom} = t + \Delta + 2\Theta + 2\Gamma$.

7.6.2 CUP bounds

We now assign values to the constants used in the analysis of CUP in Section 5.5. Recall, δ_1 is an upper bound on message latency; δ_2 is an upper bound on failure and leave detection time, and if a message is lost due to failure, then the failure is detected at most δ_2 after the lost message was sent; and δ_3 is an upper bound on the difference between different processes' initiation times.

Lemma 7.4 $\delta_1 = \Delta$

Proof: By definition, both Δ and δ_1 are defined to be upper bounds on the underlying network latency. ■

Lemma 7.5 $\delta_2 = \Delta + 3\Theta + 2\Gamma$

Proof: Assume that `CUP(k).init`$_i$`(*,W)` occurs with $j \in W$. Assume that one of the following happens at time t: `fail`$_j$, `leave`$_j$, or `net_mcast`$_j$`(m)` for a message `m` that is lost because j subsequently fails. Let `s` be the value of `current-slot`$_j$ at time t. Assume also that by time $t + \Delta + 3\Theta + 2\Gamma$, i does not decide, leave, or fail, so `CUP-status[k]`$_i$ `= running` and i is active at this time. We have to show that by this time, `fail_detect`$_i$`(j)` or `leave_detect`$_i$`(j)` occurs.

If j fails at time t, then j's slot `s` message is never sent, and therefore i detects the failure and invokes `CUP(k).fail_detect`$_i$`(j)` during `end_slot`$_i$`(s)` at the latest. By Lemma 7.1, this occurs by time $t + 2\Theta + \Delta + 2\Gamma$. Likewise, if j sends a message m while `current-slot`$_j$ `= s`, and m is lost, then by the FIFO nature of the network, j's slot `s` message is also lost and i detects j's failure during `end_slot`$_i$`(s)` at the latest.

Assume next that j leaves when `current-slot`$_j$ `= s`, i.e., j's `leave-slot` is `s+1`. If i receives a LEAVE message from j, it receives it before `end_slot`$_i$`(s+1)` occurs, and immediately triggers `CUP(k).leave_detect`$_i$`(j)`. Otherwise, i receives no slot `s+1` message from j and suspects j and invokes `CUP(k).fail_detect`$_i$`(j)` during `end_slot`$_i$`(s+1)`. This occurs by time $t + 3\Theta + \Delta + 2\Gamma$. ■

Lemma 7.6 $\delta_3 = \Gamma + \Theta$

Proof: Assume that some process process l initiates CUP(k) at time t and does not fail by time $t + \Delta$. Assume further that `CUP(k).init`$_i$`(*, W)` occurs with $j \in W$. We have to show that j initiates, abstains, leaves, or fails by time $t + \Gamma + \Theta$.

Process l triggers `CUP(k).init`$_l$`(s, *)` during the `end_slot`$_l$`(s)` action, and $k \in$`suspects`$_l$`[s]`. If j initiates CUP(k), there is a slot `s'` such that j triggers `CUP(k).init`$_j$ during the `end_slot`$_j$`(s')` action, and $k \in$`suspects`$_j$`[s']`. By Invariant A.19, `s'` $\leq$ `s+1`. Therefore, `CUP(k).init`$_j$ occurs no later than time $t + \Gamma + \Theta$, and we are done.

Assume now that j does not initiate CUP(k), and does not leave or fail by time $t + \Gamma + \Theta$. We now show that j abstains from CUP(k) by time $t + \Gamma + \Theta$.

When `CUP(k).init`$_l$`(s, *)` is triggered, l multicasts a (CUP-INIT, k) message. By Lemma A.8, `net_join_OK`$_j$ occurs before l initiates CUP(k), that is, before l multicasts this message. Moreover, by assumption, l does not fail by time $t + \Delta$ and j does not leave or fail by time $t + \Delta$ (because $\Delta \leq \Theta$). Therefore, j receives this message by time $t + \Delta$, which is before time $t + \Gamma + \Theta$. After j receives this message, `CUP-status[k]`$_j$ is `req` and `CUP-req-val[k]`$_j$ is less than or equal to `s`. By time $t + \Gamma + \Theta$, `end_slot`$_j$`(s+1)` occurs and the condition for `CUP(k).abstain`$_j$ becomes true, and remains true until `CUP(k).abstain`$_j$ occurs and changes `CUP-status[k]`$_j$. Therefore, before any time passes, `CUP(k).abstain`$_j$ occurs. ■

7.6.3 Failure free periods

We now 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 $t_1 = \delta_3 + 4\delta_2$, where δ_3 and δ_2 are bounds as given above for the difference between process initiation times and failure detection time, resp. From Lemmas 7.6 and 7.5 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 7.7 *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 $t'' + \Delta_{Atom}$.*

Proof: By Lemma 7.5, by time $t+\delta_2$ all the processes detect all the failures that occur by time t. Therefore, no process initiates an instance of CUP after time $t+\delta_2$. Since no failures occur after time $t+\delta_2$, by Theorem 5.9, all CUP instances that i initiates terminate by time $t+\delta_2+\delta_3+3\delta_2 = t+t_1$.

Let `s` be the value of $\texttt{current-slot}_j$ at time t'' (i.e., when $\texttt{mcast(m)}_j$ occurs). By Lemma 7.1, process i performs $\texttt{end\_slot}_i\texttt{(s+1)}$ by time $t''+\Delta+2\Theta+2\Gamma = t''+\Delta_{Atom}$. At this time, there are no active CUP instances, because CUP instances pertaining to failures that occurred before time t have all terminated and no new failures occur until time $t''+\Delta_{Atom}$. Therefore, for every slot `s'` $\leq$ `s`, in order of slot numbers, $\texttt{Members(P, s')}_i$ becomes enabled until it occurs. So $\texttt{Members(P, s)}_i$ occurs before any time passes. If i delivers m, $\texttt{rcv}_i\texttt{(m)}$ is triggered during the $\texttt{Members}_i\texttt{(P, s)}$ action, so $\texttt{rcv}_i\texttt{(m)}$ also occurs before any time passes. ∎

7.7 Extending Atom to Cope with Late Messages

In this paper, we assumed a synchronous model with deterministic network latency guarantees. 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.

In ongoing research we are considering networks where latency bounds are more likely to be violated. For example, some networks may support differentiated services with probabilistic latency guarantees. Moreover, loss rates may exceed the bounds assumed in the implementation of the reliable network. Such networks can be represented using the timed-asynchronous [11] failure model.

Although our algorithm cannot guarantee atomic broadcast semantics while network latency and reliability guarantees are violated, it is important for the algorithm to be able 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, and when the correct semantics are resumed (following the failure awareness approach of [14]).

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. In order to recover from such a situation, we could have the process "commit suicide" in such a situation, that is inform the application of the failure and have the application re-join as a new process. The full modification of Atom for this setting is ongoing work.

8 Conclusions

We have defined two new problems, *Dynamic Atomic Broadcast* and *Consensus with Unknown 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 and Rachid Guerraoui for 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 Applications Symposium (RTAS)*, June 1996.

[2] Z. Bar-Joseph, I. Keidar, T. Anker, and N. Lynch. QoS preserving totally ordered multicast. Technical Report MIT-LCS-TR-796, MIT Laboratory for Computer Science, January 2000. Url: http://theory.lcs.mit.edu/~idish/Abstracts/qos.html.

[3] Z. Bar-Joseph, I. Keidar, T. Anker, and N. Lynch. QoS preserving totally ordered multicast. In F. Butelle, editor, *5th International Conference On Principles Of DIstributed Systems (OPODIS)*, pages 143–162, December 2000. Special issue of *Studia Informatica Universalis.*

[4] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[6] G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *21st ACM Symposium on Principles of Distributed Computing (PODC)*, July 2002. To appear.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Comput. Surv.*, 33(4):1–43, December 2001. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

[8] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2:195–212, 1990.

[9] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1991.

[10] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Inform. Comput.*, 118:158–179, April 1995. Early version in FICS15, June, 1985.

[11] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[12] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Commun. ACM*, 39(4):64–70, April 1996.

[13] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, October 1990.

[14] 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.

[15] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Distributed Systems Engineering*, 1:29–36, 1993.

[16] A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast. In *9th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 297–309, 1990.

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

[18] S. Katz, P. Lincoln, and J. Rushby. Low-overhead time-triggered group membership. In *11th International Workshop on Distributed Algorithms (WDAG)*, pages 155–169, 1997. LNCS 1320.

[19] H. Kopetz and G. Grunsteidl. Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, pages 14–23, January 1994.

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

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

[22] M. Merritt and G. Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *14th International Symposium on DIStributed Computing (DISC)*, October 2000.

[23] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer Verlag, 1991.

[24] 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 International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–25, Oct. 1996. also Cornell University Technical Report, TR96-1611, August, 1996.

[25] L. Rodrigues and P. Verissimo. xAMp, A multi-primitive group communications service. In *11thIEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 112–121, Oct. 1992.

[26] R. D. Schlichting and F. B. Schneider. Fail stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.

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

[28] P. Verissimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of lan-based protocols. In *10th IFAC Workshop on Distributed Computer Control Systems*, September 1991.

A Correctness Proofs

A.1 Correctness of the CUP Algorithm

We consider the system consisting of a composition of automata CUP_i, one for each $i \in I$. We consider a restricted set of executions of this composition—those in which the environment safety assumptions are all satisfied. The invariants we state throughout this section should be interpreted as saying that the stated property is true for all states that occur in such executions.

A.1.1 General invariants

We say that a message is *in the Net* if a `net_mcast` event for that message has occurred or is in a `trigger-buffer`.

The first invariant lists an assortment of basic constraints. They can be proved using induction.

Invariant A.1 *1.* $\texttt{value[r,i]}_j = \perp$ *if and only if* $\texttt{world[r,i]}_j = \perp$.

2. *If* $\texttt{value[r,i]}_j = \texttt{v} \neq \perp$ *and* $\texttt{world[r,i]}_j = \texttt{W} \neq \perp$ *then an* `(i,r,v,W)` *message is in the Net.*
3. *If* `(i,r,*,*)` *is in the Net then* $\texttt{round}_i \geq \texttt{r}$.
4. *If* $\texttt{mode}_i = \perp$ *then* $\texttt{round}_i = 0$.
5. *If* $\texttt{mode}_i = \texttt{running}$ *then some* `(i,1,*,*)` *message is in the Net.*
6. *If* $i \in \texttt{failed[r]}_j$ *then* $\texttt{fail}_i$ *has occurred.*
7. *If* $i \in \texttt{failed[r]}_j$ *and* $\texttt{s} \geq \texttt{r}$*, then* $\texttt{value[s,i]}_j = \perp$.

The next invariant expresses consistency of values and worlds of the same process at different places in the system.

Invariant A.2 *1. If messages* `(j,r,v,W)` *and* `(j,r,v',W')` *are in the Net then* `v = v'` *and* `W = W'`.

2. *If* $\texttt{value[r,j]}_i \neq \perp$ *and* $\texttt{value[r,j]}_{i'} \neq \perp$ *then* $\texttt{value[r,j]}_i = \texttt{value[r,j]}_{i'}$.
3. *If* $\texttt{world[r,j]}_i \neq \perp$ *and* $\texttt{world[r,j]}_{i'} \neq \perp$ *then* $\texttt{world[r,j]}_i = \texttt{world[r,j]}_{i'}$.
4. *If* $\texttt{value[r,j]}_i \neq \perp$ *and* $\texttt{world[r,j]}_i \neq \perp$ *and a message* `(j,r,v,W)` *is in the Net then* $\texttt{value[r,j]}_i = \texttt{v}$ *and* $\texttt{world[r,j]}_i = \texttt{W}$.

The next two invariants describe some facts that follow from the existence of OUT messages and from the detection of leaves.

Invariant A.3 *1. If an* `(i,OUT)` *message is in the Net then* $\texttt{mode}_i = \texttt{done}$.

2. *If* $i \in \texttt{out[r]}_j$ *then* $\texttt{mode}_i = \texttt{done}$.

Proof: By induction. Part 2 uses the accurate leave detector assumption. ■

Invariant A.4 *If* $i \in \texttt{out[r]}_k$ *and* $\texttt{s} \geq \texttt{r}$*, then no message of the form* `(i,s,*,*)` *is in the Net, and for all* j, $\texttt{value[s,i]}_j = \perp$.

Proof: By strong induction. First, we claim that a $\texttt{net\_mcast}_i$ event cannot convert the invariant from true to false by falsifying the conclusion while leaving the hypothesis true. This is because, if the hypothesis is true, then $i \in \texttt{out[r]}_k$ in the pre-state of the $\texttt{net\_mcast}_i$, which implies, by Invariant A.3, that $\texttt{mode}_i = \texttt{done}$. But the precondition of $\texttt{net\_mcast}_i$ requires that $\texttt{mode}_i = \texttt{running}$, a contradiction.

The key steps are, therefore, those that make the hypothesis true. Index i can be added to $\texttt{out[r]}_k$ by receipt of an OUT message by k or by a $\texttt{leave\_detect}_k\texttt{(i)}$. An OUT message may result from a previous $\texttt{abstain}_i$ that occurs when $\texttt{mode}_i = \perp$, or a previous $\texttt{decide}_i$ event.

For $\texttt{abstain}_i$, by Invariant A.1, we know that in the pre-state of the $\texttt{abstain}_i$, $\texttt{round}_i = 0$. Then Invariant A.1 implies that in the pre-state, no message of the form `(i,*,*,*)` is in the Net, and for all j and all s, $\texttt{value[s,i]}_j = \perp$. Once the $\texttt{abstain}_i$ happens, $\texttt{mode}_i$ becomes `done`, which means that no later messages are sent.

For $\texttt{decide}_i$, the FIFO assumption for message delivery implies that the $\texttt{decide}_i$ event must have occurred when $\texttt{round} = \texttt{r} - 1$. Invariant A.1 then implies that in the pre-state of the $\texttt{decide}_i$, the conclusion of the invariant holds. Since the $\texttt{decide}_i$ event sets $\texttt{mode}_i$ to `done`, i sends no further messages, so the conclusion continues to hold.

For $\texttt{leave\_detect}_k\texttt{(i)}$, we know by the lossless leave assumption that before the $\texttt{leave\_detect}_k\texttt{(i)}$ occurs, k has already received every message that has ever been net_mcast by i. Since k explicitly checks that it has no values from i for round `r`, there are no such messages in the Net. ∎

The following says that any value that appears anywhere in the system is some participant's initial value.

Invariant A.5 *1. If* `(i,r,v,W)` *is in the Net then there exists j and* `W'` *such that* `(j,1,v,W')` *is in the Net.*

2. If $\texttt{value[r,k]}_i = \texttt{v} \neq \perp$ *then there exists j and* `W'` *such that* `(j,1,v,W')` *is in the Net.*

Proof: We show Parts 1 and 2 together by induction on the length of a finite execution.
Base: Trivial, because no messages are initially in the Net and no values are initially non-$\perp$.
Inductive step: We first show part 1. The interesting steps are those in which a message `(i,r,v,W)` is put into the Net. If $\texttt{r} = 1$ then `(i,r,v,W)` is put into the Net by an $\texttt{init}_i\texttt{(v,W)}$ event, which puts the `net_mcast` into $\texttt{trigger-buffer}_i$. But this immediately satisfies the conclusion. On the other hand, if $\texttt{r} \geq 2$, then `(i,r,v,W)` is put into the Net by an explicit `net_mcast(i,r,v,W)` step. In this case, `v` is obtained from a set of values already in i's `value` array in the pre-state. The induction hypothesis, part 2, then implies that some `(j,1,v,W')` is already in the Net, as needed.

For part 2, the key step is a $\texttt{net\_rcv}_i\texttt{(k,r,v,W)}$ for some `W`. In the pre-state of such a step, message `(k,r,v,W)` is in the Net. The inductive hypothesis, part 1, then implies that some `(j,1,v,W')` is already in the Net, as needed. ∎

The following invariant asserts that processes are always in their own worlds.

Invariant A.6 *1. If a message* `(i,r,v,W)` *is in the Net then* $i \in \texttt{W}$.

2. If $\texttt{world[r,i]}_j \neq \perp$ *then* $i \in \texttt{world[r,i]}_j$.

Proof: We prove part 1 by induction on the length of the execution, with a trivial base case.
Inductive step: The interesting steps are those in which a message `(i,r,v,W)` is put into the Net. If $\texttt{r} = 1$, then this is done by an $\texttt{init}_i\texttt{(v,W)}$ step. In this case, the environment well-formedness

assumption implies that no $\texttt{leave}_i$ or $\texttt{fail}_i$ event precedes the $\texttt{init}_i$, and so the world consistency assumption implies that $i \in \texttt{W}$, as needed. On the other hand, if $\texttt{r} \geq 2$, then (i,r,v,W) is put into the Net by an explicit net_mcast(i,r,v,W) step. In this case, the precondition says that $\texttt{mode}_i =$ running and $\texttt{world[1,i]}_i \neq \bot$ in the pre-state. In this pre-state, i is not in any $\texttt{failed[r]}_i$ set, because if it were, Invariant A.1 would imply that i has failed, and it would not be able to perform the net_mcast. Also, in this pre-state, i is not in any $\texttt{out[r]}_i$ set, by Invariant A.3. Therefore, i is included in W, because of the way W is defined.

Part 2 follows follows from part 1 and Invariant A.1. ∎

The following invariant describes consequences of the definition of a round r+1 world and value:

Invariant A.7 *If* (i,r+1,v,W) *is in the Net, for* $\texttt{r} \geq 1$*, then:*

1. *For every* $j \in \texttt{W}$, $\texttt{world[r,j]}_i \neq \bot$.
2. $\texttt{W} = \texttt{world[r,i]}_i \setminus \texttt{out[r]}_i \setminus \texttt{failed[r]}_i$.
3. $\texttt{v} = \texttt{min}\ \{\texttt{value[r,j]}_i\ :\ j \in \texttt{W}\}$.
4. $\texttt{W} = \texttt{world[1,i]}_i \setminus \texttt{out-by[r]}_i \setminus \texttt{failed-by[r]}_i$.

Proof: Part 1 is proved by an easy induction; the key step is net_mcast(i,r+1,v,W), and the conclusion follows immediately from the precondition.

Given part 1, we prove part 2 by induction. Now the interesting steps are $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$, $\texttt{net\_rcv}_i\texttt{(j,OUT)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, and $\texttt{fail\_detect}_i\texttt{(j)}$. The fact that $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ yields the property follows immediately from the precondition. A $\texttt{net\_rcv}_i\texttt{(j,OUT)}$ event could only falsify the property if $j \in \texttt{W}$ and the event puts j into $\texttt{out[r]}_i$. However, part 1 implies that $\texttt{world[r,j]}_i \neq \bot$ in the post-state, and hence $\texttt{value[r,j]}_i \neq \bot$ in the post-state. But this would cause the post-state to violate Invariant A.4, a contradiction. A similar argument shows that $\texttt{leave\_detect}_i\texttt{(j)}$ cannot falsify the property. Finally, $\texttt{fail\_detect}_i\texttt{(j)}$ could only falsify the property if $j \in \texttt{W}$ and the event puts j into $\texttt{failed[r]}_i$. However, part 1 implies that $\texttt{world[r,j]}_i \neq \bot$ in the post-state, and hence $\texttt{value[r,j]}_i \neq \bot$ in the post-state. But this would cause the post-state to violate Invariant A.1, a contradiction.

We prove part 3 by induction, using part 1. This time, the interesting steps are $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ and $\texttt{net\_rcv}_i\texttt{(j,r,*,*)}$. Again, the $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ step yields the property immediately. A $\texttt{net\_rcv}_i\texttt{(j,r,v',*)}$ could only falsify the property if $j \in \texttt{W}$. But in this case we know that $\texttt{value[r,j]}_i \neq \bot$ i the pre-state, and then Invariant A.2 implies that $\texttt{v'} = \texttt{value[r,j]}_i$ in the pre-state. It follows that this step does not change $\texttt{value[r,j]}_i$, and so does not falsify the property.

Part 4 is proved by induction on r (not induction on the length of the execution), using part 2. The base case, $\texttt{r} = 1$, follows immediately from part 2. For the inductive step, we suppose that the claim is true for some $\texttt{r} \geq 1$ and show it for $\texttt{r + 1}$. That is, we assume that (i,r+2,v,W) is in the Net. Then by part 2, $\texttt{W} = \texttt{world[r+1,i]}_i \setminus \texttt{out[r+1]}_i \setminus \texttt{failed[r+1]}_i$. Now, since $\texttt{world[r+1,i]}_i \neq \bot$, Invariant A.1 implies that a message of the form (i,r+1,v',W') is in the Net, where $\texttt{W'} = \texttt{world[r+1,i]}_i$. By inductive hypothesis, part 4, this implies that $\texttt{W'} = \texttt{world[1,i]}_i \setminus \texttt{out-by[r]}_i \setminus \texttt{failed-by[r]}_i$. Therefore, $\texttt{W} = \texttt{world[1,i]}_i \setminus \texttt{out-by[r]}_i \setminus \texttt{failed-by[r]}_i \setminus \texttt{out[r+1]}_i \setminus \texttt{failed[r+1]}_i$. This is equal to $\texttt{world[1,i]}_i \setminus \texttt{out-by[r+1]}_i \setminus \texttt{failed-by[r+1]}_i$, as needed. ∎

Invariant A.8 *Suppose that* $\texttt{decide}_i\texttt{(v)}$ *has happened at round* r. *Then:*

1. *For all* $j \in$ `world[r,i]`$_i$ $\setminus$ `out[r]`$_i$, `value[r,j]`$_i =$ `value[r,i]`$_i$ *and* `world[r,j]`$_i \subseteq$ `world[r,i]`$_i$.

2. *For all* $j \in$ `world[r,i]`$_i$, *if* `(j,r,v',W)` *is in the Net then* `v'` $=$ `v` *and* `W` $\subseteq$ `world[r,i]`$_i$.

Proof: Part 1 follows from an easy induction: `out` can only grow, and `value` and `world` do not change once they are non-$\perp$. Therefore, the only interesting step is `decide`$_i$`(v)`, and the result follows directly from the precondition.

For part 2, consider any state s after a `decide`$_i$`(v)` has happened at round `r`. Suppose that $j \in$ `world[r,i]`$_i$ and `(j,r,v',W)` is in the Net, in state s. Then Invariant A.4 implies that $j \notin$ `out[r]`$_i$. Thus, $j \in$ `world[r,i]`$_i \setminus$ `out[r]`$_i$. Then part 1 and Invariant A.2 imply the conclusion. ∎

The following invariants say that any process' values and worlds decrease as rounds increase.

Invariant A.9 *For any* `r` ≥ 1, *if a message* `(i,r+1,v,W)` *is in the Net then* `value[r,i]`$_i \neq \perp$, `v` $\leq$ `value[r,i]`$_i$, *and* `W` $\subseteq$ `world[r,i]`$_i$.

Proof: By induction. For the inductive step, the interesting case is when the last action of the execution is `net_mcast(i,r+1,v,W)`. Invariant A.6 implies that $i \in$ `W`. Therefore, the precondition for `net_mcast(i,r+1,v,W)` implies that, in the pre-state, `value[r,i]`$_i \neq \perp$. Therefore, this is also true in the post-state, as needed.

Next, we show that `v` $\leq$ `value[r,i]`$_i$. The value `v` is determined in the `net_mcast` event to be the minimum of the set of values of the form `value[r,j]`$_i$, for $j \in$ `W`. Since $i \in$ `W`, this minimum includes `value[r,i]`$_i$. Therefore, `v` $\leq$ `value[r,i]`$_i$.

Finally, we show that `W` $\subseteq$ `world[r,i]`$_i$. The value `W` is determined in the `net_mcast` event to be `world[r,i]`$_i$ $\setminus$ `out-by[r]` $\setminus$ `failed-by[r]`, according to the values of the out and failed sets in the pre-state. It follows immediately that `W` is a subset of `world[r,i]`$_i$. ∎

Invariant A.10 *For any* $r \geq 1$,

1. *If* `value[r+1,i]`$_i \neq \perp$ *then* `value[r,i]`$_i \neq \perp$, `value[r+1,i]`$_i \leq$ `value[r,i]`$_i$, *and* `world[r+1,i]`$_i \subseteq$ `world[r,i]`$_i$.

2. *If* `value[r,i]`$_i \neq \perp$ *and* `1` $\leq$ `s` $\leq$ `r` *then* `value[s,i]`$_i \neq \perp$, `value[r,i]`$_i \leq$ `value[s,i]`$_i$, *and* `world[r,i]`$_i \subseteq$ `world[s,i]`$_i$.

Proof: For part 1, assume that, in some reachable state, `value[r+1,i]`$_i \neq \perp$, and hence `world[r+1,i]`$_i$ $\neq \perp$. Then Invariant A.1 implies that in the same state, a message `(i,r+1,v,W)` must be in the Net, where `v` $=$ `value[r+1,i]`$_i$ and `W` $=$ `world[r+1,i]`$_i$. Invariant A.9 then yields the conclusions. Part 2 follows from part 1, using induction on `r-s`. ∎

The following invariant says that, if all the messages for a particular round `r` are "consistent", then so are all the messages for all later rounds.

Invariant A.11 *Let* `W` *be a nonempty finite set,* `v` $\in$ `V`, *and* `r` ≥ 1. *Suppose that, for every* $i \in$ `W`, *if a message of the form* `(i,r,v',W')` *is in the Net, then* `W'` $\subseteq$ `W` *and* `v'` `=` `v`.
Then for every $i, j \in$ `W` *and for every* `s` $\geq$ `r`,

1. *If a message of the form* `(i,s,v',W')` *is in the Net, then* `W'` $\subseteq$ `W` *and* `v` `=` `v'`.

2. `value[s,i]`$_j$ *is either* `v` *or* $\perp$.

3. $\texttt{world[s,i]}_j$ *is either a subset of* `W` *or* $\perp$.

Proof: We prove part 1 by induction on the length of an execution.
Base: The conclusion of the invariant is vacuously true in the start state.
Inductive step: The interesting steps are those that put some message `(i,s,v',W')` into the Net, where $i \in$ `W` and `s` $\geq$ `r`. We may restrict attention to the case where `s > r`, because if `s` $=$ `r` and the step falsifies part 1, it also falsifies the hypothesis of the invariant. Thus, the only interesting steps are of the form `net_mcast(i,s,v',W')` where $i \in$ `W` and `s` $>$ `r`. So consider such a step, and fix i, `s`, `v'`, and `W'`. Assume that the hypothesis of the invariant is true after (and hence before) the step.

We show that `W'` $\subseteq$ `W`. After the `net_mcast` step, the message is in the Net. Invariant A.9 then implies that `W'` $\subseteq$ $\texttt{world[s-1,i]}_i$. Invariant A.10 then implies that $\texttt{world[r,i]}_i \neq \perp$ and $\texttt{world[s-1,i]}_i \subseteq \texttt{world[r,i]}_i$. Therefore, `W'` $\subseteq$ $\texttt{world[r,i]}_i$. Since $\texttt{world[r,i]}_i \neq \perp$, an `(i,r,*,W'')` message is in the Net, where `W''` = $\texttt{world[r,i]}_i$. Then since the hypothesis of the invariant is true, it follows that $\texttt{world[r,i]}_i \subseteq$ `W`. Putting all the pieces together yields that `W'` $\subseteq$ `W`.

Next, we argue that `v' = v`. The value `v'` is determined by the precondition of the `net_mcast` action, as the minimum of a set of values $\texttt{value[s-1,j]}_i$, taken over all indices j in `W'`. Because `W'` $\subseteq$ `W`, every such index j is in `W`. Since $\texttt{value[s-1,j]}_i \neq \perp$, a `(j,s-1,v'',W'')` message is in the Net in the pre-state of the new `net_mcast`, where `v''` = $\texttt{value[s-1,j]}_i$. Our assumption that the conclusion of the invariant is true in the pre-state then implies that $\texttt{value[s-1,j]}_i =$ `v`. Thus, all the values considered in the min are equal to `v`, which implies that `v = v'`.

This proves part 1. Parts 2 and 3 follow from part 1 and Invariant A.1. ∎

Invariant A.12 *If* $\texttt{round}_i$ `= r > 1` *and* $\texttt{mode}_j$ `= running` *and j is not failed, then* $\texttt{round}_j \geq$ `r - 1`.

Proof: Since j is not failed, by Invariant A.1(6), $j \notin \texttt{failed[s]}_i$ for any `s`, so $j \notin \texttt{failed-by[r-1]}_i$. By Invariant A.3(2), $j \notin \texttt{out[s]}_i$ for any `s`, so $j \notin \texttt{out-by[r-1]}_i$. Since $\texttt{mode}_j$ `= running`, j initiated, and by the world consistency assumption, $j \in \texttt{world[1,i]}_i$. By Invariant A.7(4), j is in i's world for round `r`. Therefore, i must have received a round `r-1` message from j before moving to round `r`. ∎

A.1.2 CUP safety guarantees

We now prove that the CUP implementation satisfies the CUP safety guarantees, assuming the environment satisfies the safety assumptions.

Theorem A.1 *The CUP algorithm satisfies well-formedness.*

Proof: This is straightforward from the code and the well-formedness assumptions on the environment. For condition 1, assume that $\texttt{decide}_i$ occurs. Then in the preceding state, `mode = running`. `mode` is initially $\perp$, and the only way it becomes `running` is via $\texttt{init}_i$. So there must be a preceding $\texttt{init}_i$.

For condition 2, assume for the sake of contradiction that two $\texttt{decide}_i$ events occur. Part 1 implies that an $\texttt{init}_i$ precedes the first $\texttt{decide}_i$. The first $\texttt{decide}_i$ sets $\texttt{mode}_i$ to `done`. After this point, and before the second $\texttt{decide}_i$ event occurs, $\texttt{mode}_i$ must become `running`. This can happen only as a result of another $\texttt{init}_i$ event. This means that two $\texttt{init}_i$ events must occur, which contradicts the environment well-formedness assumption. Therefore, no more than one $\texttt{decide}_i$ event occurs. ∎

Theorem A.2 *The CUP algorithm satisfies uniform agreement.*

Proof: If at most one `decide` event occurs, the result follows immediately. So assume that there are at least two decide events. Consider the first decide event, $\texttt{decide}_i\texttt{(v)}$. By the precondition, we know that in the pre-state, there exists `r` such that $\texttt{world[r,i]}_i \neq \bot$ and $\forall j \in \texttt{world[r,i]}_i \setminus \texttt{out[r]}_i$, $\texttt{value[r,j]}_i = \texttt{v}$ and $\texttt{world[r,j]}_i \subseteq \texttt{world[r,i]}_i$. Since i does not leave, abstain, or decide before the $\texttt{decide}_i$ event, we know that $i \notin \texttt{out[r]}_i$ in the pre-state; therefore, $\texttt{value[r,i]}_i = \texttt{v}$. Also consider any particular later decide event, $\texttt{decide}_{i'}\texttt{(v')}$. As above, we know that in the pre-state of this event, there exists `r'` such that $\texttt{world[r',i']}_{i'} \neq \bot$ and $\forall j \in \texttt{world[r',i']}_{i'} \setminus \texttt{out[r']}_{i'}$, $\texttt{value[r',j]}_{i'} = \texttt{v'}$ and $\texttt{world[r',j]}_{i'} \subseteq \texttt{world[r',i']}_{i'}$. Moreover, $\texttt{value[r',i']}_{i'} = \texttt{v'}$.

We now show that $i' \in \texttt{world[r,i]}_i$. Since i' decides, it initiates and does not leave or fail before it decides. Since i initiates before it decides, and thus before i' decides, i' does not leave or fail before i initiates. Then the world consistency assumption implies that i' gets put into $\texttt{world[1,i]}_i$. If $\texttt{r} = 1$ then we are done, so assume that $\texttt{r} \geq 2$. Then the value of $\texttt{world[r,i]}_i$ is determined in a $\texttt{net\_mcast}_i\texttt{(i,r,*,*)}$ step. To see that i' is included in $\texttt{world[r,i]}_i$, note that that set is defined in the $\texttt{net\_mcast}_i\texttt{(i,r,*,*)}$ step to include (at least) all processes in $\texttt{world[1,i]}_i$ that do not leave, abstain, decide, or fail before the $\texttt{net\_mcast}_i\texttt{(i,r,*,*)}$ event. And i' does not leave, abstain, decide, or fail by then, because this $\texttt{net\_mcast}_i\texttt{(i,r,*,*)}$ event happens prior to the $\texttt{decide}_i$.

We also know that $i' \notin \texttt{out[r]}_i$ in the pre-state of $\texttt{decide}_i$. This is because i' has not left, abstained, or decided before the $\texttt{decide}_i$.

Next, we show that $\texttt{r'} \geq \texttt{r}$, that is, the round at which i' decides is at least as great as the round at which i decides. Since $i' \in \texttt{world[r,i]}_i$ and $i' \notin \texttt{out[r]}_i$ in the pre-state of $\texttt{decide}_i$, the precondition for $\texttt{decide}_i$ implies that $\texttt{value[r,i']}_i \neq \bot$ in the pre-state of $\texttt{decide}_i$. This means that i' must send an `(i',r,v,*)` message. This implies that the round `r'` at which i' decides is at least as great as `r`, that is, $\texttt{r'} \geq \texttt{r}$.

Finally, we argue that `v' = v`. Invariant A.8, part 2, implies that in the pre-state of $\texttt{decide}_{i'}$, if $j \in \texttt{world[r,i]}_i$ and if `(j,r,v'',W'')` is in the Net, then $\texttt{v''} = \texttt{value[r,i]}_i$ and $\texttt{W''} \subseteq \texttt{world[r,i]}_i$. Since $\texttt{r'} \geq \texttt{r}$ and $i' \in \texttt{world[r,i]}_i$, Invariant A.11, part 2, implies that in the pre-state of $\texttt{decide}_{i'}$, $\texttt{value[r',i']}_{i'}$ is either `v` or $\bot$. Since (as noted earlier) $\texttt{value[r',i']}_{i'} = \texttt{v'}$, we have that `v=v'`.

■

Theorem A.3 *The CUP algorithm satisfies validity.*

Proof: Part 1 follows from Invariant A.5. Part 2 follows from Invariant A.10. ■

A.1.3 CUP liveness guarantees

We now show that CUP satisfies its liveness property—termination. Formally, the lemmas and theorem we state in this section should be interpreted with respect to an execution α of the composition of automata CUP_i for $i \in I$ such that:

1. All the environment safety and liveness assumptions are satisfied in α.

2. α is "weakly fair" to all actions of all CUP_i automata, in the sense that if an action is enabled from some point onward, it eventually is performed.

Lemma A.4 *Let* J *be the set of processes that initiate and never decide, leave, or fail, and suppose that* $i \in$ J*. If* $\texttt{init}_i\texttt{(v, W)}$ *occurs and* $j \in$ W *then either* $j \in$ J *or else* j *abstains, leaves, decides, or fails.*

Proof: Follows from the init occurrence assumption. ■

Lemma A.5 *If process* i *initiates and never decides, leaves, or fails, then* $\texttt{round}_i$ *increases without bound.*

Proof: Let J be the set of all processes that initiate and never decide, leave, or fail. Assume for the sake of contradiction that, for some process $i \in J$, $\texttt{round}_i$ is bounded. Let r be the smallest round number such that for some process $i \in$ J, $\texttt{round}_i$ is bounded by r, and fix such $i \in$ J. Process i cannot get stuck at round 0, because the $\texttt{init}_i$ action immediately increments the round to 1. So we may assume that r > 0.

We argue that i cannot be stuck at round r, by showing that for some v, W, the $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ action is eventually enabled and stays enabled. Then weak fairness implies that $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ eventually occurs.

We claim that the last precondition of $\texttt{net\_mcast}_i\texttt{(i,r+1,*,*)}$ (the negation of the decide precondition) is always true. For if not, then $\texttt{decide}_i\texttt{(v)}$ would be enabled for some v, and would stay enabled forever. This implies, by weak fairness, that $\texttt{decide}_i$ occurs, a contradiction.

Next, we claim that for every $j \in$ world[1,i], either i receives a round r message from j, or else i puts j into its failed[r'] set or out[r'] set for some r' $\leq$ r. Fix any such j. Lemma A.4 implies that either $j \in$ J or j eventually abstains, leaves, decides, or fails. If $j \in$ J then by choice of r, j does not get stuck at any round less than r, and so j eventually sends a round r message, which i eventually receives.

If j fails, then eventually a $\texttt{fail\_detect}_i\texttt{(j)}$ occurs, which makes i put j into one of its failed[r'] sets. If r' $\leq$ r then we are done; on the other hand, if r' $>$ r then i receives a round r message from j.

If j abstains and does not fail, then eventually i puts j into its out[1] set (which suffices because $1 \leq$ r). If j leaves or decides at a round r' $\leq$ r, then eventually i puts j into its out[r'] set. Finally, if j leaves or decides at a round r' $>$ r, then eventually i receives a round r message from j.

This claim implies that eventually the precondition of $\texttt{net\_mcast}_i\texttt{(i,r+1,v,W)}$ is satisfied for some v, W. Because the values and worlds can only decrease, eventually the precondition is satisfied, and remains satisfied, for the same v, W. Then weak fairness implies that the action eventually occurs, which moves j to round r + 1. This is a contradiction. ■

Lemma A.6 *Let* J *be the set of processes that initiate and never decide, leave, or fail, and suppose that* $i \in$ J*. Then for* r *sufficiently large,* $\texttt{world[r,i]}_i$ = J.

Proof: The result follows from two claims: that for all r, J $\subseteq$ $\texttt{world[r,i]}_i$, and that for sufficiently large r, $\texttt{world[r,i]}_i$ is a subset of J.

First, we show that for all r, J $\subseteq$ $\texttt{world[r,i]}_i$. World consistency implies that J $\subseteq$ $\texttt{world[1,i]}_i$. Since no element of J ever abstains, leaves, fails, or decides, no element of J is ever put into any $\texttt{failed[r]}_i$ or $\texttt{out[r]}_i$. Then the definition of $\texttt{world[r,i]}_i$ (in net_mcast(i,r,*,*)) implies that for all r, J $\subseteq$ $\texttt{world[r,i]}_i$.

Second, we show that for sufficiently large r, $\texttt{world[r,i]}_i$ is a subset of J. Let j be any element of $\texttt{world[r,i]}_i$. Lemma A.4 implies that if $j \notin$ J, then j eventually abstains, leaves, decides, or fails. But in any of these cases, j eventually gets put into some `failed[r]`$_i$ or `out[r]`$_i$. This means that j is excluded from `world[r,i]`$_i$ for sufficiently large `r`. ∎

Theorem A.7 *The CUP algorithm satisfies termination.*

Proof: We prove that every process that initiates eventually decides, leaves, or fails. Assume for the sake of contradiction that there is at least one initiator that does not decide, leave, or fail. Let `J` be the set of processes that initiate and never decide, leave, or fail; then `J` is not empty. Then Lemma A.5 implies that the rounds of all processes in `J` increase without bound, and Lemma A.6 implies that for sufficiently large `r`, `world[r,i]`$_i$ `= J` for all $i \in$ `J`. Thus from some round onward, every process in `J` bases its new value on values heard from exactly the members of `J`.

Thereafter, each $i \in$ `J` eventually reaches some minimum value of `value[r,i]`$_i$ (by monotonicity and the fact that only finitely many values can be used). Consider a round beyond which all the minima have been attained. If these are all identical, then all processes can decide based on this value and world `J`, and we are done. On the other hand, if they are not all identical, then let i be a process whose minimum is larger than some other process' minimum. Then i would see a smaller value and reduce its value further, a contradiction. ∎

A.2 Atom Correctness Proof: Safety Arguments

A.2.1 General Invariants

The following invariants follow immediately from the code:

Invariant A.13 *If* `join-slot`$_i \neq \infty$ *then* `leave-slot`$_i$ `>` `join-slot`$_i$.

Invariant A.14 *Suppose* $s \in$ `ended-slots`$_i$. *Then:*

1. *If* $j \in$ `joiners[s]`$_i$ *then* `join-slot`$_j$ `= s`.
2. *If* $j \in$ `leavers[s]`$_i$ *then* `leave-slot`$_j$ `= s`.

Proof: Process j can be inserted into `joiners[s]`$_i$ (`leavers[s]`$_i$) only if i receives a `(j, JOIN, s-1)` (resp. `(j, LEAVE, s)`) message, which can be sent only by j and only if `join-slot`$_j$ `= s` (resp. `leave-slot`$_j$ `= s`). ∎

The following invariant asserts that from the join slot onward, slot messages (bulk, join, or leave) are multicast in order.

Invariant A.15 *If* `join-slot`$_i - 1 \leq$ `s'` $\leq$ `s` *and* `s` $\in$ `mcast-slots`$_i$ *then* `s'` $\in$ `mcast-slots`$_i$.

Proof: `join-slot`$_i$ had to have been set before `current-slot`$_i$ becomes `s'+1` because it is always chosen to be in the future. Therefore, `net_mcast`$_i$`(i, *, s')` is enabled once `current-slot`$_i$ becomes `s'+1`. This is earlier than the time at which `net_mcast`$_i$`(i, *, s)` can occur, so time could not have passed beyond that point without `net_mcast`$_i$`(i, *, s-1)` occurring. ∎

The following invariant is central to the rest of the proof. It asserts that by the time of `end_slot`$_i$`(s)`, i has all the right processes in `alive[s-1]`, `alive[s]`, `joiners[s]`, and `joiners[s+1]`.

Invariant A.16 *If* $s \in$ `ended-slots`$_i$ *then*

1. *If* `join-slot`$_j \leq$ `s` *and* `s` $\in$ `mcast-slots`$_j$ *then* $j \in$ `alive[s-1]`$_i\cup$ `joiners[s]`$_i$.

2. *If* `join-slot`$_j \leq$ `s+1` *and* `s+1` $\in$ `mcast-slots`$_j$ *then* $j \in$ `alive[s]`$_i\cup$ `joiners[s+1]`.

Proof: If j joined by slot `s`, it registered for the network before starting slot `s-1`. Moreover, if `s` $\in$ `mcast-slots`$_j$, then by Invariant A.15, `s-1` $\in$ `mcast-slots`$_j$, and therefore j multicasts either a `(j, JOIN, s-1)` or a bulk message in slot `s-1`, and by the Net's *reliable delivery* property, this message is not lost due to j's failure because j multicasts a message in the following slot, which occurs Θ time later, and we assume that $\Theta > \Delta$, and messages sent more than Δ time before the failure are not lost. Likewise, if `join-slot`$_j \leq$ `s+1` and `s+1` $\in$ `mcast-slots`$_j$, then `s` $\in$ `mcast-slots`$_j$ (by Invariant A.15), and j multicasts a bulk or join message in slot `s`, which is not lost due to j's failure.

We will now show two things: first, that i joined early enough to get j's slot `s-1` bulk or join message; and second, that `end_slot`$_i$`(s)` occurred late enough for i to have received j's slot `s` bulk or join message.

Since i does `end_slot` for `s`, `join-slot`$_i \leq$ `s`. Process i chooses its `join-slot` following the `net_join_OK`$_i$ to be `current-slot`$_i$ + 2 + $\lceil \Gamma/\Theta \rceil$, so `current-slot`$_i$ becomes `s-1` at least Γ time after the `net_join_OK`$_i$. Since the maximum clock difference between i and j is Γ, j sends its message (join or bulk) for slot `s-1` no earlier than the time of the `net_join_OK`$_i$, so i joined early enough to get j's message for slot `s`.

It is left to show that i gets j's slot `s` bulk or join message for slot `s` before `end_slot`$_i$`(s)`. This follows from the precondition for `end_slot`$_i$ which asserts that `clock`$_i$ $> (s+1)\Theta + \Delta + \Gamma$. That is, that at least $\Delta + \Gamma$ time has elapsed since slot `s+1` has begun at i. Since the clock difference between i and j is at most Γ, we get that at least Δ time has elapsed since slot `s+1` has begun at j. Since j sends its slot `s` message once slot `s+1` begins at j, and the network latency is bounded by Δ, the message reaches i before `end_slot`$_i$`(s)`. ■

The following invariants are related to the `suspects[s]` sets.

Invariant A.17 *If* `suspects[s]`$_i$ *is not empty, then* `join-slot`$_i \leq$ `s`.

Proof: `suspects[s]`$_i$ gets set only upon `end_slot`$_i$`(s)`, for which this is a precondition. Once `join-slot`$_i$ is set to a non-∞ value, it does not change, by the singularity of `join` and `net_join_OK`. ■

Invariant A.18 *If* $j \in$ `suspects[s]`$_i$ *then* j *has failed.*

Proof: Since j gets inserted to `suspects[s]`$_i$ during `end_slot`$_i$`(s)`, j is in (`alive[s-1]`$_i$ $\cup$ `joiners[s]`$_i$ $\setminus$ `leavers[s]`$_i$) $\setminus$ `alive[s]`$_i$). In particular, j is in `alive[s-1]`$_i\cup$ `joiners[s]`$_i$, and therefore `join-slot`$_j \leq$ `s`. Moreover, j is not in `alive[s]`$_i$, so by the contrapositive of Invariant A.16(2), `s+1` $\notin$ `mcast-slots`$_j$, which implies that j either fails or leaves before sending a slot `s+1` message. Since j is also not in `leavers[s]`$_i$, j must have failed. ■

Invariant A.19 *If* $j \in$ `suspects[s]`$_i$ *and* $j \in$ `suspects[s']`$_{i'}$ *then* `|s' - s|` ≤ 1.

Proof: Without loss of generality, assume `s'` $\geq$ `s`. Since $j \in$ `suspects[s]`$_i$, then $j \in$`alive[s-1]`$_i \cup$ `joiners[s]`$_i$, and therefore `join-slot`$_j \leq$ `s`. Moreover, j is not in `alive[s]`$_i$ when `s` $\in$ `ended-slots`$_i$, so by the contrapositive of Invariant A.16(2), `s+1` $\notin$ `mcast-slots`$_j$. By Invariant A.15, for any slot `r > s`, `r` $\notin$ `mcast-slots`$_j$, and therefore $j \notin$ `alive[r]`$_{i'}$ for any `r > s`. Since i' suspects j in slot `s'`, j is in `alive[s'-1]`$_{i'}$, and therefore `s'-1` $\leq$ `s`. ■

The following invariant states that a process does not abstain from CUP instances pertaining to processes that it suspects.

Invariant A.20 *If* $k \in$ `suspects[s]`$_i$ *and* `CUP-status[k]`$_i$ `= done` *then* `CUP-dec-val[k]`$_i \neq \perp$.

Proof: Assume by contradiction that the invariant is false. Since `CUP-status[k]`$_i$ `= done` while `CUP-dec-val[k]`$_i = \perp$, then i must have performed `CUP(k).abstain`$_i$. By the precondition for `CUP(k).abstain`$_i$, `CUP-status[k]`$_i$ was `req` when `abstain`$_i$ occurred, which implies that `end_slot(s)`$_i$ could not have already occurred, that is, all the slots in `ended-slots`$_i$ were smaller than `s` at the time of `CUP(k).abstain`$_i$. By the precondition for `CUP(k).abstain`$_i$, when it occurred, `CUP-req-val[k]`$_i$ had some non-$\perp$ value, `v`, such that `v < s-1`. This, in turn, implies that a (CUP-INIT, k) message with slot v had previously arrived. That means that such a message was previously sent by some j, which implies that k is added to `suspects[v]`$_j$, during `end_slot`$_j$`(v)`, and remains there henceforward. But $k \in$ `suspects[s]`$_i$ and `v < s-1`, a contradiction to Invariant A.19. ■

Invariant A.21 *If* $k \in$ `alive[s]`$_i$, $k \notin$ `alive[s]`$_j$, *and* `s` $\in$ `ended-slots`$_j$ *then* $k \in$ `suspects[s]`$_j$. *Moreover, if* `s+1` $\in$ `ended-slots`$_i$ *then* $k \in$ `suspects[s+1]`$_i$.

Proof: Since $k \in$ `alive[s]`$_i$, we know that `join-slot`$_k \leq$ `s < leave-slot`$_k$ and that `s` $\in$ `mcast-slots`$_k$. Therefore, by Invariant A.16(1), if `s` $\in$ `ended-slots`$_j$ then $k \in$ `alive[s-1]`$_j \cup$ `joiners[s]`$_j$. Additionally, k is neither in `leavers[s]`$_i$ nor in `leavers[s]`$_j$, because it does not leave at slot `s`. Therefore, since $k \notin$ `alive[s]`$_j$, in `end_slot`$_j$`(s)`, k gets inserted into `suspects[s]`$_j$.

Since $k \notin$ `alive[s]`$_j$, by the contrapositive of Invariant A.16(2), we get that `s+1` $\notin$ `mcast-slots`$_k$. That is, k does not send a bulk or leave message for slot `s+1`. Therefore, $k \notin$ `alive[s+1]`$_i \cup$ `leavers[s+1]`$_i$ when `end_slot`$_i$`(s+1) occurs`, and k gets inserted into `suspects[s+1]`$_i$ when `s+1` is inserted to `ended-slots`$_i$. ■

Invariant A.22 *If* $k \in$ `alive[s]`$_i$ *and* `s+1` $\in$ `ended-slots`$_i$ *and* `CUP-dec-val[k]`$_j \leq$ `s` *then* $k \in$ `suspects[s+1]`$_i$.

Proof: Since $k \in$ `alive[s]`$_i$, `join-slot`$_k \leq$ `s < leave-slot`$_k$. Since `CUP-dec-val[k]`$_j \leq$, then by the validity property of CUP, some process l must have initiated `CUP(k)` with an initial value `s'` $\leq$ `s`. This implies that $k \in$ `suspects[s']`$_l$, and therefore $k \notin$ `alive[s']`$_l$ and `s'` $\in$ `ended-slots`$_l$. By contrapositive of Invariant A.16, `s'+1` $\notin$ `mcast-slots`$_k$, and therefore also `s'+1` $\notin$ `mcast-slots`$_k$. So i does not hear a bulk or leave message from k for slot `s+1`, and $k \in$ `suspects[s+1]`$_i$. ■

Lemma A.8 *Assume that for some processes* j, k, l `CUP(k).init`$_l$`(v, W)` *occurs with* $j \in$`W`, *and that* `CUP(k).init`$_i$`(v', W')` *also occurs. Then* `net_join_OK`$_j$ *has occurred before* `CUP(k).init`$_i$`(v', W')`.

Proof: By the precondition for $\texttt{CUP(k).init}$, $k \in \texttt{suspects[v']}_i$ and $k \in \texttt{suspects[v]}_l$, so by Invariant A.19, $v' \geq v - 1$. When $\texttt{CUP(k).init}_l\texttt{(v, W)}$ occurs, $\texttt{W = alive[v]}_l \cup \texttt{ joiners[v+1]}_l$. Since $j \in$W, this implies that $\texttt{join-slot}_j \leq \texttt{v+1} \leq \texttt{v'+2}$. Assume $\texttt{CUP(k).init}_i\texttt{(v', W')}$ occurs at time t. So at time t, $\texttt{v'} \in \texttt{ended-slots}_i$. By the precondition for $\texttt{end\_slot}_i\texttt{(v')}$, at time t $\texttt{clock}_i > (v'+1)\Theta + \Delta + \Gamma$. Since $\texttt{v'+1} \geq \texttt{join-slot}_j - 1$, at this time $\texttt{clock}_i > \texttt{(join-slot}_j - 1)\Theta + \Delta + \Gamma$. Since the clock skew is bounded by Γ, at time t, $\texttt{clock}_j > \texttt{(join-slot}_j - 1)\Theta + \Delta$. So t is at least Δ time after j begins slot $\texttt{join-slot}_j - 1$. But $\texttt{join-slot}_j$ is chosen to be at least 2 slots after the slot at which $\texttt{net\_join\_OK}_j$ occurs at j, so j begins $\texttt{join-slot}_j - 1$ after the $\texttt{net\_join\_OK}_j$, i.e., before time t. ∎

A.2.2 Safety environment conditions for CUP

Well-formedness $\texttt{CUP(k).init}_i$ only occurs when k becomes suspected at i. Once k is suspected, it is never again alive. Therefore, it is never suspected again and $\texttt{CUP(k).init}_i$ occurs at most once. By Invariant A.20, since k is suspected at i, i does not abstain. Thus, at most one $\texttt{init}_i$ or $\texttt{abstain}_i$ event occurs.

The fact that at most one $\texttt{leave}_i$ event occurs and at most one $\texttt{fail}_i$ event occurs is ensured by the application, since `leave` and `fail` actions are routed directly from the application to all instances of CUP.

The fact that no $\texttt{fail}_i$ precedes an $\texttt{init}_i$ follows from the fact that failures affect all components and processes do not take any steps after they fail.

One of the preconditions $\texttt{end\_slot}_i$ is that $\texttt{leave-slot}_i = \infty$, that is, that $\texttt{leave}_i$ did not occur. Therefore, no $\texttt{leave}_i$ precedes an $\texttt{init}_i$.

World consistency Assume that $\texttt{CUP(k).init}_i\texttt{(s, W)}$ occurs at time t, j does not leave or fail before time t, and $\texttt{CUP(k).init}_j\texttt{(s', *)}$ also occurs. We need to show that $j \in$W.

$\texttt{CUP(k).init}_i\texttt{(s, W)}$ is triggered during $\texttt{end\_slot}_i\texttt{(s)}$. We need to show that at this time $j \in \texttt{alive[s]}_i \cup \texttt{ joiners[s+1]}_i$. This is true if i receives j's slot `s` bulk or join message.

By the precondition for $\texttt{end\_slot}_i\texttt{(s)}$, $\texttt{clock}_i > \texttt{(s+1)}\Theta + \Delta + \Gamma$ at time t. Since the difference between a process clock and real time is at most $\Gamma/2$, the real time associated with point t is at least $\texttt{(s+1)}\Theta + \Delta + \Gamma/2$. By assumption, j does not fail or leave until this time.

By Invariant A.19, $\texttt{s'} \leq \texttt{s+1}$. When $\texttt{CUP(k).init}_j\texttt{(s',*)}$ occurs, $k \in \texttt{suspects[s']}_j$. By Invariant A.17, $\texttt{join-slot}_j \leq \texttt{s'}$. Together these two inequalities imply that $\texttt{join-slot}_j \leq \texttt{s+1}$. Therefore, if j does not fail or leave before $\texttt{clock}_j$ becomes $\texttt{s+1}\Theta$, j multicasts its slot `s` bulk or join message when $\texttt{clock}_j = \texttt{(s+1)}\Theta$ (a join message is multicast if $\texttt{join-slot}_j = \texttt{s+1}$; otherwise j multicasts a slot `s` bulk message). When $\texttt{clock}_j = \texttt{(s+1)}\Theta$, the real time is at most $\texttt{(s+1)}\Theta + \Gamma/2$. If j does not fail until the real time becomes $\texttt{(s+1)}\Theta + \Gamma/2 + \Delta$, then j's message is not lost, and i receives it by time $\texttt{(s+1)}\Theta + \Gamma/2 + \Delta$. But we assume that j does not fail or leave until this time.

Accurate failure detector $\texttt{CUP(k).fail\_detect}_i\texttt{(j)}$ occurs only if for some slot `s` $j \in \texttt{suspects[s]}_i$. Therefore, by Invariant A.18, j has previously failed. Moreover, since a process is never again alive after it is suspected, it is never again suspected, and $\texttt{CUP(k).fail\_detect}_i\texttt{(j)}$ does not recur.

Accurate leave detector $\texttt{CUP(k).leave\_detect}_i\texttt{(j)}$ occurs only if a LEAVE message is received from j; j sends at most one such a message and only if it actually leaves.

Lossless leave Assume a CUP process at j multicasts a message m, and subsequently, $\texttt{leave}_j$ occurs. When $\texttt{leave}_j$ occurs, a LEAVE message is inserted to $\texttt{out-buf}_j$ to be sent in the ensuing slot. This LEAVE message is multicast after m. $\texttt{leave\_detect}_i\texttt{(j)}$ occurs when this LEAVE message is received. By the FIFO property of Net, $\texttt{net\_rcv}_j\texttt{(m)}$ occurs beforehand.

A.2.3 Proving the total order property

We now prove that all the process deliver messages in a consistent total order. We define the total order S as follows: Let P_s be the union of all sets P such that an action $\texttt{members(P, s)}_i$ occurs. The set of messages S_s is defined to be those messages included in slot s bulk messages by processes in P_s. The set of messages in S is defined to be the union of all sets S_s.

The ordering is based on slots, so that for s < s', all messages in S_s precede all messages in S'_s. For messages pertaining to the same set S_s, the ordering is by process indices. For the same slot and process index, the ordering is the temporal order of sending (at the external boundary of Atom).

We have to show that every process delivers a contiguous subsequence of S. We first prove Lemma A.9, asserting that every two processes that perform a members(P, s) action for a slot s do so with the same membership set P. As part of this action, processes deliver messages for slot s. Next, we prove Lemma A.10, asserting that if a process i performs $\texttt{members}_i\texttt{(P, s)}$ with $j \in$P, then i has received a bulk message for slot s from j, and therefore triggers the delivery of all the messages included in it as an effect of the $\texttt{members}_i\texttt{(P, s)}$ action. Thus, every process that performs members(P, s), triggers the delivery of all the messages in S_s. These messages are delivered in order of the sender's process index, and for each process, in FIFO order. Therefore, these messages are delivered in the order defined on S_s.

Since every process performs members(P, s) for a contiguous subsequence of slots, every process delivers a contiguous subsequence of the messages in S.

We now prove the lemmas:

Lemma A.9 *If* $\texttt{members(P,s)}_i$ *and* $\texttt{members(P',s)}_j$ *occur, then* P= P'.

Proof: Let k be a process in P. At the time $\texttt{members(P,s)}_i$ occurs, $k \in \texttt{alive[s]}_i$, $\texttt{s+1} \in \texttt{ended-slots}_i$, and $\texttt{CUP-dec-val[k]}_i$ is either $\perp$ or larger than s. Assume by way of contradiction that $k \notin$ P', then either $k \notin \texttt{alive[s]}_j$ or $\texttt{CUP-dec-val[k]}_j \leq$ s when $\texttt{members(P',s)}_j$ occurs.

Assume first that $k \notin \texttt{alive[s]}_j$. Note that s $\in \texttt{ended-slots}_j$ when $\texttt{members(P',s)}_j$ occurs, so by Invariant A.21, $k \in \texttt{suspects[s]}_j$ at the time $\texttt{members(P',s)}_j$ occurs. By the precondition for $\texttt{members(P',s)}_j$, $\texttt{CUP-status[k]}_j$ = done when it occurs, and by Invariant A.20, $\texttt{CUP-dec-val[k]}_j \neq \perp$, that is, $\texttt{CUP(k).decide}_j\texttt{(v)}$ occurred for some v and set $\texttt{CUP-dec-val[k]}_j$ = v. By the well-formedness property of CUP, j initiated CUP(k). Since $k \in \texttt{suspects[s]}_j$, k cannot be included in $\texttt{suspects[s']}_j$ for any s' $\neq$ s, and so j initiated CUP(k) with s. By the validity condition of CUP, v $\leq$ s.

Since s+1 $\in \texttt{ended-slots}_i$ when $\texttt{members(P,s)}_i$ occurs, by Invariant A.21, $k \in \texttt{suspects[s+1]}_i$ at this time. Therefore, by the precondition for $\texttt{members(P,s)}_i$, $\texttt{CUP-status[k]}_i$ = done. By A.20, $\texttt{CUP-dec-val[k]}_i \neq \perp$, that is, $\texttt{CUP(k).decide}_i$ occurred, and by the uniform agreement property, $\texttt{CUP-dec-val[k]}_i$ = $\texttt{CUP-dec-val[k]}_j \leq$ s. A contradiction.

Now, assume that $\texttt{CUP-dec-val[k]}_j \leq$ s when $\texttt{members(P',s)}_j$ occurs. Since s+1 $\in \texttt{ended-slots}_i$ when $\texttt{members(P,s)}_i$ occurs, by Invariant A.22, $k \in \texttt{suspects[s+1]}_i$ at this time. Therefore, by the precondition for $\texttt{members(P,s)}_i$, $\texttt{CUP-status[k]}_i$ = done. By Invariant A.20, $\texttt{CUP-dec-val[k]}_i \neq$

$\perp$, that is, CUP(k).decide$_i$ occurred, and by the uniform agreement property, CUP-dec-val[k]$_i$ = CUP-dec-val[k]$_j$ $\leq$ s. A contradiction. ∎

Lemma A.10 *If* members$_i$(P,s) *occurs, then for every* j ∈P, i *received a bulk message for slot* s *from* j *prior to the* members$_i$(P,s) *action.*

Proof: Assume members$_i$(P,s) occurs. Since j ∈P, by the precondition for members$_i$(P,s), $j \in$ alive[s]$_i$. By definition of alive[s], in-buf[s,j] $\neq \perp$, that is, i received a bulk message from j for slot s. ∎

A.3 Atom Correctness Proof: Liveness Arguments

In the liveness proof, we can use the safety guarantees of CUP, since they depend only on the safety assumptions about the environment.

A.3.1 General liveness lemmas

Lemma A.11 *Time passes.* current-slot$_i$ *increases through all slot values from zero onward, as long as* i *does not fail.*

Lemma A.12 *If* i *does not leave or fail, then* end_slot$_i$(s) *occurs for every slot* s $\geq$ join-slot$_i$.

Lemma A.13 *If* i *leaves and does not fail, then eventually* i *multicasts a* (i, LEAVE, s) *message.*

A.3.2 Liveness environment conditions for CUP

Init occurrence Assume that init$_i$(s, W) event occurs and $j \in$ W, and neither i nor j leaves or fails.

Since j ∈W, j ∈alive[s] $\cup$ joiners[s+1] at the time init$_i$(s, W) is triggered, which means that net_join_OK$_j$ had already occurred prior to the init$_i$(s, W) event. When init$_i$(s, W) is triggered, i multicasts an (CUP-INIT, k) message. Since neither i nor j leaves or fails, j receives this message.

Consider the pre-state value of CUP-status[k] when the (CUP-INIT, k) message from i arrives at j. If CUP-status[k] is running or done, then either CUP(k).init$_j$ or CUP(k).abstain$_j$ had to have already occurred and we are done. Otherwise, after this step CUP-status[k] = req, and CUP-req-val[k] =v. Since j does not leave or fail, by Lemma A.12, it eventually has slots larger than v in ended-slots$_j$, so either CUP(k).init$_j$(*) or abstain CUP(k).abstain$_j$ becomes enabled, depending on whether k is suspected in some slot or in none.

Reliable delivery Assume that for some processes j, k, l CUP(k).init$_l$(v, W) occurs with j ∈W, and that either CUP(k).init$_i$(v', W') or CUP(k).abstain$_i$ occurs. We will show that by the time that either CUP(k).init$_i$(v', W') or CUP(k).abstain$_i$ occurs, net_join_OK$_j$ had already occurred. This will imply that for any net_mcast$_i$(m) that occurs after this event, a net_rcv$_j$(m) will occur unless either i will fail, or j will fail or leave.

If CUP(k).init$_i$(v', W') occurs, by Lemma A.8, net_join_OK$_j$ occurs first. Now, consider the case that CUP(k).abstain$_i$ occurs. Process i can only abstain after it receives an (CUP-INIT, k) message which could have only been sent if some other process i' has already triggered CUP(k).init$_{i'}$. By Lemma A.8, net_join_OK$_j$ must have occurred before the CUP(k).init$_{i'}$ event.

Complete leave and failure detector If `CUP(k).init`$_i$`(v,W)` occurs with $j \in W$, then $j \in$`alive[v]`$_i$ $\cup$ `joiners[v+1]`. Assume that i does not decide or leave or fail. Then `CUP-status[k]`$_i$ remains `running` from the time of the `CUP(k).init`$_i$`(v,W)` event onward. If `leave`$_j$ occurs, j sends a LEAVE message which i receives. When i receives j's LEAVE message, i triggers `leave_detect`$_i$`(j)` and we are done. Otherwise, assume j does not leave and `fail`$_j$ occurs, then eventually there is a slot for which i does not receive j's messages. Let `s` be the first such slot, so $j \notin$`alive[s]`$_i$ while $j \in$`alive[s-1]`$_i$ $\cup$ `joiners[s]`$_i$, so since j does not leave in `s`, $j \in$`suspects[s]`. Since $j \in$`alive[v]`$_i$ $\cup$ `joiners[v+1]`, `s > v`, and i triggers `fail_detect`$_i$`(j)` while performing `end_slot`$_i$`(s)`.

A.3.3 Liveness of Atom

Eventual join Assume no `fail`$_i$ occurs. When `join`$_i$ occurs, `net_join`$_i$ is triggered, and by fairness, eventually occurs. By the eventual join property of Net, `net_join_OK`$_i$ eventually occurs. At that point, `join-slot`$_i$ is set to be bigger than `current-slot`$_i$. `join-slot`$_i$ does not change from that point onward, since by the join integrity property of Net, no more `net_join_OK`$_i$ events occur. By Lemma A.11, `current-slot`$_i$ eventually becomes equal to `join-slot`$_i$. When that happens, `join_OK`$_i$ becomes enabled, and remains enabled, as long as no time passes, until it occurs. By our assumption on time passage, no time passes until `join_OK`$_i$ occurs. Therefore, by fairness, it eventually occurs.

Eventual leave Assume no `fail`$_i$ occurs. When `leave`$_i$ occurs, `leave-slot`$_i$ is set to be bigger than `current-slot`$_i$. `leave-slot`$_i$ does not change from that point onward, since by our assumption on the application, no more `leave`$_i$ events occur. By Lemma A.13, i eventually multicasts a `(i, LEAVE, leave-slot`$_i$`)` message, at which point `leave-slot`$_i$ is added to `mcast-slots`$_i$. When that happens, `net_leave`$_i$ becomes enabled and remains enabled until it occurs. Then, by the eventual leave property of Net, `net_leave_OK`$_i$ eventually occurs and triggers `leave_OK`$_i$.

Message delivery The following lemma asserts that a process that participates in the algorithm and does not leave or fail continues to perform `members(P, s)` forever.

Lemma A.14 *If* `mcast`$_i$`(m)` *occurs for some* m *when* `s = current-slot`$_i$, *and no* `fail`$_i$ *or* `leave`$_i$ *occurs, then for every* `s'` $\geq$ `s`, `members`$_i$`(P, s')` *occurs.*

Proof: Since `mcast`$_i$`(m)` occurs, by our assumption about the application, it is preceded by a `join_OK`$_i$. Therefore, m is appended to `out-buf[s]`$_i$. By Lemma A.11, `current-slot`$_i$ becomes `s+1`, and so i eventually sends its bulk message for slot `s` with m included in it. By liveness of Net, `net_rcv`$_i$`(i, m', s)` occurs, where m' is i's slot `s` bulk message.

By Lemma A.12, `end_slot`$_i$`(s)` occurs for every slot `s` $\geq$ `join-slot`$_i$. The sets `suspects[s]`$_i$ and `suspects[s+1]`$_i$ are set when `end_slot`$_i$`(s)` (resp. `end_slot`$_i$`(s+1)`) occurs, at which point a CUP instance for each process in these sets is initiated, and the set do not change afterwards. By the termination property of CUP, these instances of CUP eventually terminate, setting the corresponding `CUP-status` to `done`. Therefore, `members(P, s)`$_i$ eventually becomes enabled for some `P`, and by fairness, occurs. ■

We now prove that the message delivery liveness property holds.

Assume `mcast`$_i$`(m)` occurs, and no `fail`$_i$ or `leave`$_i$ occurs. We first show that S contains m and `rcv`$_i$`(m)` occurs. Let `s = current-slot`$_i$ when `mcast`$_i$`(m)` occurs. By Lemma A.14, `members(P,` s$)_i$ occurs. We now show that $i \in P$. This will imply that $m \in S$ (by definition of S), and that $\texttt{rcv}_i\texttt{(m)}$ occurs (since it is triggered by $\texttt{members(P, s)}_i$).

To show that $i \in$P, we have to show that j $\in$ $\texttt{alive[s]}_i$ and $\texttt{CUP-dec-val[i]}_i$ = $\perp$ at the time $\texttt{members(P, s)}_i$ occurs. Since at this time s+1 $\in$ $\texttt{ended-slots}_i$, by Invariant A.16, j $\in$ $\texttt{alive[s]}_i$. By Invariant A.18, since i does not fail it never becomes a suspect, and therefore, no instance of CUP is run for i, and $\texttt{CUP-dec-val[i]}_i$ = $\perp$.

It remains to show that for every m' that follows m in S, $\texttt{rcv}_i\texttt{(m')}$ also occurs. By definition of S, m' is included in a bulk message for some slot s' $\geq$ s from some process i', such that $i' \in P'$ and $\texttt{members}_j\texttt{(P', s')}$ occurs for some j. By Lemma A.14, $\texttt{members(P'', s')}_i$ also occurs, and by Lemma A.9, P' = P''. Therefore, $\texttt{rcv}_i\texttt{(m')}$ is triggered by the $\texttt{members(P'', s')}_i$ action.