MIT-CSAIL-TR-2006-070 October 12, 2006

Implementing Atomic Data through Indirect Learning in Dynamic Network

K. Konwar, P.M. Musial, N.C. Nicolau, and A.A. Shvartsman.

Implementing Atomic Data through Indirect Learning in Dynamic Networks

Anonymous

Abstract

Developing middleware services for dynamic distributed systems, e.g., ad-hoc networks, is a challenging task given that such services must deal with communicating devices that may join and leave the system, and fail or experience arbitrary delays. Algorithms developed for static settings are often not usable in dynamic settings because they rely on (logical) all-to-all connectivity or assume underlying routing protocols, which may be unfeasible in highly dynamic settings. This paper explores the indirect learning *approach to information dissemination within a dynamic distributed data service. The indirect learning scheme is used to improve the liveness of the atomic read/write object service in the settings with uncertain connectivity. The service is formally proved to be correct, i.e., the atomicity of the objects is guaranteed in all executions. Conditional analysis of the performance of the new service is presented. This analysis has the potential of being generalized to other similar dynamic algorithms. Under the assumption that the network is connected, and assuming reasonable timing conditions, the bounds on the duration of the read/write operations of the new service are calculated. Finally, the paper proposes a deployment strategy where indirect learning leads to an improvement in communication costs relative to a previous solution.*

Keywords: *Distributed algorithms, atomic objects, dynamic networks, performance*

1 Introduction

Distributed middleware services for dynamic systems must deal with communicating devices that may fail, join, or voluntarily leave the system, and experience arbitrary delays in message delivery. A common design approach in such settings is to have the participating network nodes periodically exchange their local state information with the goal of approximating the global state of the system and ensuring progress of local computation. Performance of a service implemented in this way depends on the prompt update of the local state at each node, hence requiring (logical) all-to-all communication, which can be quite expensive. The communication cost associated with all-to-all communication can be reduced by minimizing the number of bits in the message [2], or by limiting the communication by assigning to each sender a proper subset of the nodes to communicate with [11]. Such methods can lead to good results in static environments, however their utility is diminished in highly dynamic networks. A weakness of all-to-all gossip is its reliance on the existence of point-to-point connectivity. This is an important limitation, since in dynamic systems such as ad-hoc and mobile networks, maintenance of routing information is prohibitively expensive, where significant amount of power, memory, and communication are needed to keep the routing tables up to date [18, 9, 19, 20]. Furthermore, routing protocols provide a general solution and are oblivious to the data flows of specific applications, which results in unnecessary communication burden. On the other hand, in the absence of a routing service no predictable progress can be ensured in algorithms depending on all-to-all gossip.

In this paper we incorporate an indirect learning protocol within a distributed algorithm implementing atomic objects with the purpose of enhance its effectiveness in dynamic networks. Our algorithm is based on RAMBO [15] and it ensures atomicity in all executions while tolerating node departures, joins, failures, and message loss. Data objects are replicated to ensure survivability. To maintain consistency in the presence of small and transient changes, the algorithm uses *configuration* consisting of *quorums* of locations. To accommodate larger and more permanent changes, the algorithm supports *reconfiguration,* by which the configurations are modified. All decisions regarding the locally initiated operations on the replica are made by examining the local state. In order to update the local state and ensure operation liveness, RAMBO relies on point-to-point connectivity and uses all-to-all gossip to periodically exchange information about the state of replicas. Our goal is to enable progress of data access operations (reads and writes) as long as there are quorums in active configurations whose nodes are connected, either directly or indirectly, and without relying on routing protocols.

Contributions. We present an atomic service for read/write objects in dynamic networks that incorporates an indirect learning mechanism designed to take advantage of the semantics of the data flow within the service to effectively disseminate object replica information among participating nodes. We call the new algorithm ATILA (atomicity through indirect learning algorithm). The dynamic settings considered include mobile ad-hoc networks (MANETS), and we do not assume an underlying routing protocol or all-to-all direct connectivity.

The algorithm implements indirect learning through local gossip and it achieves improvements in liveness in dynamic network settings at the expense of higher memory consumption. Implementing indirect gossip requires each node to maintain an estimate of the state of every other participating node. This information is included in the state messages that are exchanged between direct neighbors only. We first present a general solution that is oblivious to the communication structure or existence of routing protocols. This solution trades service liveness for inefficiency in memory and communication cost, however allows optimizations that improve its performance. In this presentation we discuss one example of one such optimization.

We formally prove that ATILA implements atomic objects. The performance of read and write operations of the service is affected by the properties of the service deployment graph, where the edges are direct communication links between nodes. We give probabilistic analysis estimating the duration of read/write operations; we also analyze possible savings in cost per message bit. Of independent interest, we believe that our analysis approach can be generalized to other algorithms that use quorums.

For lack of space, the formal code specification using Input/Output Automata notation [16] appears in [12].

Related work. Dynamic distributed systems with an unknown and possibly unbounded number of participants that may join, voluntarily leave, and fail, are becoming increasingly common. Problems that often need to be solved in these settings include leader election [17], consensus [13], and maintenance of consistent memory [3].

Group communication services (GCS) [1] are important building blocks in distributed systems and can be used to implement shared memory abstractions. However, communication required for group maintenance limits the utility of common GCSs in dynamic environments such as MANETS. Here the mobility of nodes results in frequent group membership changes and group maintenance becomes an expensive task requiring high communication overhead and energy consumption [10].

The GEOQUORUMS approach of [3] uses stationary *focal points*, implemented by mobile nodes, to provide atomic shared read/write memory where consistency is maintained by using quorums of focal points. However this service relies on the availability of *geocast* that can deliver messages to specific geographic locations. The earlier RAMBO service [15] was developed for dynamic overlay networks, where messages are routed automatically. The specification of RAMBO trades mathematical simplicity for practicality, and while the successive refinements [7, 4, 5, 8] improved this service's usability each still relies on automatic all-to-all connectivity.

Overlay networks provide the ability to transparently route messages atop diverse communication structures. Nodes communicate using virtual point-to-point channels with the help of routing protocols. Many routing algorithms for ad-hoc and mobile networks have been proposed, e.g., DSDV [19], TORA [18], DSR [9], and AODV [20]. However, routing protocols have the following drawbacks: (i) Maintenance of overlay routes in systems where nodes join, migrate, depart, and fail, is expensive in terms of processing, memory consumption, and communication; additionally, if the devices are mobile, then the topology of the network may change frequently and the new virtual routes have to be recalculated often in order to maintain integrity of the overlay network. (ii) Routing protocols are oblivious to the semantics of the communication among the participating nodes. Hence, there may be substantial redundancy in communication. In the networks that are sensitive to throughput, increased communication burden may have adverse effects on the performance of the routing algorithms themselves and on the message-passing applications.

Document structure. In Section 2 we present the model and definitions. We describe our algorithm in Section 3. The proof of atomicity is given in Section 4 (for lack of space the proofs are not stated). Probabilistic performance analysis is presented in Section 5 and the deterministic analysis in Section 6. We conclude in Section 7. For presentation reasons we present the full proofs and the complete code of the algorithm in the attached appendix.

2 System Model and Definitions

We assume a message-passing model with asynchronous processors with unique identifiers. We denote by I the set of node identifiers (I need not be finite). Processors may join, crash, and voluntarily leave the system.

Processors communicate via point-to-point, direct, asynchronous channels. A processor can send a message to another processor if a direct link between the processors exists. In safety (atomicity) proofs we do not make any assumptions about the length of time it takes for a message to be delivered. To evaluate performance of the algorithms, we assume that either messages are delivered in bounded time or not delivered at all. The nodes and the point-to-point communication links form the *service deployment graph*. The deployment graph may change over time, as nodes join, depart, and fail during the computation. In performance analysis we also assume that the graph is connected.

We denote by C the set of *configuration identifiers*. For each $c \in C$ we define: (i) *members*(c), a finite subset of node identifiers, (ii) *read-quorums*(c), a set of finite subsets of *members*(c), and (iii) *write-quorums*(c), a set of finite subsets of *members*(c). We require that for every $R \in \mathit{read\text{-}quorums}(c)$, and every $W \in \mathit{write\text{-}quorums}(c)$, $R \cap W \neq \emptyset$. No intersection requirement is imposed on the sets of members or on the quorums from distinct configurations.

We define $C_\perp = C \cup \{\perp\}$ and $C_\pm = C \cup \{\perp, \pm\}$ to be the partially ordered sets, such that: $\perp < c$ and resp. $\perp < c < \pm$, for $c \in C$. We define the set *CMap*, the set of configuration maps, as the set of mapping $\mathbb{N} \to C_\pm$. In any sequence in *CMap*, the symbol $\perp$ represents an unknown configuration and $\pm$ represents obsolete configuration that has been removed. We define $Usable$ to be the subset of $CMap$ such that $cm \in Usable$ iff the pattern occurring in cm consists of a prefix of finitely many $\pm$s, followed by an element of C, followed by an infinite sequence of elements of $C_\perp$ in which all but finitely many elements are $\perp$. We define $Truncated$ to be the subset of $CMap$ such that $cm \in Truncated$ iff the pattern occurring in cm consists of a prefix of finitely many $\pm$s, followed by a finite number of elements from C, followed by an infinite sequence of $\perp$. We define *truncate* to be a unary operation on $cm \in CMap$ that removes all configuration identifiers that appear after the first $\perp$ in cm. Finally, we define *update* to be a binary operation on $cm, cm' \in CMap$ that updates any element in cm with the corresponding element in cm' if that element is greater according to the partial order $C_\pm$.

3 The Algorithm

We now present the algorithm implementing a dynamic atomic object service using an indirect learning protocol. The algorithm is based on RAMBO [15] and its refinements in [7, 4], and we call the new algorithm ATILA (atomicity through indirect learning algortihm). The service is defined for a single object — given that atomicity is preserved under composition a complete shared memory is implemented by composing multiple instances of the service. The pseudocode of the algorithm appears in Figures 1 and 2.

---

read() or write(v) operation at node i:

- **RW-Start:** Node i resets its local structures pertaining to the read/write operations, such as: $op\text{-}configs$, $op\text{-}Nums$. Also, it notes that a read or a write operation was initiated.
- **RW-Phase-1a:** Node i increments its local phase number and updates the $pNums$ set with the new information. A snapshot of the information stored in $configs$ and $pNums$ is recorded in $op\text{-}configs$ and $op\text{-}pNums$. At this point node i sets out to query configurations found in $op\text{-}configs$ for the most recent tag and $value$ information. Next, i sends $\langle RW1a, tag, val, configs, world, pNums\rangle$ message to all known participants of the service, i.e. $world$.
- **RW-Phase-1b:** Upon receipt of a $\langle RW1a, t, v, c, w, pn\rangle$ message from i, node j compares its local knowledge (local state values) with the information included in the message. For instance if its local tag is strictly smaller than t, then it updates its tag with t and $value$ with v. Also, it updates its $configs$, $world$, and $pNums$. Next, j replies to i with $\langle RW1b, tag, val, configs, world, pNums\rangle$.
- **RW-Phase-1c:** Upon receipt of a $m = \langle RW1b, t, v, c, w, pn\rangle$ message from j, node i updates its state based on comparison of the values of its local state with the related information found in the message. If $m.c$ contains configurations previously unknown to i, then the current phase is restarted.
- **RW-Phase-2a:** Node i compares $m.pn$ and $op\text{-}pNums$ to check if at least one read quorum of each configuration found in $op\text{-}configs$ has an adequately recent state information of i (i.e. has at least learned the phase number of i from **RW-Phase-1a**) If so then the first phase is complete – i is now in the position of the highest tag. At this point node i sets out to propagate to the members of configurations found in $op\text{-}configs$ the most recent tag and $value$ information. Node i increments its phase number and updates its $pNums$ with the new information, it also records current values of $configs$ and $pNums$ in $op\text{-}configs$ and $op\text{-}pNums$. Next, i broadcasts $\langle RW2a, tag, val, configs, world, pNums\rangle$ message where tag and $value$ depend on whether it is a read or a write operation: in the case of a read, they are just equal to the local tag and $value$; in the case of a write, they are a newly chosen tag, and v, the value to write.
- **RW-Phase-2b:** If node j receives a $\langle RW2a, t, v, c, w, pn\rangle$ message from i, it updates its state accordingly, and responds to i with $\langle RW2b, tag, val, configs, world, pNums\rangle$.
- **RW-Phase-2c:** Same as **RW-Phase-1c**.
- **RW-Done:** If node i can determine that at least one write quorum of *all* configurations in $op\text{-}configs$ has an adequately recent state information of i (i.e. has at least learned the phase number of i from **RW-Phase-2a**), then the read or write operation is complete and the tag is marked confirmed. If it is a read operation, node i returns its current value to client. Node i marks that the operation is now terminated. At this point new read/write operation may be initiate at node i.

Figure 1. Description of the phases of the read and write protocols.

In order to ensure fault tolerance, object data is replicated at several nodes. The algorithm uses *quorum configurations* to maintain consistency. Configurations can be modified on-the-fly through *reconfiguration*. Main parts of the algorithm deal with communication with replicas during read and write operations, and the removal of the obsolete configurations using *configuration upgrade* operations. Network topology may change during the lifetime of the service, where links may be created and consequently destroyed. However, if the service deployment graph maintains its connectivity, then our algorithm is eventually able to propagate the replica information throughout the system and allow indirect communication with the replicas during individual operations.

Participant Information. Each participant maintains the *value* and the associated *tag* of the object being replicated. The *tag*s are used to totally order write operations with respect to each other and all read operations with respect to the writes — this forms the basis for the proof of atomicity (Section 4). Each node maintains a set of node identifiers, *world*, representing the nodes that are locally known to have joined the service, and the configuration information stored in variable $configs$ of type $CMap$ (Section 2).

Each node uses *phase numbers* to logically timestamp the messages it sends to other nodes indicating the "freshness" of the state conveyed in the messages. The phase number of a node is incremented following an "important" event at a node, such as the start of a new phase of a read or a write, or a configuration upgrade operation. Most importantly, phase numbers are used to implement indirect learning as discussed later in this section. Each node i maintains a matrix of phase numbers, *pNums*, where rows and columns are indexed by node identifiers, hence its size is $|world| \times |world|$. The variable $pNums[i][j]$ represents the most recent phase information known to i about another participating node j. This means that i has learned the replica information known to j when j's phase number was equal to $pNums[i][j]$. The variable $pNums[j][k]$, for some $j, k \in world$ and $i \neq j$, represents the most recent phase number known to i about the phase of node k that is known to j. Each of these variables reflects the latest information locally known at a node, but not necessarily the most up-to-date global information.

Each node i also maintains two records used to store information about the ongoing operations. Record op is used to keep track of the phases of read and write operations. The following fields of op are initialized when a new phase of a read or write operation is initiated: $op\text{-}configs$ records the value of $configs$, $op\text{-}Nums$ records the value of $pNums$, and $op\text{-}acc$, initially $\emptyset$, records the identifiers of the nodes that contain adequately current information regarding i's state. Similarly, record upg is used to keep phase information of the configuration upgrade operation, where the fields $upg\text{-}configs$, $upg\text{-}Nums$ and $upg\text{-}acc$ are defined analogously to the fields of op record. In addition, the upg record contains field $upg\text{-}target$ containing the index of the configuration being upgraded. (The phases of read, write, and configuration operations are discussed later in this section).

Information Propagation and Indirect Learning. Periodically, and following certain events, any non-failed participant of the service sends state messages to all nodes found in its local $world$. These messages include sender's current values of: tag, val, $configs$, $world$, and $pNums$. Although a node attempts to send messages to all nodes in its $world$, only the messages addressed to the nodes with a direct connection may be delivered, all other messages may be lost. (In a practical implementation of the service, a node may use timeouts or other means of failure detection to stop sending messages to the nodes without a direct connection. This does not affect the safety.)

We now narrate the update process based on an example of a message exchange between two non-failed service participants, say i and j. When i receives message from j it compares values of variables comprising its state against the information included in the message. Assume that node i receives message $m = \langle tag, val, configs, world, pNums\rangle$ from j. If $m.tag \geq tag$ then node i updates its tag with $m.tag$ and the value with $m.val$. Next, node i includes in its $world$ any new identifiers found in $m.world$. For each new node identifier, matrix *pNums* is extended with a new column and a new row, intitalized to zeros. Node i also sets its $configs$ to $update(configs, m.configs)$.

The last step updates the phase information, where i compares its phase matrix with the one in the sender's message. This update captures the indirect learning process. For all $k, \ell \in m.world$, if $m.pNums[k][\ell] > pNums[k][\ell]$, then j knows that k has learned about a higher phase number of ℓ. Therefore, whenever $m.pNums[k][\ell] > pNums[k][\ell]$ then i assigns $pNums[k][\ell] \leftarrow m.pNums[k][\ell]$.

Observe that all bookkeeping information (except for value) is monotonically growing with each update, i.e., a tag is updated only when the arriving tag is larger, nodes are only added to the $world$ set, and the phase number information is updated if the incoming phase number information is more recent than what i is aware of. Therefore, if some node k learns that i's phase number is p, then k has learned of a tag (resp. value) of the replica that is at least as recent as when i's phase number was p. Phase numbers are updated either following a receipt of a message directly from k or indirectly from some other node. Thus if i is performing some operation and p is its current phase number then if $pNums[k][i] \geq p$, then i can deduce that k learned the information that is at least as recent as the information communicated by i to its $world$ in phase p. (Finally, if the service deployment graph is connected and the network is reasonably well-behaved, then eventually i will (indirectly) learn that k (indirectly) learned the information disseminated by i.)

Joining. Nodes join the service by sending a join request to the nodes provided by the user ("seeds"). Our well-formedness assumption is that when the set of seed nodes is empty, the node processing the join request is the "creator" of a new object. If an active participant of the service receives a join request it will add sender's identifier to its local *world* set and reply with a state message. The joinee becomes operational (*active*), when a response message to the join-request is received.

Read and Write Operations. The read and write operations are conducted in two phases (see Figure 1): The first phase called **RW-Phase-1**, or *query* phase, is identical for both operations. In this phase the initiator of the operation queries the replica owners in order to obtain the most recent *tag* and the associated *value*. The second phase is called **RW-Phase-2**, or *propagation* phase. In case of a read, the initiator of this operation *propagates* the information learned in the *query* phase. Since the aim of the write operation is to change the value of the replica, in the *propagation* phase the new *tag* is created which is strictly larger than the one discovered during the *query* phase and the new value is associated with this tag. This is the information that is propagated to the replica owners.

The termination point of each phase is determined only after the node conducting this operation can certify that at least one quorum of replica owners from each active quorum set has responded to (directly or indirectly) to its latest phase information.

Reconfiguration and Configuration Upgrade. The reconfiguration is performed in two steps (see Figure 2, where these steps are similar to ones performed by the write operation). First, a new configuration is chosen by the members of the most recent cfg-upgrade(k) at node i (similar to the phases of read/write operations):

- **UPG-Phase-1a:** Node i chooses an index k, such that k is a configuration identifier that ends the prefix of the sequence of configurations known to i, where there are zero or more configurations up to some ℓ that have been marked as removed, and all configurations with index $\ell + 1$ to k are active. Next, i increments is phase number and updates its $pNums$ with the new information, it also records current values of $configs$ and $pNums$ in $upg.configs$ and $upg.pNums$. A message $\langle UPG1a, tag, val, configs, world, pNums\rangle$ is sent by i to all nodes in its $world$.
- **UPG-Phase-1b:** If node j receives a $\langle UPG1a, t, v, c, w, pn\rangle$ message from i, it performs all necessary updates based on the information contained the message, and replies to i with $\langle UPG1b, tag, val, configs, world, pNums\rangle$.
- **UPG-Phase-2a:** If node i receives $m = \langle UPG1b, t, v, c, w, pn\rangle$ message from j, it updates its state accordingly. If based on the latest $m.pn$ it can determine that at least one read and one write quorum of each configuration in $upg.configs$ has an adequately recent state information of i (i.e. has at least learned the phase number of i from **UPG-Phase-1a**), then the first phase is complete. Then, i increments its phase number, updates $pNums$ and records current values of $configs$ and $pNums$ in $upg.configs$ and $upg.pNums$. Node i sends a $\langle UPG2a, tag, val, configs, world, pNums\rangle$ message to all members of its $world$.
- **UPG-Phase-2b:** If node j receives a $\langle UPG2a, t, v, c, w, pn\rangle$ message from i, it updates its state and replies to i with message $\langle UPG2b, tag, val, configs, world, pNums\rangle$.
- **UPG-Done:** If node i receives a $\langle UPG2b, t, v, c, w, pn\rangle$ message and if from that message i can determine that at least one write quorum of configuration $c(k)$ has an adequately recent state information of i (i.e. has at least learned the phase number of i from **UPG-Phase-2a**), then the upgrade operation is complete. Node i marks all configurations with identifier smaller than k as removed.

Figure 2. Description of the phases of the *configuration upgrade* protocol.

configuration. This is handled by an external service, called *Recon*, as in [15]. Then obsolete configurations are removed using the *configuration upgrade* operation. This operation upgrades a configuration at a node by removing every configuration with a smaller index from its $configs$ variable. Once a configuration has been upgraded, it is responsible for maintaining the data. Note that we assume that old configurations remain operational until they are removed. In Section 5 we describe the timing conditions on configuration viability.

4 Proof of Atomic Consistency

In this section we formally show that ATILA implements atomic objects by applying necessary refinements on the safety proofs of RAMBO [7]. The challenge here is to show that atomic access to the object is ensured when indirect mechanism is used. In the following discussion we present the lemmas that required modification and only a brief discussion of the remaining lemmas leading up to the main theorem. The omitted details may be found in the optional appendix.

4.1 Definitions and notation.

In the rest of the presentation, we consider "good" executions of the algorithm: the assumptions are that the client requests are well-formed requests, i.e., clients follow the protocols for joining and initiating reconfiguration; clients initiate only one operation at a time; clients wait for appropriate acknowledgments before proceeding.

We denote by α an arbitrary, good execution of the algorithm. We let π_1 and π_2 be two read or write operations that occur at nodes i and j respectively, where i and j are participants of ATILA service. Additionally, we assume that π_1 completes before π_2 begins in α. When we do not refer to any ordering of operations we use π to denote an arbitrary read or a write operation. Also let γ denote the configuration upgrade operation initiated by some active participant of the service. Before proceeding with the safety claims we state additional definitions.

For every π, the query-fix (resp. prop-fix) event occurs immediately after the *query* (resp. *prop*) phase of π completes. Therefore, query-fix point occurs at the point when node i determines that at least one read quorum of each configuration in $op\text{-}configs$ has a sufficiently recent state information of i, which happens in phase **RW-Phase-2a** (Figure 1). A similar relation exists between prop-fix and **RW-Done**. For every configuration upgrade operation γ, the cfg-upg-query-fix and cfg-upg-prop-fix events are defined analogously.

Next we introduce history variables. First, the $query\text{-}cmap(\pi)$ is a mapping: $\mathbb{N} \to C_{\pm}$, initially undefined. It is set in the query-fix step of π, to the value of $op\text{-}configs$ in the pre-state. The history variable $prop\text{-}cmap(\pi)$ is defined analogously for the propagation phase of operation π. The query-phase-start(π), initially undefined, is defined in the query-fix step of π, to be the unique earlier event at which the collection of query results was started and not subsequently restarted (the last time $op\text{-}acc$ set is assigned $\emptyset$). This is either in **RW-Start** step of a read or a write operation, or in **RW-Phase-1c** step. The event prop-phase-start(π) is defined analogously, but with respect to the propagation phase.

For every read or write operation π at node i, we define the history variable $tag(\pi)$ to be the value of tag_i when the query-fix event occurs for π at node i. If π is a read operation then $tag(\pi)$ is the largest tag that node i encounters during the query phase. If π is a write operation, $tag(\pi)$ is the new tag that is chosen by i for performing the write. Similarly, for a configuration upgrade operation γ at node i, we define $tag(\gamma)$ to be the tag at node i (i.e., tag_i) when the cfg-upg-query-fix event occurs, that is, the largest tag encountered at node i during the query phase of γ.

The history variable $removal\text{-}set(\gamma)$, is defined for the configuration upgrade operation γ. It is a subset of $\mathbb{N}$, initially undefined, and records the configuration identifiers of configurations that are marked for removal (whose identifiers are less less than the value of $upg\text{-}target$ for γ.) The history variable $in\text{-}transit$, defined as a set of all messages that are sent by any participant of the service.

Finally for any operation π we define the history variable $R(\pi, k)$, for $k \in \mathbb{N}$, as a subset of I, initially undefined. It is set in the query-fix step of π, for each k such that $query\text{-}cmap(\pi)(k) \in C$, to an arbitrary $R \in read\text{-}quorums(c(k))$ such that $R \subseteq op\text{-}acc$ in the pre-state, where $c(k) \in C$. Similarly we define $W(\pi, k)$, for $k \in \mathbb{N}$, to be a subset of I, initially undefined and set during the prop-fix step of π, for each k such that $prop\text{-}cmap(\pi)(k) \in C$, to an arbitrary $W \in write\text{-}quorums(c(k))$ such that $W \subseteq op\text{-}acc$ in the pre-state. Similarly we define $R(\gamma, \ell), W_1(\gamma, \ell), W_2(\gamma)$ for any configuration upgrade operation γ. $R(\gamma, \ell)$ and $W_1(\gamma, \ell)$ are set in the cfg-upg-query-fix step of γ, for each $\ell \in removal\text{-}set(\gamma)$, to an arbitrary $R \in read\text{-}quorums(c(\ell))$ (resp. $W \in write\text{-}quorums(c(\ell))$, such that $R \subseteq upg\text{-}acc$(resp. $W \subseteq upg\text{-}acc$) in the pre-state. $W_2(\gamma)$ is set in the cfg-prop-query-fix of γ to arbitrary $W \in write\text{-}quorums(c(k))$ such that $W \subseteq upg\text{-}acc$ in the pre-state, where $c(k) \in C$ is the target of γ.

Note that the only updates on the *CMap* in various places in the system are allowed via the *update* and *truncate* operations. Hence, in any state of the execution *CMap* that is a part of a message that is in transit, $configs_i$, $op\text{-}configs_i$, $query\text{-}cmap(\pi)$, $prop\text{-}cmap(\pi)$, and $upg\text{-}configs_i$, for some $i \in I$ and any operation π, always has the *Usable* property. Moreover, a *CMap* that appears as $op\text{-}configs_i$, $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$, for some $i \in I$ and any operation π that has initiated a read/write operations which has not terminated yet, always has the $Truncated$ property. (These properties are easily described as invariants on the service, however such formal presentation is omitted from this discussion.)

Phase guarantees. Lemmas presented in this section discuss the effects of query and propagation phases of read/write and configuration upgrade operations. In more detail, we describe the information flow that must occur during these phases to allow operation completion. We show that if node i initiates a phase of a read/write or a configuration upgrade operation and if there exists a specific sequence of message exchanges that starts and ends at i, then if that phase terminates, i is in possession of the most recent tag and its value cannot be smaller than what i knew at the start of the phase. Moreover, we show that configuration information and value of the tag at each node that participated in the examined communication sequence has specific properties. Our claims are based on the following observation: A node send the most recent state information that includes its configuration information, value and tag, and phase information of all service participants. By the specification of the algorithm, the receiver of this message can only increase its tag and increment the phase information in any cell of its phase number matrix. Also, the configuration information is updated only with a more recent one. This means that nodes may learn about configuration information, tag, and phase information of other participants indirectly.

Note, the case $j = i$ is treated uniformly with the case where $j \neq i$. This is because, in the ATILA, communication from a location to itself is treated uniformly with communication between two different locations. First, we consider how the *tag* information is propagated in the query phase of the configuration upgrade operation. Since the flow of information in the propagation phase is analogous to that in the query phase of the configuration-upgrade operation, we compress two lemmas into one.

Lemma 4.1 *Suppose that a* cfg-upg-query-fix$(k)_i$ *(resp.* cfg-upg-prop-fix$(k)_i$*) event for configuration upgrade operation γ occurs in execution α and $k' \in removal\text{-}set(\gamma)$. Suppose $j \in R(\gamma, k') \cup W_1(\gamma, k')$ (reps. $j \in W_2(\gamma)$). Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that: (i) The message m_{ι_1,ι_2} is sent after the* cfg-upgrade$(k)_i$ *(resp.* cfg-upg-query-fix$(k)_i$*) event of γ. (ii) Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received. (iii) The message m_{ι_{n-1},ι_n} is received before the* cfg-upg-query-fix$(k)_i$ *(resp.* cfg-upg-prop-fix$(k)_i$*) event of γ. (iv) In any state after j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, $configs(\ell)_j \neq \perp$ for all $\ell \leq k$. (v) $tag(\gamma) \geq t$, where t is the value of tag_j in any state before j sends message $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$.*

Next, we consider how the *tag* information is propagated in the query phase of the read and write operation. Again, since the flow of information in the propagation phase is analogous to that in the query phase, we compress two lemmas into one.

Lemma 4.2 *Suppose that a* query-fix$_i$ *(resp.* prop-fix$_i$*) event for a read or write operation π occurs in α. Leg $k, k' \in \mathbb{N}$. Suppose $query\text{-}cmap(\pi)(k) \in C$ and $j \in R(\pi, k)$ (resp. $prop\text{-}cmap(\pi)(k) \in C$ and $j \in W(\pi, k)$). Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that: (i) The message m_{ι_1,ι_2} is sent after the* query-phase-start(π) *(resp.* prop-phase-start(π)*) event. (ii) Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received. (iii) The message m_{ι_{n-1},ι_n} is received before the* query-fix *(resp.* prop-fix *) event of π. (iv) If t is the value of the tag_j in any state before j sends $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$, then: (a) $tag(\pi) \geq t$. (b) If π is a write operation then $tag(\pi) > t$. (v) If $configs(\ell)_j \neq \perp$ for all $\ell \leq k'$ (resp. $\ell < k'$) in any state before j send $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$, then $query\text{-}cmap(\pi)(\ell) \in C$ (resp. $prop\text{-}cmap(\pi)(\ell) \in C$) for some $\ell \geq k'$.*

Atomicity. We show atomicity using the framework of Lemma 13.16 in [14]. Recall that α is an arbitrary, good execution of the algorithm. We need to show that in α if all the read and write operations that are invoked complete, then the read and the write operations can be partially ordered by an ordering $\prec$ and the following properties are satisfied. *(P1)*: $\prec$ totally orders all write operations in α. *(P2)*: $\prec$ orders every read operation in α with respect to every write operation in α. *(P3)*: for each read operation, if there is no preceding write operation in $\prec$, then the initial value is returned by this read; else, the read operation returns the value of the unique write operation immediately preceding it in $\prec$. *(P4)*: if some operation, π_1, completes before another operation, π_2, begins in α, then π_2 does not precede π_1 in $\prec$. If such ordering $\prec$ can be constructed for α, then the algorithm guarantees atomic consistency.

We define $\prec$ in terms of the lexicographic order on tags of operations π. Observe that *(P1)* to *(P3)* are essentially immediate. Lemmas 4.1 and 4.2 stated above and the additional lemmas presented in [15, 7, 4], which describe the behavior of configuration upgrade operation and read and write operations in any execution, are used to establish the monotonically increasing order on tags with respect to non-concurrent read or write operations. Based on the tags we define a partial order on operations and verify that property *(P4)* is enforced. Therefore, it follows immediately that the tags induce a partial order $\prec$ that meets the necessary and sufficient requirements for atomic consistency. Hence, the main result follows:

Theorem 4.3 ATILA *implements atomic read/write objects.*

5 Conditional Analysis of Operation Latency

In this section we examine the operation latency under similar timing assumptions as in the analysis of operations in RAMBO presented in [15, 7, 4, 6]. The analysis is done in parts: (i) we state the connectivity properties of the service deployment graph of ATILA, (ii) we present the new upper bound on the operation latency, and (iii) we present the expected operation latency in the case of restricted asynchrony under reasonable assumptions of probabilistic behavior of the algorithm. The novelty of our analysis as compared to the type of analysis done in [15, 7, 4, 6] is that here we use a more realistic assumption on the duration of message delivery. The previous analysis assumed that all messages were delivered within a fixed time interval; instead we assume a probability distribution on the delivery time of messages with finite variance.

ATILA is specified as a nondeterministic algorithm for asynchronous environments with arbitrary message delays and node crashes, departures, and new nodes joining. In such dynamic environments it is hard to quantify the speed of information propagation throughout the known universe of nodes. For the purpose of analysis, we restrict asynchrony, resolve the non-determinism of the algorithm, and impose constraints sufficient to guarantee that the universe is connected.

Assumptions. Assume α is an admissible timed execution and α' a finite prefix of α. Let $\ell time(\alpha')$ denote the time of the last event in α'. Let α be a *timed admissible execution* then we say that α is an α'-$normal$ execution if (i) no message sent in α after α' is lost, and (ii) if a message is sent at time t in α, it is delivered within bounded time (unknown to the participants).

For the purpose of latency analysis, we restrict the sending pattern of the service participants: we assume that each sends messages at the first possible time and at regular intervals of d thereafter, as measured by the local clock, and each node will immediately send messages to all of its immediate neighbors following: (i) receipt of a join request, (ii) new configuration is discovered, and (iii) receipt of a message that indicates that phase information of any node has changed. Also, the non-send and locally controlled events occur just once, and are assumed to be instantaneous.

As with all quorum-based algorithms, operational liveness depends on all the nodes in some quorums remaining active. Let us denote by $t(c)$ the time at the end of the installation of configuration c. Observe that we can always specify such a time by using the well-known axioms of time passage actions [14]. Also, we denote by c' the next configuration that has been installed after configuration c. We say that an execution α is (α',e,τ)-*configuration-viable* if for every installed configuration c, there exists a read-quorum, R, and a write-quorum, W, such that no process in $R \cup W$ fails or departs before time $\max\{t(c') + \tau, \ell time(\alpha') + e + \tau\}$, where τ is the time required to mark c as obsolete by the first configuration upgrade operation that upgrades configuration with index higher than that of c. We say that execution α satisfies (α', τ)-*recon-spacing* if after α', at least time τ elapses between the event that reports the new configuration c and any following event that proposes the new configuration c'. In other words, after α', when the system stabilizes, reconfigurations are not too frequent. Execution α is said to satisfy (α', e)-*join-connectivity* if after α', for any two nodes that both have joined the system at time t such that $t \geq \ell time(\alpha')$, they know about each other by time $t + e$. Execution α satisfies (α', τ)-*recon-readiness* if after α', every $\mathsf{recon}(c)$ event proposing a new configuration includes a node i in c only if i joined at least time τ ago. This, in conjunction with (α', e)-*join-connectivity*, ensures that all the nodes in active configurations are aware of each other.

Operation liveness depends on the connectivity property of the service deployment graph, hence we require that there is a path between any two nodes (consisting of nodes and edges). We define the connectivity property on the service deployment graph, G, as a timing assumption (α')-*connectivity*. This means that the nodes and the direct communication links may fail, but in such a way that the connectivity assumption is not violated.

Analysis. Now we provide analysis that estimates the duration of read (resp. write) operation when reconfiguration is present. To make this estimate more realistic we provide minimum timing restrictions on spacing of certain events in the system and delays on message delivery. One way of carrying out the conditional analysis is to assume fixed bounds on the delivery time of all messages as in [15, 7, 4, 6]. However, imposing rigid timing bounds on the asynchronous behavior of the assumed model (physical deployment) is too restrictive often far from reality. A more realistic approach is to assume certain probability distribution on the delivery time of the messages. Unfortunately, such probability distribution may be difficult to determine for a complex algorithm as ATILA. Under expected conditions, i.e., where the rate at which nodes join, leave, or fail and the reconfiguration of the system is not very high, we may estimate the mean delay or the standard deviation on message delivery delay.

For the purpose of analysis we consider a non-faulty participant of the service, node i, that locally initiates a read (resp. write) operation. As described in Section 3, read (resp. write) operations consist of two phases. During each phase node i must be able to deduce from examination of its state that all members of at least one read-quorum (resp. write-quorum) of each configuration found in $op\text{-}configs_i$ has a good estimate of i's state, which is a condition to reach the fix point of the current phase.

In the analysis that follows, we consider a subgraph of the service deployment graphs that is induced by members of active configurations. Let D represent the diameter of this graph. Now, consider some non-failed quorum member, j, such that the length of the communication path between i and j is D. Note that new nodes may join the service at any time and at any active participant. If a new node joined only at j and is included as a member of a configuration installed in the next reconfiguration, then the diameter D will increase. Therefore, we are interested in estimating the time required to complete a single phase of the read (resp. write) operation in a situation when new nodes join the service and become members of new configuration during the following reconfiguration attempt.

Suppose that the mean time required for a message delivery between any two nodes is λ_A with finite variance ${\sigma_A}^2$ and the mean time of a new member being inducted into the quorum is λ_B and with finite variance ${\sigma_B}^2$. Also, we assume that $\lambda_A < \lambda_B$. Meaning that on an average it takes less time for a message to be delivered from its source to its destination than the time for a new configuration to be proposed and installed (a reconfiguration attempt), for example 1 to 12 (a timing assumption used in the analysis of RAMBO algorithms in [15, 7, 4]). It is noteworthy that in a situation where the system is undergoing a rapid change or behaving perversely then the above parameters may not be estimable easily or reliably.

To simplify the analysis notationally we assume the following notations. Let $i = p_0, p_1, \cdots, p_D = j$ be a sequence of non-failed nodes and let A and B be two pointers, such that: A initially points to p_0 and B initially points to p_D. Pointer A represents the farthest node along the communication path from p_0 to p_D that has a good estimate of i's state. Pointer B points to the quorum member that is currently farthest from i.

The following argument is based on the position of these pointers along the path which help us model the performance of a read (or write). Next, we estimate the duration of a read (or write) operation that is initiated by i in the presence of reconfiguration, according from the knowledge about the first two moments of their distributions. We assume that messages are exchanged between adjacent nodes in the communication path within some random amount of time according to some probability distribution, but with the first two moments as mentioned above. Since the reconfiguration is in progress, new nodes that join at the end of the $i = p_0, p_1, \cdots, p_D = j$ which would result in a longer path $i = p_0, p_1, \cdots, p_j, p_{j+1}, \cdots, p_D$ where p_D (i.e. pointer B) is a few steps further away from p_j (i.e., $p_{j+1}, \cdots, p_D$ are the newly joined nodes). The new arrivals will join at the p_D, at the rate governed by some other probability distribution, but with the first two moments known to us. For the pointer A we denote by X_i the random variable that represents the random amount of time following the same unknown distribution, to jump from point $p_{\ell-1}$ to p_ℓ. We also assume that the random variables $X_1, X_2, \ldots$ are identically and independently distributed. Clearly, we have $\mathbb{E}(X_\ell) = \lambda_A$ and $Var(X_\ell) = {\sigma_A}^2$ for $\ell \in \mathbb{N}$. Similarly, we define a set of random variables $Y_1, Y_2, \ldots$ that are independently and identically distributed according to some distribution such that $\mathbb{E}(Y_\ell) = \lambda_B$ and $Var(Y_\ell) = {\sigma_B}^2$ for $\ell = 1, 2, \ldots$, where Y_ℓ represents the random amount of time the pointer B takes to jump from the point $D + \ell - 1$ to $D + \ell$. As mentioned before, we assume that $\lambda_A < \lambda_B$, i.e., on average the pointer A jumps more frequently than pointer B.

Definition 5.1 *We say that pointer A "catches up" with pointer B by time t if $\exists\, n, m \in \mathbb{N}, n, m > D$, such that, $n \geq m + D$ and $\sum_{1 \leq \ell \leq n} X_\ell \leq \sum_{1 \leq \ell \leq m} Y_\ell \leq t$.*

The following Lemma quantifies the time required to perform a read/write operation, with high probability, under certain *normal* behavior, which is explained in greater detail below. Intuitively, the expected time of completion of a read/write operation is sharply concentrated under certain reasonable well-behaved execution of ATILA.

Lemma 5.2 *Suppose initially pointer A points at point p_0 and pointer B points at the point p_D then A catches up with B by time $\frac{D\lambda_B}{\lambda_B - \lambda_A}$ with high probability.*

Now in the case of ATILA, we assume that the average time of delivering a point-to-point message is k times smaller than the average time of a new configuration being proposed and installed. Typically, the range of k is somewhere between 1 to 12. where the pointer A, at any time t, represents node that is aware of the initiation of the read/write operation (by node i) and closest to the node pointed to by B which represents the quorum member that is currently farthest from i. Here the distance between two nodes is measured in terms of the length of the shortest path (possibly many) between the two nodes in the communication graph where each edge has unit weight. Therefore, the time of delivering a point-to-point message is $\lambda_A = \frac{\lambda_B}{k}$ where λ_B is the average time of of a new being configured and installed. From Lemma 5.2 we see that the read/write operation takes $\frac{D\lambda_B}{\lambda_B - \lambda_A} = \frac{kD\lambda_A}{k\lambda_A - \lambda_A} = \frac{kD}{k-1}$ to complete with high probability We say that an event $\mathcal{E}$ occurs with high probability to mean that $\Pr[\mathcal{E}] = 1 - O(n^{-\alpha})$ for some constant $\alpha > 0$. where D is the diameter of the communication graph induced by the quorums.

The deterministic upper bound. Under assumptions stated above we consider the following worst case scenario. Let i be the node that initiates a read or a write operation, we denote this by the progress of the first pointer in the above analysis. At the start of the operation, let j be the node farthest from i, this distance is at most the diameter of the service deployment graph at the time when i initiates its operation, this is referred to as the second pointer. Soon after i initiates its operation, new nodes join the service. The first new node connects to j and each new node may join at the last node that joined the service. In essence the nodes that joined the service form a line. By the *recon* spacing assumption a new node may become a member of the next configuration at least $12d$ time after it joined the service.

Theorem 5.3 *Let α be a α'-normal execution of the* ATILA *that satisfies* (α', τ)-recon-spacing *then a read/write operation takes $O(N)$ time to complete since its invocation, where N is the number of nodes present at the time of invocation of the operation and $\tau > \epsilon N$, for some constant ϵ.*

Proof. This is clear by the existence of a sequence of identifiers $\langle \iota_1, ..., \iota_N \rangle$ of the participating nodes in ATILA, that respects the conditions of Lemma 4.1. $\square$

6 Analysis of communication cost in ATILA

Now, we describe a scenario where the message bit cost complexity of ATILA is less than the one of RAMBO and yet the necessary redundancy in the case of direct link failure is provided. Such a scenario can occur in a wide class of mobile systems. The message bit cost complexity is the total cost of sending the individual bits across the links, governed by some cost function.

The RAMBO algorithm involves point-to-point perpetual dissemination of information which eventually helps to infer liveness of the protocol. However, such approach is obviously wasteful when nodes are separated by long geographical distances. We assume that communication within the local area networks is less expensive than in wide area networks. A more reasonable solution to the above problem is to reduce the communication over long distances, hence reducing the total message bit cost.

Consider the following grouping. Let the participants of the service be divided into disjoint groups based on their proximity in terms of cost/reliability of communication among the nodes. For each group we define a non-empty subset to which we refer as the *representatives* of the group. Within a group nodes communicate using the all-to-all gossip protocol, however only the nodes designated as representatives may communicate with other representatives in the different groups. In this setting the indirect learning protocol allows a reduction of message bit cost complexity. (The set of representatives may be agreed upon using an arbitrary consensus service, and handled in a similar fusion as ATILA does the configuration reconfiguration.) Note that in this setting the correctness issues are vacuously satisfied — we only impose a communication policy that restricts certain nodes from sending messages to certain other nodes.

Notation. We denote the set of all nodes that are participating in the service by $\mathcal{U}$ and let $N = |\mathcal{U}|$. Let i and j be any two non-failed participants of the service, hence $i, j \in \mathcal{U}$. The cost function which represents the cost of sending a message between any pair of nodes in $\mathcal{U}$ is defined as $\chi : \mathcal{U} \times \mathcal{U} \to \mathbb{R}^+$. Hence, $\chi(i, j)$ denotes the cost of sending a message from node i to j. We assume that $\chi(i, i) = 0$ and $\chi(i, j) = \chi(j, i)$ and that $\chi(\cdot, \cdot)$ satisfies the triangle inequality. Thus $(\mathcal{U}, \chi)$ is a metric space with the metric χ.

We partition $\mathcal{U}$ into groups $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_m$, such that, $\mathcal{G}_\iota \subseteq \mathcal{U}$, $\cup_{\iota=1}^{m} \mathcal{G}_\iota = \mathcal{U}$ and $\mathcal{G}_\iota \cap \mathcal{G}_{\iota'} = \emptyset$ for $1 \leq \iota \neq \iota' \leq m$. We also require that $\forall i, j \in \mathcal{G}_\iota$, $\chi(i, j) \leq d$ and that for some $1 \leq \iota \neq \iota' \leq m$ there is a pair of nodes $i \in \mathcal{G}_\iota$ and $j \in \mathcal{G}_{\iota'}$ such that $\chi(i, j) > d$, for an appropriately chosen d. Finally, for every group $\mathcal{G}_\iota$ we define a subset $\mathcal{L}_\iota \subseteq \mathcal{G}_\iota$, which we call the *representatives* of $\mathcal{G}_\iota$.

Analysis of message cost. Next, we compare the communication cost complexities of the RAMBO and ATILA and show that the use of indirect gossip can lead to substantial cost savings. Note that the following analysis does not account for the cost per message bit contributed by the maintenance of the overlay network on which RAMBO relies on for message routing. Also, observe that proposed here partitioning is based on the communication cost involved between each pair of nodes and hence is general from the point of view of the distance function. Let $\mathcal{U}$ be partitioned into m groups, as previously described. To simplify the analysis we assume that all groups are of equal size, $|\mathcal{G}_\iota| = g$, and that the size of representative subgroups also has equal size, $|\mathcal{L}_\iota| = \ell$, for all $1 \leq \iota \leq m$.

The gossip messages in RAMBO have the form $\langle tag, val, configs, world, pnum_i, pnum_j \rangle$. Clearly, $|world| = |\mathcal{U}| = N$. Therefore, the size of a message is $\Delta + N \times \delta$, where Δ represent the constant size of the remaining message components and δ is the size of a node identifier. Hence, the size of each message is $O(N)$.

Now we compute the message bit cost complexity of ATILA. We begin by considering the following two cases: First, messages exchanged between a non-representative nodes are of the form $\langle tag, val, configs, world, pNums[i][i], pNums[i][j] \rangle$. Second, messages sent out by a representative node are of the form $\langle tag, val, configs, world, pNums \rangle$. Observe that in the first case the size of a a message is $O(N)$ and in the second case it is $O(N^2)$.

The following equation compares the communication bit complexity per a single round of gossip in ATILA, left hand side, and RAMBO, right hand side.

$$g^2 m(\Delta + \delta N) + \ell \tfrac{m(m-1)}{2}\big(\Delta + \delta(N^2 + N)\big) + \ell(g - \ell)m\big(\Delta + \delta(N^2 + N)\big) \leq N^2(\Delta + \delta N) = O(N^3)$$

On left hand side, the first term is the bit complexity of the messages exchanged inside all of the m groups, second term is the bit complexity of the communication between all representatives, and the third term is the bit complexity of messages exchanged between the representatives and the rest of the group, for each group.

Observe that g, m, and ℓ have the following relationships $m = N/g$ and that $1 \leq \ell \leq g$. Clearly ATILA benefits when ℓ is small with respect to g. Therefore, under the assumption that the cost of communication within a group is cheaper, then if $\ell \leq \log g$ and $m \leq \sqrt{N}$ then the message bit cost complexity is minimized for ATILA, i.e. when the number of groups is not very large and ATILA can take advantage of reducing the number of bits sent over the expensive links – between different groups. Otherwise, RAMBO has the lesser message complexity than ATILA. However, the liveness of the RAMBO depends on the fact that links between the nodes do not fail and messages are not indefinitely delayed.

7 Conclusions

In this work we investigate an indirect learning mechanism within a consistent replicated object service for dynamic networks that do not support automatic routing. We provide an algorithm that implements atomic read/write objects where the participating nodes communicate with their direct neighbors only, thus obviating the need for a global routing protocol. The indirect learning approach, as presented in this work, has the potential of making more robust other algorithms that, for example, employ all-to-all gossip as means for information exchange. The algorithmic development presented here is formally proved to guarantee atomicity in all executions. The indirect learning protocol allows operations to progress as long as the underlying network remains connected. We also presented a novel analysis of the operational latency under reasonable assumptions about the message delivery time. Lastly, we considered scenarios where our algorithm helps reduce messaging costs. A distributed implementation of the algorithm presented here is underway. Experiments with the implementation will provide further insight into the behavior of algorithms using the indirect learning approach and the impact of our approach on communication costs in ad-hoc networks.

References

[1] Special issue on group communication services. *Communications of the ACM*, 39(4), 1996.

[2] J.-C. Bermond, L. Gargano, A. A. Rescigno, and U. Vaccaro. Fast gossiping by short messages. In *Automata, Languages and Programming*, pages 135–146, 1995.

[3] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in ad hoc networks. In *Proc. of 17th International Symposium on Distributed Computing*, pages 306–320, 2003.

[4] C. Georgiou, P. Musial, and A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. In *Proc. of 11th Colloq. on Structural Information and Communication Complexity*, pages 185–196, 2004.

[5] C. Georgiou, P. Musiał, and A. Shvartsman. Developing a consistent domain-oriented distributed object service. In *Proc. 4th IEEE Int-l Symposium on Network Computing and Applications*, pages 149–158, July 2005.

[6] S. Gilbert. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. Master's thesis, MIT, August 2003.

[7] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of International Conference on Dependable Systems and Networks*, pages 259–268, 2003.

[8] V. Gramoli, P. Musiał, and A. Shvartsman. Operation liveness in a dynamic distributed atomic data service with efficient gossip management. In *Proc. 18th International Conference on Parallel and Distributed Computing Systems*, August 2005.

[9] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Kluwer Academic*.

[10] I. Keidar, J. B. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for wans. *ACM Trans. Comput. Syst.*, 20(3):191–238, 2002.

[11] S. Khuller, Y. Kim, and Y. Wan. On generalized gossiping and broadcasting, 2003.

[12] K. Konwar, P. Musial, N. Nicolaou, and A. Shvartsman. Implementing atomic data through indirect learning in dynamic networks, 2005. `http://www.cse.uconn.edu/~piotr/pubs/TRs/KMNS06.ps`.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

[15] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[16] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical report, 1987.

[17] N. Malpani, J. L. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *DIALM '00: Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103. ACM Press, 2000.

[18] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proc. of IEEE INFOCOM*, April 1997.

[19] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Proc. of ACM SIGCOMM*, August 1994.

[20] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of IEEE WMCSA*, February 1999.

Appendix

7.1 A. Atomic Consistency of ATILA

In this section we present the omitted details of proofs of lemmas presented in Section 4.

Definitions. We introduce another operation that allowed on the $CMap$. It is a binary function on $C_{\pm}$, for any $c, c' \in C_{\pm}$, defined by $extend(c, c') = c'$ if $c = \bot$ and $c' \in C$, and $extend(c, c') = c$ otherwise.

Configuration map invariants. Invariants are the properties of the algorithm that are true in every state of any good execution. Here we state two invariants. The first invariant describes the patterns of C, $\bot$, and $\pm$ values that may occur in configuration maps in various places in the system in any state. The variables $upg\text{-}configs$ is defined similarly as $op\text{-}configs$ and is used to maintain the list of configurations used during the configuration upgrade operation.

Invariant 1 [Inv. 4.3.3 in [7]] *Let cm be a $CMap$ that appears as one of the following: (i) The cm component of some message in in-transit. (ii) $configs_i$ for any $i \in I$. (iii) $op\text{-}configs_i$ for some $i \in I$ that has initiated a read/write operations which has not terminated yet. (iv) $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π. (v) $upg\text{-}configs_i$ for some $i \in I$ that initiated configuration upgrade operation which has not terminated yet. Then $cm \in Usable$.*

Invariant 1 ensures that the configuration map in each of the listed places has the $Usable$ property, which describes the patten of configurations. The next invariant strengthens Invariant 1 and states additional properties of the *CMap*s that are used for read and write operations.

Invariant 2 [Inv. 4.3.4 in [7]] *Let cm be a $CMap$ that appears as $op\text{-}configs_i$ for some $i \in I$ that has initiated a read/write operations which has not terminated yet, or as $query\text{-}cmap(\pi)$ or $prop\text{-}cmap(\pi)$ for any operation π. Then $cm \in Truncated$.*

Invariant 2 ensures that the configuration map used during read and write operations has no gaps in it, i.e. has the $Truncated$ property. Upon detection of a gap in the local configuration map, the operation is restarted as to take advantage of the new configuration information.

Omitted proofs of referenced Lemmas.

Lemma 7.1 *Suppose that a* cfg-upg-query-fix$(k)_i$ *event for configuration upgrade operation γ occurs in α and $k' \in removal\text{-}set(\gamma)$. Suppose $j \in R(\gamma, k') \cup W_1(\gamma, k')$.*
Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that:

1. *The message m_{ι_1,ι_2} is sent after the* cfg-upgrade$(k)_i$ *event of γ.*
2. *Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received.*
3. *The message m_{ι_{n-1},ι_n} is received before the* cfg-upg-query-fix$(k)_i$ *event of γ.*
4. *In any state after j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, $configs(\ell)_j \neq \bot$ for all $\ell \leq k$.*
5. *$tag(\gamma) \geq t$, where t is the value of tag_j in any state before j sends message $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$.*

Proof. The phase number discipline implies the existence of the claimed sequence of messages $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$.

For Part 4, individually consider each h in the range $2 \leq h \leq n$. The precondition of cfg-upgrade(k) implies that, when the cfg-upgrade$(k)_i$ event of γ occurs, $configs(\ell)_i \neq \bot$ for all $\ell \leq k$. Therefore, each node whose identifier is found in the sequence $\langle \iota_2, \ldots, \iota_n \rangle$, which includes $\iota_{\hat{h}} = j$, sets $configs(\ell)_j \neq \bot$ for all $\ell \leq k$ when it receives the message m_{ι_{h-1},ι_h}. Monotonicity of $configs_h$, for each $1 \leq h \leq n$ including j, ensures that this property persists forever.

For Part 5, consider each h in the range $1 \leq h \leq n-1$. Let t_{ι_h} be the value of tag_{ι_h} in any state before ι_h sends message $m_{\iota_h,\iota_{h+1}}$. Let t'_{ι_h} be the value of tag_{ι_h} in the state just after ι_h sends $m_{\iota_{\iota_h},\iota_{\iota_h}+1}$. Then $t_{\iota_h} \leq t'_{\iota_h}$, by monotonicity. Hence, $t_{\iota_1} \leq t'_{\iota_{n-1}}$. The tag component of m_{ι_{n-1},ι_n} is equal to $t'_{\iota_{n-1}}$, by the code for send. Since i receives this message before the cfg-upg-query-fix$(k)_i$, it follows that $tag(\gamma)$ is set by i to a value $\geq t$. □

Next, we consider the propagation phase of a configuration upgrade.

Lemma 7.2 *Suppose that a* cfg-upg-prop-fix$(k)_i$ *event for a configuration upgrade operation γ occurs in α. Suppose that $j \in W_2(\gamma)$.*
Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that:

1. *The message m_{ι_1,ι_2} is sent after the* cfg-upg-query-fix$(k)_i$ *event of γ.*
2. *Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received.*
3. *The message m_{ι_{n-1},ι_n} is received before the* cfg-upg-prop-fix$(k)_i$ *event of γ.*

4. *In any state after j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, $tag_j \geq tag(\gamma)$.*

Proof. The phase number discipline implies the existence of the claimed sequence of messages $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$.

For Part 4, when j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, it sets tag_j to be $\geq tag(\gamma)$. Monotonicity of tag_j ensures that this property persists in later states. □

Next, we consider the query phase of read/write operations.

Lemma 7.3 *Suppose that a* query-fix$_i$ *event for a read or write operation π occurs in α. Leg $k, k' \in \mathbb{N}$. Suppose $query\text{-}cmap(\pi)(k) \in C$ and $j \in R(\pi, k)$.*
Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that:

1. *The message m_{ι_1,ι_2} is sent after the* query-phase-start(π) *event.*
2. *Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received.*
3. *The message m_{ι_{n-1},ι_n} is received before the* query-fix *event of π.*
4. *If t is the value of the tag_j in any state before j sends $m'_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$, then:*
   (a) $tag(\pi) \geq t$.
   (b) If π is a write operation then $tag(\pi) > t$.
5. *If $configs(\ell)_j \neq \bot$ for all $\ell \leq k'$ in any state before j send $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$, then $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'$.*

Proof. The phase number discipline implies the existence of the claimed sequence of messages $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$.

For Part 4, individually consider each h in the range $1 < h < n$. The tag component of message $m_{\iota_h,\iota h+1}$ is at least as great as the tag component in the message $m_{\iota_{h-1},\iota h}$. Hence, in the message m_{ι_{n-1},ι_n} and during the query phase of π node i receives a tag $\geq t$. Therefore, $tag(\pi) \geq t$. Also, if π is a write, the effects of the query-fix imply that $tag(\pi) > t$.

Finally, we show Part 5. In the cm component of message $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$, $cm(\ell) \neq \bot$ for all $\ell \leq k'$. Then by the code of recv code each h, where $\hat{h} < h < n$, sets its $configs(\ell)_h \neq \bot$ for all $\ell \leq k'$, from the property of $configs_{h-1}$ and the code of send action. Hence, we conclude that cm component of message m_{ι_{n-1},ι_n} has $cm(\ell) \neq \bot$ for all $\ell \leq k'$. Therefore, $truncate(cm)(\ell) = cm(\ell)$ for all $\ell \leq k'$, so $truncate(cm) \neq \bot$ for all $\ell \leq k'$.

Let cm' be the configuration map $extend(op.configs_i, truncate(cm))$ computed by i during the effects of the recv event for m_{ι_{n-1},ι_n}. Since i does not reset $op.acc$ to $\emptyset$ in this step, by definition of the query-phase-start(π) event, it follows that $cm' \in Truncated$, and cm' is the value of $op.configs_i$ just after the recv step.

Fix ℓ, $0 \leq \ell \leq k'$. We claim that $cm'(\ell) \neq \bot$. We consider cases:

1. $op.configs(\ell)_i \neq \bot$ just before the recv step. Then the definition of $extend$ implies that $cm' \neq \bot$, as needed.
2. $op.configs(\ell)_i = \bot$ just before the recv step and $truncate(cm)(\ell) \in C$. Then the definition of $extend$ implies that $cm'(\ell) \in C$, which implies that $cm'(\ell) \neq \bot$, as needed.
3. $op.configs(\ell)_i = \bot$ just before the recv step and $truncate(cm)(\ell) \notin C$. Since $truncate(cm))(\ell) \neq \bot$, it follows that $truncate(cm)(\ell) \notin C$. By the case assumption, $op.configs(\ell)_i = \bot$ just before the recv step. Since by Invariant 2, $op.configs_i \in Truncated$, it follows that $op.configs(\ell') = \bot$ before the recv step. Then by definition of $extend$, we have that $cm'(\ell) = \bot$ while $cm'(\ell) \in C$. This implies that $cm' \notin Truncated$, which contradicts the fact, already shown,that $cm' \in Truncated$. So this case cannot arise.

Since this argument holds for all ℓ, $0 \leq \ell \leq k'$, it follows that $cm'(\ell) \neq \bot$ for all $\ell \leq k'$. Since $cm'(\ell) \neq \bot$ for all $\ell \leq k'$, Invariant 1 implies that $cm' \in Usable$, which implies by definition of $Usable$ that $cm'(\ell) \in C$ for some $\ell \geq k'$. That is, $op.configs_i(\ell) \in C$ for some $\ell \geq k'$ immediately after the recv step. This implies that $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'$, as needed. □

And finally, we consider the propagation phase of read and write operations.

Lemma 7.4 *Suppose that a* prop-fix$_i$ *event for a read or a write operation π occurs in α. Suppose $prop\text{-}cmap(\pi)(k) \in C$ and $j \in W(\pi, k)$.*
Then there exists a sequence of identifiers $\langle \iota_1, ..., \iota_n \rangle$ where for all $1 \leq h \leq n$ each $\iota_h \in I$, and the corresponding message sequence $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$, where $\iota_1 = \iota_n = i$ and that there is $\iota_{\hat{h}} = j$, for some $1 < \hat{h} < n$. Such that:

1. *The message m_{ι_1,ι_2} is sent after the* v-phase-start(π) *event.*
2. *Each message $m_{\iota_h,\iota_{h+1}}$ is sent after m_{ι_{h-1},ι_h} is received.*

3. *The message* m_{ι_{n-1},ι_n} *is received before the* prop-fix *event of* π.
4. *In any state after* j *receives* $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, $tag_j \geq tag(\pi)$.
5. *If* $configs(\ell)_j \neq \perp$ *for all* $\ell < k'$ *in any state before* j *sends* $m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}$*, then* $prop\text{-}cmap(\pi)(\ell) \in C$ *for some* $\ell \geq k'$.

Proof. The phase number discipline implies the existence of the claimed sequence of messages $\left\langle m_{\iota_1,\iota_2}, \ldots, m_{\iota_{\hat{h}},\iota_{\hat{h}+1}}, \ldots, m_{\iota_{n-1},\iota_n} \right\rangle$.

For Part 4, individually consider each h in the range $1 < h < n$. Let t_h be the value of a tag at node h just before h receives m_{ι_{h-1},ι_h} and t'_h after h received m_{ι_{h-1},ι_h}. From the code of recv we know that $t'_h \geq t_h$. It is easy to see that $t'_n \geq t_1$, hence $t'_{\iota_{\hat{h}}} \geq t_1$. Let $m_{\iota_1,\iota_2}.tag$ be the tag field of message m_{ι_1,ι_2}. Since m_{ι_1,ι_2} is sent after the prop-phase-start(π) event, which is not earlier than the query-fix$_i$, it must be that $m_{\iota_1,\iota_2}.tag \geq tag(\pi)$. Therefore, by the effects of the recv, just after j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$, $tag_j \geq m_{\iota_1,\iota_2}.tag \geq tag(\pi)$. Then monotonicity of tag_j implies that $tag_j \geq tag(\pi)$ in any state after j receives $m_{\iota_{\hat{h}-1},\iota_{\hat{h}}}$.

For Part 5, the proof is analogous to the proof of part 5 of Lemma 7.3. In fact, it is identical except for the final conclusion, which now says that $prop\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k'$. □

Using the above lemmas in conjunction with those presented in [7, 4] we arrive at the main result of this work.

Theorem 7.5 ATILA *implements atomic read/write objects.*

Proof.[**(sketch)**] Follows that of Theorem 5.4.3 of [6], where the above Lemmas 7.1, 7.2, 7.3, and 7.4 are used in place of Lemmas 4.4.1, 4.4.2, 4.4.3, and 4.4.4 in [6] respectively. □

7.2 B. Complete Specification of ATILA

In this section we present the complete code listing of ATILA algorithm, which includes the following published improvements [7, 4, 5]. Recall that in [7] a new rapid reconfiguration service is proposed that allows removal of multiple configurations during a single configuration upgrade operation. In [4] a long-lived version of the RAMBO service is presented, where explicit leave protocol and incremental gossip mechanism improve performance of the service by substantially reducing the number and size of state messages exchanged by the *Reader-Writer* automata. Finally, an efficient implementation of a multi object RAMBO service is presented in [5]. The user groups all of the related objects into a domain, which is maintained by a single instance of the RAMBO algorithm per participating node. Note that the same techniques used to extend RAMBO to the domain-RAMBO are used to extend specification of ATILA to the domain-ATILA. Also, the methods used to show that domain-RAMBO implements atomic read/write objects can be used to show that the same is true of domain-ATILA.

The IOA specification of ATILA components is in the following order: (i) first we present the *Joiner* component, (ii) *Reader-Writer* component follows, and (iii) we conclude with the specification of the *Recon* component.

Domains:

I, a set of processes
D, a set of domains
X_d, a set of object identifiers from domain d, where $d \in D$
$V_{d,x}$, a set of legal values of object x from domain d, where $x \in X_d$ and $d \in D$
C, a set of configurations, each consisting of members, read-quorums, and write-quorums

Input:

$\mathsf{join}(\mathsf{rambo}, J)_{d,i}$, J a finite subset of $I - \{i\}$, $i \in I$, such that if $i = i_0$ then $J = \emptyset$, $d \in D$
$\mathsf{read}(x)_{d,i}$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{write}(x, v)_{d,i}$, $v \in V$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{recon}(c, c')_{d,i}$, $c, c' \in C$, $i \in members(c)$, $i \in I$, $d \in D$
$\mathsf{leave}_{d,i}$, $i \in I$, $d \in D$
$\mathsf{fail}_{d,i}$, $i \in I$, $d \in D$

Output:

$\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i}$, $i \in I$, $d \in D$
$\mathsf{read\text{-}ack}(x, v)_{d,i}$, $v \in V$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{write\text{-}ack}(x)_{d,i}$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{recon\text{-}ack}(b)_{d,i}$, $b \in \{ok, nok\}$, $i \in I$, $d \in D$
$\mathsf{report}(c)_{d,i}$, $c \in C$, $i \in I$, $d \in D$

Figure 3. RAMBO$_d$: External signature.

Signature:

Input:
```
  $\mathsf{join}(\mathsf{rambo}, J)_{d,i}$, $J$ a finite subset of $I - \{i\}$, $d \in D$
  $\mathsf{join\text{-}ack}(r)_{d,i}$, $r \in \{\mathsf{recon}, \mathsf{rw}\}$, $d \in D$
  $\mathsf{leave}_{d,i}$, $d \in D$
  $\mathsf{fail}_{d,i}$, $d \in D$
```

Output:
```
  $\mathsf{send}(\mathsf{join})_{d,i,j}$, $j \in I - \{i\}$, $d \in D$
  $\mathsf{join}(r)_{d,i}$, $r \in \{\mathsf{recon}, \mathsf{rw}\}$, $d \in D$
  $\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i}$, $d \in D$
```

State:

$status \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially idle

$child\text{-}status \in \{\mathsf{recon}, \mathsf{rw}\} \rightarrow \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially everywhere idle

$hints \subseteq I$, initially $\emptyset$

$failed$, a Boolean, initially $false$

Transitions:

```
Input $\mathsf{join}(\mathsf{rambo}, J)_{d,i}$
Effect:
    if $\neg failed$ then
      if $status = \mathsf{idle}$ then
        $status \leftarrow \mathsf{joining}$
        $hints \leftarrow J$

Input $\mathsf{join\text{-}ack}(r)_{d,i}$
Effect:
    if $\neg failed$ then
      if $status = \mathsf{joining}$ then
        $child\text{-}status(r) \leftarrow \mathsf{active}$

Input $\mathsf{leave}_{d,i}$
Effect:
    $failed \leftarrow \mathsf{true}$

Input $\mathsf{fail}_{d,i}$
Effect:
    $failed \leftarrow \mathsf{true}$
```

```
Output $\mathsf{join}(r)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{joining}$
    $child\text{-}status(r) = \mathsf{idle}$
Effect:
    $child\text{-}status(r) \leftarrow \mathsf{joining}$

Output $\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{joining}$
    $\forall r \in \{\mathsf{recon}, \mathsf{rw}\} : child\text{-}status(r) = \mathsf{active}$
Effect:
    $status \leftarrow \mathsf{active}$

Output $\mathsf{send}(\mathsf{join})_{d,i,j}$
Precondition:
    $\neg failed$
    $status = \mathsf{joining}$
    $j \in hints$
Effect:
    none
```

Figure 4. $Joiner_{d,i}$**: Signature, state, and transitions**

Signature:

Input:

- $\mathsf{read}(x)_{d,i}$, $x \in X_d$, $d \in D$
- $\mathsf{write}(x, v)_{d,i}$, $v \in V$, $x \in X_d$, $d \in D$
- $\mathsf{new\text{-}config}(c, k)_{d,i}$, $c \in C, k \in \mathbb{N}^+$, $d \in D$
- $\mathsf{recv}(\mathsf{join})_{d,j,i}$, $j \in I - \{i\}$, $d \in D$
- $\mathsf{recv}(m_x)_{d,j,i}$, $m \in M$, $j \in I$, $x \in X_d$, $d \in D$
- $\mathsf{join}(\mathsf{rw})_{d,i}$, $d \in D$
- $\mathsf{leave}_{d,i}$, $d \in D$
- $\mathsf{fail}_{d,i}$, $d \in D$

Output:

- $\mathsf{join\text{-}ack}(\mathsf{rw})_{d,i}$, $d \in D$
- $\mathsf{read\text{-}ack}(x, v)_{d,i}$, $v \in V$, $x \in X_d$, $d \in D$
- $\mathsf{write\text{-}ack}(x)_{d,i}$, $x \in X_d$, $d \in D$
- $\mathsf{send}(m_x)_{d,i,j}$, $m \in M$, $j \in I$, $x \in X_d$, $d \in D$

Internal:

- $\mathsf{query\text{-}fix}(x)_{d,i}$, $x \in X_d$, $d \in D$
- $\mathsf{prop\text{-}fix}(x)_{d,i}$, $x \in X_d$, $d \in D$
- $\mathsf{cfg\text{-}upgrade}(k)_{d,i}$, $k \in \mathbb{N}^+$, $d \in D$
- $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_{d,i}$, $k \in \mathbb{N}$, $d \in D$
- $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_{d,i}$, $k \in \mathbb{N}$, $d \in D$
- $\mathsf{cfg\text{-}upgrade\text{-}ack}(k)_{d,i}$, $k \in \mathbb{N}$, $d \in D$

State:

$\mathit{status} \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially idle
world, a finite subset of I, initially $\emptyset$
$\mathit{leave\text{-}world}$, a finite subset of I, initially $\emptyset$
$\mathit{departed}$, a finite subset of I, initially $\emptyset$
$\mathit{value}(x) \in V_x$, $x \in X_d$, initially $\forall x \in X_d : \mathit{value}(x) = (v_0)_x$
$\mathit{tag} \in X \rightarrow T$, initially $\forall x \in X_d : \mathit{tag}(x) = (0, i_0)$
$\mathit{configs} \in \mathit{CMap}$, initially $\mathit{configs}(0) = c_0$, $\mathit{configs}(k) = \perp$ for $k \geq 1$
$\mathit{igpnum1} \in \mathbb{N}$, initially 0
$\mathit{igpnum2} \in I \times I \rightarrow \mathbb{N}$, initially everywhere 0
$\mathit{pnum1} \in X_d \rightarrow \mathbb{N}$, initially $\forall x \in X_d : \mathit{pnum1}(x) = 0$
$\mathit{pnum2} \in I \times I \times X_d \rightarrow \mathbb{N}$, initially $\forall x \in X_d, \forall j, k \in I$, where $j \neq i \wedge k \neq i : \mathit{pnum2}(j, k, x) = 0$
failed, a Boolean, initially false

op(x), an array of records (one for each object $x \in X_d$) with fields:

- $\mathit{type} \in \{\mathsf{read}, \mathsf{write}\}$
- $\mathit{phase} \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}, \mathsf{done}\}$, initially idle
- $\mathit{pnum} \in \mathbb{N}$
- $\mathit{configs} \in \mathit{CMap}$
- acc, a finite subset of I
- $\mathit{value} \in V_x$

upg, a record with fields:

- $\mathit{phase} \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}\}$, initially idle
- $\mathit{pnum}(x) \in \mathbb{N}, \forall x \in X_d : \mathit{pnum}(x) = 0$
- $\mathit{configs} \in \mathit{CMap}$
- $\mathit{acc}(x)$, a finite subset of I, $\forall x \in X_d$
- $\mathit{target} \in \mathbb{N}$

$\mathit{ig} \in \mathit{IGMap}$, initially $\forall k \in I$:

- $\mathit{ig}(k).\mathit{w\text{-}known} = \emptyset$
- $\mathit{ig}(k).\mathit{w\text{-}unack} = \emptyset$
- $\mathit{ig}(k).\mathit{d\text{-}known} = \emptyset$
- $\mathit{ig}(k)_k.\mathit{d\text{-}unack} = \emptyset$
- $\mathit{ig}(k).\mathit{p\text{-}ack} = 0$

Figure 5. *Reader-Writer*$_{d,i}$**: Signature and state**

```
Input join(rw)$_{d,i}$
Effect:
    if $\neg failed$ then
      if $status = \mathsf{idle}$ then
        if $i = i_0$ then
          $status \leftarrow \mathsf{active}$
        else
          $status \leftarrow \mathsf{joining}$
        $world \leftarrow world \cup \{i\}$

Input recv(join)$_{d,j,i}$
Effect:
    if $\neg failed$ then
      if $status \neq \mathsf{idle}$ then
        $world \leftarrow world \cup \{j\}$

Input fail$_{d,i}$
Effect:
    $failed \leftarrow \mathsf{true}$

Output join-ack(rw)$_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
Effect:
    none
```

Figure 6. *Reader-Writer*$_{d,i}$**: Join-related and failure transitions**

```
Output send($\langle W, D, obj, v, t, cm, igns, ignr, pnc\rangle$)$_{d,i,j}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $x \in X_d$
    $j \in (world - departed)$
    $W = world - ig(j).w\text{-}known$
    $D = departed - ig(j).d\text{-}known$
    $\langle obj, v, t\rangle =$
      $\langle x, value(x), tag(x,j)\rangle$
    $\langle cm, igns, ignr, pnc\rangle =$
      $\langle configs, igpnum1(x), igpnum2(x,j), pnum2\rangle$
Effect:
    $igpnum1 \leftarrow igpnum1 + 1$

Input recv(leave)$_{d,i,j}$
Effect:
    if $\neg failed \wedge status = \mathsf{active}$ then
      $departed \leftarrow departed \cup \{j\}$

Output send(leave)$_{d,i,j}$
Precondition:
    $j \in leave\text{-}world$
Effect:
    $leave\text{-}world \leftarrow leave\text{-}workd - \{j\}$

Input recv($\langle W, D, obj, v, t, cm, igns, ignr, pnc\rangle$)$_{d,j,i}$
Effect:
    if $\neg failed \wedge status \neq \mathsf{idle}$ then
      $status \leftarrow \mathsf{active}$
      $world \leftarrow world \cup W$
      $departed \leftarrow departed \cup D$
      $pnum2 \leftarrow \max(pnum2, pnc)$
      $ig(j).w\text{-}known \leftarrow ig(j).w\text{-}known \cup W$
      $ig(j).w\text{-}unack \leftarrow ig(j).w\text{-}unack - W$
      $ig(j).d\text{-}known \leftarrow ig(j).d\text{-}known \cup D$
      $ig(j).d\text{-}unack \leftarrow ig(j).d\text{-}unack - D$
      if $ignr > ig(j).p\text{-}ack$ then
        $ig(j).w\text{-}known \leftarrow$
            $ig(j).w\text{-}known \cup ig(j).w\text{-}unack$
        $ig(j).w\text{-}unack \leftarrow world - ig(j).w\text{-}known$
        $ig(j).d\text{-}known \leftarrow$
            $ig(j).d\text{-}known \cup ig(j).d\text{-}unack$
        $ig(j).d\text{-}unack \leftarrow departed - ig(j).d\text{-}known$
        $ig(j).p\text{-}ack \leftarrow igpnum1$
      if $t > tag(obj)$ then
        $(value(obj), tag(obj)) \leftarrow (v, t)$
      $configs \leftarrow update(configs, cm)$
      for $k \in world \wedge x \in X_d$ do
        $pnum2(i,k,x) \leftarrow \max(pnum2(\cdot,k,x))$
        if $op(x).phase \in \{\mathsf{query}, \mathsf{prop}\}$ then
          if $pnum2(k,i,x) \geq op(x).pnum$ then
            $op(x).configs \leftarrow$
              $extend(op(x).configs, truncate(cm))$
            if $op(x).configs \in Truncated$ then
              $op(x).acc \leftarrow op(x).acc \cup \{j\}$
            else
              $pnum1(x) \leftarrow pnum1(x) + 1$
              $op(x).acc \leftarrow \emptyset$
              $op(x).configs \leftarrow truncate(configs)$
        if $upg.phase \in \{\mathsf{query}, \mathsf{prop}\}$ then
          if $pnum2(k,i,x) \geq upg.pnum(x)$ then
            $upg.acc(obj) \leftarrow upg.acc(x) \cup \{k\}$
```

Figure 7. *Reader-Writer*$_i$**: Transitions of send and receive actions**

```
Input $\mathsf{leave}_{d,i}$
Effect:
    if $\neq$ $failed$ then
      $failed \leftarrow \mathsf{true}$
      $departed \leftarrow departed - \{i\}$
      $leave\text{-}world \leftarrow world - departed$

Input $\mathsf{new\text{-}config}(c,k)_{d,i}$
Effect:
    if $\neg failed \wedge status \neq \mathsf{idle}$ then
      $configs(k) \leftarrow update(configs(k), c)$

Input $\mathsf{read}(x)_{d,i}$
Effect:
    if $\neg failed \wedge status \neq \mathsf{idle}$ then
      $pnum1(x) \leftarrow pnum1(x) + 1$
      $op(x).pnum \leftarrow pnum1(x)$
      $op(x).type \leftarrow \mathsf{read}$
      $op(x).phase \leftarrow \mathsf{query}$
      $op(x).cmp \leftarrow truncate(cmap)$
      $op(x).acc \leftarrow \emptyset$

Input $\mathsf{write}(x,v)_{d,i}$
Effect:
    if $\neg failed \wedge status \neq \mathsf{idle}$ then
      $pnum1(x) \leftarrow pnum1(x) + 1$
      $op(x).pnum \leftarrow pnum1(x)$
      $op(x).type \leftarrow \mathsf{write}$
      $op(x).phase \leftarrow \mathsf{query}$
      $op(x).cmp \leftarrow truncate(cmap)$
      $op(x).acc \leftarrow \emptyset$
      $op(x).value \leftarrow v$

Internal $\mathsf{restart}(x)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $op(x).phase \neq \mathsf{idle}$
Effect:
    $pnum1(x) \leftarrow pnum1(x) + 1$
    $op(x).pnum \leftarrow pnum1(x)$
    $op(x).configs \leftarrow truncate(configs)$
    $op(x).acc \leftarrow \emptyset$

Internal $\mathsf{query\text{-}fix}(x)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $op(x).type \in \{\mathsf{read}, \mathsf{write}\}$
    $op(x).phase = \mathsf{query}$
    $\forall k \in \mathbb{N}, c \in C : (op(x).configs(k) = c)$
      $\Rightarrow (\exists R \in read\text{-}quorums(c) : R \subseteq op(x).acc)$
Effect:
    if $op(x).type = \mathsf{read}$ then
      $op(x).value \leftarrow value(x)$
    else
      $value(x) \leftarrow op(x).value$
      $tag(x) \leftarrow \langle tag(x).seq + 1, i \rangle$
    $pnum1(x) \leftarrow pnum1(x) + 1$
    $op(x).pnum \leftarrow pnum1(x)$
    $op(x).phase \leftarrow \mathsf{prop}$
    $op(x).configs \leftarrow truncate(configs)$
    $op(x).acc \leftarrow \emptyset$

Internal $\mathsf{prop\text{-}fix}(x)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $op(x).type \in \{\mathsf{read}, \mathsf{write}\}$
    $op(x).phase = \mathsf{prop}$
    $\forall k \in \mathbb{N}, c \in C : (op(x).configs(k) = c)$
      $\Rightarrow (\exists W \in write\text{-}quorums(c) : W \subseteq op(x).acc)$
Effect:
    $op(x).phase = \mathsf{done}$

Output $\mathsf{read\text{-}ack}(x,v)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $op(x).type = \mathsf{read}$
    $op(x).phase = \mathsf{done}$
    $v = op(x).value$
Effect:
    $op(x).phase = \mathsf{idle}$

Output $\mathsf{write\text{-}ack}(x)_{d,i}$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $op(x).type = \mathsf{write}$
    $op(x).phase = \mathsf{done}$
Effect:
    $op(x).phase = \mathsf{idle}$
```

Figure 8. *Reader-Writer*$_i$**: Transitions pertaining to read/write operations and to leave and new configuration notification actions**

```
Internal cfg-upgrade$(k)_{d,i}$
Precondition:
    $\neg failed$
    $status = \text{active}$
    $upg.phase = \text{idle}$
    $configs(k) \in C$
    $\forall l \in \mathbb{N}, l < k : configs(l) \neq \bot$
Effect:
    for all $x \in X_d$ do
      $pnum1(x) \leftarrow pnum1(x) + 1$
      $upg.pnum(x) \leftarrow pnum1(x)$
      $upg.acc(x) \leftarrow \emptyset$
    $upg.phase \leftarrow \text{query}$
    $upg.configs \leftarrow configs$
    $upg.target \leftarrow k$

Internal cfg-upgrade-ack$(k)_{d,i}$
Precondition:
    $\neg failed$
    $status = \text{active}$
    $upg.target = k$
    $\forall l \in \mathbb{N}, l < k : configs(l) = \pm$
Effect:
    $upg.phase = \text{idle}$

Internal cfg-upg-query-fix$(k)_{d,i}$
Precondition:
    $\neg failed$
    $status = \text{active}$
    $upg.phase = \text{query}$
    $upg.target = k$
    $\forall l \in \mathbb{N}, l < k : upg.configs(l) \in C$
      $\Rightarrow \exists R \in read\text{-}quorums(upg.configs(l)) :$
        $\exists W \in write\text{-}quorums(upg.configs(l)) :$
        $R \cup W \subseteq upg.acc(x), \forall x \in X_d$
Effect:
    for all $x \in X_d$ do
      $pnum1(x) \leftarrow pnum1(x) + 1$
      $upg.pnum(x) \leftarrow pnum1(x)$
      $upg.acc(x) \leftarrow \emptyset$
    $upg.phase \leftarrow \text{prop}$

Internal cfg-upg-prop-fix$(k)_{d,i}$
Precondition:
    $\neg failed$
    $status = \text{active}$
    $upg.phase = \text{prop}$
    $upg.target = k$
    $\exists W \in write\text{-}quorums(upg.configs(k+1)) :$
      $W \subseteq upg.acc, \forall x \in X_d$
Effect:
    for $l \in \mathbb{N} : l < k$ do
      $configs(l) \leftarrow \pm$
```

Figure 9. *Reader-Writer*$_{d,i}$: **Configuration-Management transitions**

Input:	Output:
init$(v)_{d,k,c,i}$, $v \in V$, $i \in members(c)$, $d \in D$	decide$(v)_{d,k,c,i}$, $v \in V$, $i \in members(c)$, $d \in D$
leave$_{d,i}$, $i \in members(c)$, $d \in D$	
fail$_{d,i}$, $i \in members(c)$, $d \in D$	

Figure 10. *Cons*(k, c, d): **External signature**

Input:	Output:
join(recon)$_{d,i}$, $i \in I$, $d \in D$	join-ack(recon)$_{d,i}$, $i \in I$, $d \in D$
recon$(c, c')_{d,i}$, $c, c' \in C$, $i \in members(c)$, $d \in D$	recon-ack$(b)_{d,i}$, $b \in \{\text{ok}, \text{nok}\}$, $i \in I$, $d \in D$
leave$_i$, $i \in I$, $d \in D$	report$(c)_{d,i}$, $c \in C$, $i \in I$, $d \in D$
fail$_i$, $i \in I$, $d \in D$	new-config$(c, k)_{d,i}$, $c \in C$, $k \in \mathbb{N}^+$, $i \in I$, $d \in D$

Figure 11. *Recon*$_{d,i}$: **External signature**

Signature:

Input:

- $\mathsf{join}(\mathsf{recon})_{d,i}$, $d \in D$
- $\mathsf{recon}(c, c')_{d,i}$, $c, c' \in C, i \in \mathit{members}(c)$, $d \in D$
- $\mathsf{decide}(c)_{k,d,i}$, $c \in C, k \in \mathbb{N}^+$, $d \in D$
- $\mathsf{recv}(\langle \mathsf{config}, c, k\rangle)_{d,j,i}$, $c \in C$, $k \in \mathbb{N}^+$, $i \in \mathit{members}(c)$, $j \in I - \{i\}$, $d \in D$
- $\mathsf{recv}(\langle \mathsf{init}, c, c', k\rangle)_{d,j,i}$, $c, c' \in C$, $k \in \mathbb{N}^+$, $i, j \in \mathit{members}(c)$, $j \neq i$, $d \in D$
- $\mathsf{leave}_{d,i}$, $d \in D$
- $\mathsf{fail}_{d,i}$, $d \in D$

Output:

- $\mathsf{join\text{-}ack}(\mathsf{recon})_{d,i}$, $d \in D$
- $\mathsf{new\text{-}config}(c, k)_{d,i}$, $c \in C, k \in \mathbb{N}^+$, $d \in D$
- $\mathsf{init}(c, c')_{d,k,i}$, $c, c' \in C, k \in \mathbb{N}^+$, $i \in \mathit{members}(c)$, $d \in D$
- $\mathsf{recon\text{-}ack}(b)_{d,i}$, $b \in \{\mathsf{ok}, \mathsf{nok}\}$, $d \in D$
- $\mathsf{report}(c)_{d,i}$, $c \in C$, $d \in D$
- $\mathsf{send}(\langle \mathsf{config}, c, k\rangle)_{d,i,j}$, $c \in C$, $k \in \mathbb{N}^+$, $j \in \mathit{members}(c) - \{i\}$, $d \in D$
- $\mathsf{send}(\langle \mathsf{init}, c, c', k\rangle)_{d,i,j}$, $c, c' \in C$, $k \in \mathbb{N}^+$, $i, j \in \mathit{members}(c)$, $j \neq i$, $d \in D$

State:

$\mathit{status} \in \{\mathsf{idle}, \mathsf{active}\}$, initially idle.
$\mathit{rec\text{-}cmap} \in \mathit{CMap}$, initially $\mathit{rec\text{-}cmap}(0) = c_0$ and $\mathit{rec\text{-}cmap}(k) = \bot$ for all $k \neq 0$.
$\mathit{did\text{-}new\text{-}config} \subseteq \mathbb{N}^+$, initially $\emptyset$
$\mathit{reported} \subseteq C$, initially $\emptyset$

$\mathit{op\text{-}status} \in \{\mathsf{idle}, \mathsf{active}\}$, initially idle
$\mathit{op\text{-}outcome} \in \{\mathsf{ok}, \mathsf{nok}, \bot\}$, initially $\bot$
$\mathit{cons\text{-}data} \in (\mathbb{N}^+ \to (C \times C))$, initially everywhere $\bot$
$\mathit{did\text{-}init} \subseteq \mathbb{N}^+$, initially $\emptyset$
failed, a Boolean, initially false

Figure 12. $\mathit{Recon}_{d,i}$**: Signature and state**

```
Input $\mathsf{join}(\mathsf{recon})_{d,i}$
Effect:
  if $\neg failed \wedge status = \mathsf{idle}$ then
    $status \leftarrow \mathsf{active}$

Output $\mathsf{join\text{-}ack}(\mathsf{recon})_{d,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
Effect:
  none

Output $\mathsf{new\text{-}config}(c,k)_{d,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $rec\text{-}cmap(k) = c$
  $k \notin did\text{-}new\text{-}config$
Effect:
  $did\text{-}new\text{-}config \leftarrow did\text{-}new\text{-}config \cup \{k\}$

Output $\mathsf{send}(\langle \mathsf{config}, c, k\rangle)_{d,i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $rec\text{-}cmap(k) = c$
Effect:
  none

Input $\mathsf{recv}(\langle \mathsf{config}, c, k\rangle)_{d,j,i}$
Effect:
  if $\neg failed \wedge status = \mathsf{active}$ then
    $rec\text{-}cmap(k) \leftarrow c$

Output $\mathsf{report}(c)_{d,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $c = rec\text{-}cmap(k)$
  $\forall \ell > k : rec\text{-}cmap(\ell) = \bot$
  $c \notin reported$
Effect:
  $reported \leftarrow reported \cup \{c\}$

Input $\mathsf{recon}(c, c')_{d,i}$
Effect:
  if $\neg failed \wedge status = \mathsf{active}$ then
    $op\text{-}status \leftarrow \mathsf{active}$
    let $k = \max(\{\ell : rec\text{-}cmap(\ell) \in C\})$
    if $c = rec\text{-}cmap(k) \wedge cons\text{-}data(k+1) = \bot$ then
      $cons\text{-}data(k+1) \leftarrow \langle c, c'\rangle$
      $op\text{-}outcome \leftarrow \bot$
    else
      $op\text{-}outcome \leftarrow \mathsf{nok}$

Output $\mathsf{init}(c')_{d,k,c,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $cons\text{-}data(k) = \langle c, c'\rangle$
  if $k \geq 1$ then $k - 1 \in did\text{-}new\text{-}config$
  $k \notin did\text{-}init$
Effect:
  $did\text{-}init \leftarrow did\text{-}init \cup \{k\}$

Output $\mathsf{send}(\langle \mathsf{init}, c, c', k\rangle)_{d,i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $cons\text{-}data(k) = \langle c, c'\rangle$
  $k \in did\text{-}init$
Effect:
  none

Input $\mathsf{recv}(\langle \mathsf{init}, c, c', k\rangle)_{d,j,i}$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      if $rec\text{-}cmap(k-1) = \bot$ then
        $rec\text{-}cmap(k-1) \leftarrow c$
      if $cons\text{-}data(k) = \bot$ then
        $cons\text{-}data(k) \leftarrow \langle c, c'\rangle$

Input $\mathsf{decide}(c')_{d,k,c,i}$
Effect:
  if $\neg failed$ then
    if $status = \mathsf{active}$ then
      $rec\text{-}cmap(k) \leftarrow c'$
      if $op\text{-}status = \mathsf{active}$ then
        if $cons\text{-}data(k) = \langle c, c'\rangle$ then
          $op\text{-}outcome \leftarrow \mathsf{ok}$
        else
          $op\text{-}outcome \leftarrow \mathsf{nok}$

Output $\mathsf{recon\text{-}ack}(b)_{d,i}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op\text{-}status = \mathsf{active}$
  $op\text{-}outcome = b$
Effect:
  $op\text{-}status = \mathsf{idle}$

Input $\mathsf{fail}_i$
Effect:
  $failed \leftarrow \mathsf{true}$
```

Figure 13. $Recon_{d,i}$**: Transitions.**