A New Fully-Distributed Arbitration-Based Membership Protocol

Shegufta Bakht Ahsan
University of Illinois at Urbana Champaign
sbahsan2@illinois.edu

Indranil Gupta
University of Illinois at Urbana Champaign
indy@illinois.edu

***Abstract*—Recently, a new class of "arbitrator-based" membership protocols have been proposed. These claim to provide time bounds on how long membership lists can stay inconsistent—this property is critical in many distributed applications which need to take timely recovery actions. In this paper, we: 1) present the first fully decentralized and stabilizing version of membership protocols in this class; 2) formally prove properties and claims about both our decentralized version and the original protocol; and 3) present experimental results from both a simulation and a real cluster implementation.**

***Index Terms*—failure detection, consistency, membership**

I. Introduction

A group membership protocol, containing a failure detector, is a critical component of large-scale distributed systems and applications. Membership lists are used in datacenters for various purposes including for traffic routing [9], [33], [50], for multicast [6], [22], for replication [31], etc. These are used in distributed databases [38], publish-subscribe systems [4], [18], peer-to-peer systems [42], online gaming [23], [30], etc.

A membership service provides, at each node (process) in the distributed system, a view of a subset of the other nodes (processes) that are alive. We consider only fail-stop failures (when a process crashes it stops further actions; recovering processes rejoin with a new ID) . The membership protocol automatically updates the membership list(s) upon node joins, voluntary departures, and especially upon fail-stop failures. The failure detector component of the membership protocol must be efficient in messages and detection time, not miss any failures (called *Completeness*) [11], make few mistakes in detection (called *Accuracy*), and scale with group size [14].

An additional critical requirement that we focus on is *consistency*. Membership protocols in use today sit at two opposite extremes of the consistency spectrum:

- **Weakly-consistent membership protocols** provide an eventual guarantee on membership lists, with some providing a (large) time bound on convergence. Nodes may see inconsistent views of the membership lists for very long periods of time, even under realistic conditions like zero clock drift. Examples include SWIM [14], ring-based heartbeating [27], gossip-style heartbeating [21], [49], etc. These are used in peer-to-peer systems [42] and key-value/NoSQL databases [25], [35], because these applications are themselves weakly-consistent.
- **Strongly-consistent membership protocols** ensure that membership lists are identical at all nodes. If the membership list delivery is totally ordered at alive nodes, this is called virtual synchrony (or view synchrony) [5], [10], [22]. At the same time, there are no timing guarantees on detection–alive nodes may see inconsistent views of the membership lists for indeterminately-long periods of time, even under realistic conditions like zero clock drift and reliable communication (but with unbounded latencies).

It is essential to design membership protocols which provide consistency that is a *timing guarantee* for how long two nodes' membership lists can stay mutually inconsistent w.r.t. a membership change. This is critical in many real-world applications such as banking, stock markets, air traffic control, vehicle routing, etc. [34], [39], [47]. In all these applications, consistent recovery actions need to be taken in a timely manner at multiple nodes, and concurrent incorrect actions by different alive nodes may cause significant application errors.

Recently, a new class of membership protocols has emerged which claim to provide timing guarantees on how long membership lists stay inconsistent. First proposed as part of the Microsoft's Service Fabric system in Eurosys 2018 [28], these membership protocols have provided sufficient consistency to build applications like Azure SQL DB, Azure Cosmos DB, Microsoft Skype, Azure Event Hub, Microsoft Intune, Azure IoT Suite, Microsoft Cortana etc.

At the same time, the consistency properties of the Service Fabric's failure detector were never formally proven (only hypothesized and sketched). In addition, the Service Fabric failure detector relies on a centralized component called the "arbitrator" to arbitrate conflicting failure detection decisions.

In this paper we make the following contributions:

- We fully decentralize the arbitrator in this new class of failure detectors (Sec. IV).
- We propose an efficient node join mechanism to accompany the decentralized arbitrator (Sec. V).
- We present theoretical analysis for our new algorithm (Sec. VI). Some of this analysis also extends to the original algorithm in the Service Fabric paper [28].
- We briefly present both simulation and cluster deployment results to measure and compare the performance of our new algorithms (Section VII).

II. System Model and Service Fabric's Centralized Membership Protocol

We describe the system model, and background on the Service Fabric's [28] *Centralized* Membership Protocol.

System Model: We assume that clock drifts are zero, i.e., clock rates are identical. Clocks can have skew. Messaging is reliable, timely, and ordered, e.g., via TCP. These are reasonable assumptions in datacenters today. We consider fail-stop failures only. There might be heterogeneity in the system, however all nodes are susceptible to failure.

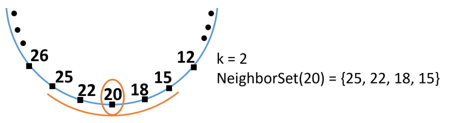

Fig. 1. **Ring topology.** *Nodes are organized in a virtual ring. Each node maintains neighborhood set consists of k successors and k predecessors.*

Service Fabric's Centralized Membership Protocol, using Arbitrators: Service Fabric's new class of membership protocols [28] separate out failure *detection* (i.e., the act of finding a failure), from failure *decision* wherein detecting nodes start recovery actions for the failure. Failure detection is done via a fully distributed lease-based mechanism (akin to heartbeats). Failure decisions, on the other hand, are executed via a centralized group of nodes called arbitrators, which act as judges to arbitrate inconsistent detections. While centralized detection approaches like ZooKeeper [26] place the traffic of both detection and decision on a central group of nodes, the new approach distributes heavy detection traffic and arbitrators only handle the relatively-rare decision traffic and work.

In Service Fabric [28], nodes are organized in a virtual ring (Fig. 1) consisting of 2^m points. A node is mapped to a point on the ring, and so is a key (e.g., via hashing). A key is owned by its closest node, with ties won by the predecessor.

However, the arbitrator is still centralized (at a node or at a group of nodes). This means that if the arbitrator were to fail, or if a majority of an arbitrator group were to fail, then no decisions can be made. In such circumstances, all conflicts will result in mistaken detection+decisions and nodes being forced to leave the system. Other details of Service Fabric–beyond those covered in the main paper [28]–are proprietary; such proprietary details are not discussed in this paper as they are not relevant to our contributions.

III. Mechanisms Borrowed by Our Decentralized Arbitrator

Our fully-decentralized failure detector borrows two kinds of mechanisms from the original Service Fabric detector: *leasing mechanism* and *failure decision*. Later, Section IV will build atop this to achieve the fully decentralized detector.

While the original paper [28] only cursorily sketched the central failure detector algorithm, here for completeness we present these important components in a formal manner. Table I presents all symbols used in this paper.

Symbol	Uses
T_a	Arbitration timeout interval
T_l	Leasing period
T_{arb}	$(2T_l + T_a)$; Once an arbitrator locally logs a node as failed, after this time units it can safely trim the log entry (Corollary 2)
L_{PQ}	Lease from node P to node Q
$L_{PQ(n)}$	n^{th} leasing session (building block of L_{PQ})
$LR_{PQ(n)}$	n^{th} leasing request sent from node P to node Q
$LKRQ_{CP}$	Lock Request from a new candidate node C to the existing node P
$ACK_{PQ(n)}$	Reply of $LR_{PQ(n)}$, sent from node Q to node P
N_P	Neighborhood set of node P
k	Neighbor count in clockwise/counter-clockwise direction in the ring. $\|N_P\| = 2k$
$\mathcal{A}_{PQ}$	Node P's view of the arbitrator group for the pair (P,Q)
$Arb(P \rightarrow Q)$	Arbitration request send from node P, suspecting node Q
$Propose(ver : P*, Q)$	Proposal message, send from node P to upgrade the arbitrator-group $\mathcal{A}_{PQ}$. $P*$ is the proposed arbitrator-group version-number.

TABLE I
Symbols used throughout the paper.

1. Lease-Based Monitoring: We describe the leasing scheme in detail, and an example is depicted in Fig. 2. Consider a lease L_{PQ} between a monitor node P and its monitored node Q. The protocol for maintaining and renewing the lease consists of consecutive, monotonically increasing, non-overlapping leasing sessions $(L_{PQ(*)})$, each lasting for a duration of T_l time units. Initially, at P, the status of both node Q and lease L_{PQ} are $Alive$. At the beginning of the n^{th} leasing session $L_{PQ(n)}$, node P sends a Lease Request $LR_{PQ(n)}$ to node Q and marks the lease session's status as $Pending$. If node P receives the ack $ACK_{PQ(n)}$ in a timely manner, the status of the leasing session is changed to $Established$.

At the end of the ongoing leasing session (T_l time units after $LR_{PQ(n)}$ was sent) node P checks the session's status and initiates the $(n+1)^{th}$ leasing session only if the current status is marked as $Established$. On the other hand, if the status still remains as $Pending$ then this is a timeout and $ACK_{PQ(n)}$ was not received–then node P terminates the lease L_{PQ} and marks $STATUS(L_{PQ(n)}) = Timeout$. It also considers node Q as a $Suspected$ node.

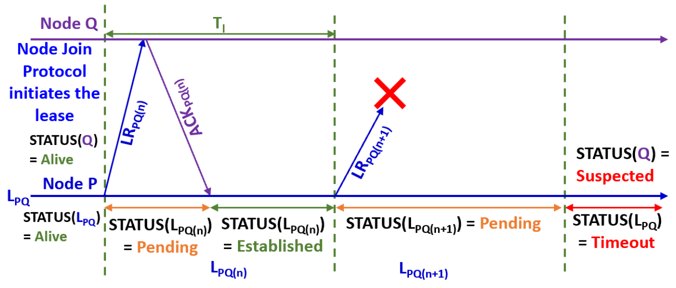

Fig. 2. **Lease based monitoring.** *Monitor node P maintains lease L_{PQ} with its monitored node Q.*

Monitoring relationships are symmetric, stated formally as:

Rule 1 (Symmetric Monitoring (SM))**.** *Neighbor nodes P and Q independently establish leases to each other: L_{PQ} and L_{QP}. Without loss of generality, if $L_{PQ}(n)$ times out, then P ignores all subsequent $LR_{QP(*)}$ lease requests arising from Q.*

Algorithm 1 Monitor-Arbitrator Protocol: Arbitrator Actions

```
 1: Input: Arbitrator receives arb. request $Arb(P \rightarrow Q)$
 2: if arbitrator has been up for less than $T_{arb}$ time units then
 3:     Recently-Failed ← (P ∪ Q ∪ Recently-Failed)
 4:     return reject
 5: else if node P ∈ Recently-Failed list then
 6:     return reject
 7: else if node Q ∈ Recently-Failed list then
 8:     return accept
 9: else        // First Come First Serve approach
10:     Recently-Failed ← (Q ∪ Recently-Failed)
11:     return accept
12: end if
```

2a. Failure Detection: If P's lease request to Q times out, P detects Q as failed and marks node Q as $Suspected$.

However, the lack of further knowledge makes it impossible to draw an accurate conclusion about the suspected node's current status. This is one of the fundamental limitations of failure detection in asynchronous systems. Node Q could have been Suspected due to a plethora of reasons: i) The lease-request from node P was lost; ii) The ack from node Q was lost; iii) Slow or flaky network prevented the lease-request/ack from arriving in time; iv) Node Q actually died; v) Nodes P and Q are partitioned out. As such, it is possible that P and Q (and perhaps other mutual monitors) concurrently and contradictorily *suspect* each other.

In order to resolve such conflicts, when a node P detects Q as Suspected, it does not take failure recovery actions right away (e.g., reorganize the ring). Instead, P moves to a *Failure Decision* mode in order to confirm Q's failure.

2b. Failure Decision: Failure Decisions are done via a centralized group of nodes called as *arbitrator-nodes*. If a failure detection of Q by P is confirmed by the arbitrator-group, then this implies P has permission to recover from Q's failure: P can remove Q from its view of the ring, claim some of Q's keys, etc.

Rule 2 (Arbitration Request)**.** *If two nodes (P,Q) maintain Symmetric Monitoring and (without loss of generality), L_{PQ} times out, then Node P immediately sends an arbitration request $Arb(P \rightarrow Q)$ to the arbitrator group. If it receives no response from the arbitrators within T_a time units (T_a is a fixed parameter system-wide), P voluntarily leaves the system. Otherwise it obeys the arbitrator group's decision. Arbitrators follow Algo. 1.*

3. The Arbitrator-Group: The arbitrator-group acts as a referee and provides failure decisions. Each arbitrator node maintains small state but acts independently–there is zero sharing across arbitrator-nodes. Each arbitrator-node maintains a list, called *Recently-Failed*, which contains node IDs that it has recently declared as *dead*. The list contains only failures confirmed within a fixed recent time duration (Corollary 2).

Algo. 1 describes the actions taken by an arbitrator-node on receiving a *suspect* request from P about Q. If this arbitrator is new (e.g., a new joiner replacing a failed arbitrator), then it rejects all requests and marks both suspecter and suspected nodes as failed, and responds with a *reject* to P (Line 4). This boostrapping rule is needed to ensure zero-sharing across arbitrators, and avoid bad decisions by new arbitrators.

Otherwise, if the arbitrator is well-established, it checks if P was recently declared dead–then its request is rejected and P is again asked to leave the system (Line 6). If P is considered alive but Q was recently marked as dead, then P's request is accepted (Line 8). Finally (Line 9) if both P and Q were alive, then P's request is accepted–this means that if P and Q simultaneously suspected each other, the winner is the one among them whose request arrives *first* at the arbitrator-node.

4. Obeying the Arbitrator-Nodes' Decisions: Once node P detects a failure of Q, it sends arbitration requests individually to each arbitrator-node and awaits responses for T_a time units. A request to an arbitrator-node results is one of three outcomes: *accept*, *reject* or *timeout* (Algo. 1). After receiving all responses or after T_a time units–whichever occurs earlier–P marks Q as failed if and only if a majority of arbitrator-nodes (quorum in arbitrator-group) responded with an *accept* vote. Notice that arbitrator-nodes respond independent of each other and do not need to coordinate among each other. Consequently, after a wait of $(2T_l + T_a)$ time units since P sent arbitration requests (this wait time is calculated in Corollary 3), P can safely assume that the suspected node Q has left the system. Thereafter P can take actions to recover from Q's failure.

Otherwise, if a majority of P's arbitration requests result in a response that is in (*reject OR timeout*), then P voluntarily leaves the system. In this way, P sacrifices itself for the consistency of the system's failure decisions. We call such departures as *forced departures*, and later our experiments will measure them. While undesirable, forced departures are a crucial mechanism needed to maintain membership-consistency across the system.

IV. New Distributed Arbitrator-based Consistent Failure Detector

The downsides of the algorithm described in Sec. III [28], [45] arise from the use of the central arbitrator-group. During periods marked by a large number of node failures, arbitrator-nodes may become congested by a high volume of requests, causing timeouts at requesting nodes. If a majority of arbitrator-nodes are slow or faulty, all *suspecting* nodes will be forced to leave the group, causing massive forced departures of healthy nodes. Additionally, failed arbitrators need to be replaced manually in the original protocol (this is also true in Zookeeper via "rolling-restart")—our protocol allows automated arbitrator replacement, as they are chosen in a self-stabilizing way from alive nodes.

To address these issues, we decentralize the arbitrator-group itself. The key idea is to eschew having a fixed set of arbitrator-nodes. Instead, we allow each pair of monitoring nodes (P, Q) to select its own arbitrator-group. The challenge is to ensure that this is done in a way that retains correctness of the membership. This arbitration selection mechanism is our first contribution. Our second contribution is handling changes in the arbitrator-group itself–because of failures or departures, the arbitrator-group's membership cannot stay static, and needs to be changed over time. These two contributions are then combined with the leasing protocol (Fig. 2) and the Monitor-Arbitrator protocol from Algo. 1 to produce our overall membership protocol.

A. Decentralized Arbitrator Selection

Eliminating the centralized arbitrators would be straightforward had they been fully stateless. We observe first that the only state an arbitrator-node maintains is the Recently-Failed list (Section III–The Arbitrator-Group).

We note that at a minimum, to maintain consistency, this Recently-Failed list needs to be checked only upon mutually-conflicting failures. That is, only under circumstances when P suspects Q, and Q suspects P, and thus both P and Q send arbitration requests. For all other requests (e.g., R suspects P after P has suspected Q), the Recently-Failed list minimizes the number of forced detections, but is not required for consistency.

Using this observation we eliminate the arbitrator-nodes as follows: we replace them with a subgroup of nodes chosen for pairwise arbitration. That is, we use a separate arbitrator-group for every pair of monitoring (neighbor) nodes P and Q. Later, our experiments will show that this does not cause an increase in forced departures.

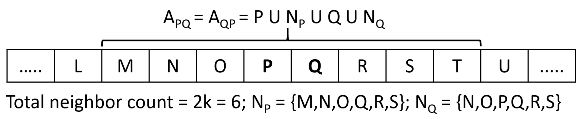
$A_{PQ} = A_{QP} = P \cup N_P \cup Q \cup N_Q$
.....
L
M
N
O
P
Q
R
S
T
U
.....
Total neighbor count = 2k = 6; N_P = {M,N,O,Q,R,S}; N_Q = {N,O,P,Q,R,S}

Fig. 3. **Pairwise Arbitrator Group Formation Strategy.**

Fig. 3 depicts our pairwise arbitrator-group formation strategy via an example. The pairwise arbitrator set $\mathcal{A}_{PQ}$ maintained at P for the pair (P, Q) consists of P, Q, as well as their respective sets of neighbors (in the ring) N_P, N_Q. Formally:

$$\mathcal{A}_{PQ} = \mathcal{A}_{QP} = P \cup N_P \cup Q \cup N_Q \tag{1}$$

Both P and Q maintain this mutual arbitrator list, and if their mutual leases expire, they refer to $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$ respectively for the Failure Decision.

B. Dynamic Arbitrator-Groups

Maintaining Consistency Between $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$: Node P and Q must maintain a consistent view of $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$. Otherwise, in case of failure there is the risk that P and Q consult with two different set of arbitrator-nodes–these partially-overlapping/non-overlapping arbitrator-groups might independently *accept* both P and Q's arbitration requests, causing both P and Q to stay in the system but believing each other is failed.

At any time P's neighborhood N_P might change to $N_{P(ver:P1)}$ (because of node join/leave/failure, etc.). Therefore, to reflect the new neighborhood, $\mathcal{A}_{PQ}$ needs to be upgraded to a new version, denoted as $\mathcal{A}_{PQ(ver:P1)}$. However, node Q might not immediately be aware of node P's neighborhood change. Therefore, an immediate upgrade of $\mathcal{A}_{PQ}$ might cause inconsistency between $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$.

Safe and Consistent Arbitrator-Group Upgrade Protocol: We use a novel approach that seamlessly upgrades the

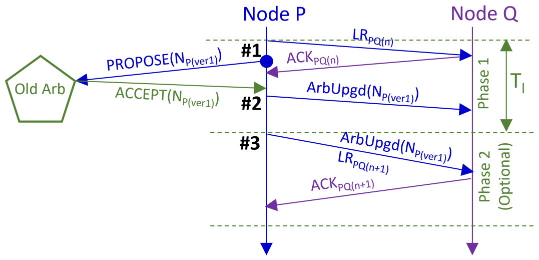

Fig. 4. **Safe and Consistent Arbitrator Group Upgrade Strategy.** *For simplicity all lease requests initiated from node Q ($LR_{QP(*)}$) and their corresponding ACKs are omitted.*

pair-wise arbitrator-groups and prevents inconsistency between them. Whenever P needs to upgrade the current arbitrator-group for Q, it follows a *two-phase* protocol that leverages the current arbitrator-group $\mathcal{A}_{PQ}$ in order to perform a safe and consistent upgrade. This is depicted via an example in Fig. 4, and we describe the phases below.

Phase 1: Before upgrading the arbitrator-group from $\mathcal{A}_{PQ}$ to $\mathcal{A}_{PQ(ver:P1)}$, P sends an *arbitrator-group upgrade* proposal $Propose(ver : P1, Q)$ to its current arbitrator-group $\mathcal{A}_{PQ}$ (Fig. 4:#1). The proposal contains the new *arbitrator-group version-number* proposed by P (in this case P1). These arbitrator-nodes record the proposal, and respond with acks. This results in one of three outcomes. These outcomes and P's subsequent actions are as follows:

1) **The majority times out:** Node P voluntarily leaves.
2) **The majority rejects:** Node P aborts the current arbitration change attempt and retries later.
3) **The majority accepts:** Node P upgrades the arbitrator-group and sends an explicit *arbitrator-group upgrade* confirmation message $ArbUpgrd(N_{P(ver:P1)})$ to Q (Fig. 4:#2). This message contains the updated neighborhood $N_{P(ver:P1)}$ and the current *arbitrator-group version-number* $P1$. Node Q uses $N_{P(ver:P1)}$ to upgrade $\mathcal{A}_{QP}$ and attaches the new *arbitrator-group version-number* along with its future arbitration requests.

Phase 2: This phase piggybacks the *arbitrator-group upgrade* confirmation message with the next scheduled lease-request/ack and thus implicitly notifies Q about the arbitration-group change (Fig. 4:#3). This redundant phase ensures that even if the previous explicit *arbitrator-group upgrade* confirmation message was lost, the next lease-request/ack will convey it. In other words, *if P and Q continue to maintain successful leases into the future, then the arbitrator-group upgrade information will be propagated to Q within $2T_l$ time units after P upgrades its arbitrator-group (Theorem 1).*

Modified Arbitration Policy: Arbitrator-nodes perform normal request processing based on Algo. 1. In addition, arbitrator-group members need to deal with the arbitrator-group upgrade proposals as follows: when an arbitrator-node receives the *arbitrator-group upgrade* proposal from node P (i.e., $Propose(ver : P1, Q)$), it checks if it has seen any arbitration-group upgrade proposal from Q (i.e., $Propose(ver : Q*, P)$), or an arbitration request $Arb(Q \rightarrow P)$ from node Q since the last T_{arb} time units (Corollary 2). If any of these is true, the arbitrator-node *rejects* the request. Otherwise it locally stores the *current version (P1)* for $\mathcal{A}_{PQ}$ and replies an *accept* to P.

Handling the stale arbitrator-group in node Q: However, in a rare scenario Q might detect P as failed just before receiving the *arbitrator-group upgrade* message. In that case Q sends the *arbitration request* to the *old arbitrator-group* along with the *old arbitrator-group version-number*. Because at least majority of the *old arbitrator-group* has already accepted node P's *arbitrator-group change* request, they *detect* and *reject* Q's *stale* arbitration request. Therefore, in the case of such inconsistency, Q gracefully leaves the system.

Recently-Failed List: In the decentralized version, a node C that belongs to the arbitrator set for a pair of nodes (P, Q) still keeps a finite Recently-Failed list. This list consists of at most two entries: whether P was previously marked as dead (and when), and whether Q was previously marked as dead (and when). Unlike the Recently-Failed list in the centralized protocol (Sec. III) which could be arbitrarily long, our Recently-Failed lists are limited in size to 2 entries. Further, since each node has $2k$ arbitrator sets and each such set has at-worst $4k$ members, by symmetry each node participates in at-most $2k \times 4k = 8k^2$ arbitrator sets.

V. Node Join Protocol under Decentralized Arbitrators

Because we have replaced the central arbitrator with a decentralized arbitrator, we need to design a new node join protocol to keep the required state consistent. This is is a four-phase protocol.

Phase 1 (Neighbor Discovery): The candidate joining node (say node C) sends a *neighbor discovery request* to that node Q which currently owns the key C. (The consistent ring routing algorithm described in [28] can be used for this).

If Q is currently serving another *node join* request, it *rejects* node C's request (C can retry). Otherwise Q replies with *accept*– this reply also piggybacks Q's current neighbor set N_Q.

In the case of *time out* or *rejection*, node C backs-off and retries later. Otherwise it moves forward by using the N_Q set to calculate its own potential neighborhood set N_C. This is feasible because N_C is always a subset of $(Q \cup N_Q)$ (Fig. 5).

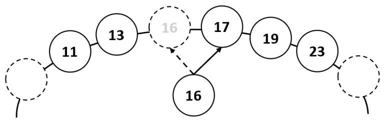

Fig. 5. **Calculating the potential neighborhood.** *The ring-member node 17 currently holds the key 16. Neighbor count per direction,* $k = 2$. *The candidate node (node 16) gets the key holder's current neighborhood set* $N_{17} = \{11, 13, 19, 23\}$. *Node 16's potential neighbourhood set would be* $N_{16} = \{11, 13, 17, 19\}$.

Phase 2 (Lock Request): For correctness, our protocol needs to ensure that prospective neighbors (monitors) of the joining node are not involved in processing another node join. Hence joining node C next sends a *lock request* to all of its $2k$ potential neighbors, N_C (Fig. 6). A node receiving this lock request rejects the request only if it is actively processing another node join. Otherwise the lock is granted for the next $(3 \cdot T_l)$ time units (time to finish Phases 2-4). If C can acquire all the locks in time, it moves to Phase 3. Otherwise, it releases all the established locks, backs off and retries from Phase 1.

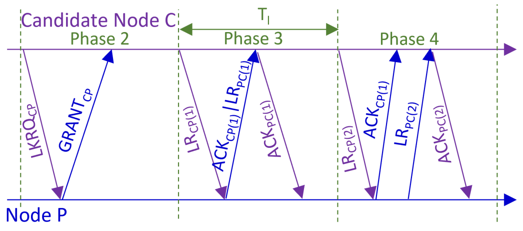

Fig. 6. **Node join protocol (initiated from the candidate node C).** *For simplicity, Phase 1 is omitted and only one neighbor is shown.*

Phase 3 (Lease Invitation): Node C establishes independent leases L_{C*} and sends the *first lease requests* $LR_{C*(1)}$ to all members of N_C. It piggybacks its current potential neighborhood set N_C (this is used by C's neighbor P to create the mutual arbitrator list A_{PC} based on Equation 1).

Once a neighbor (node P) receives the first lease request $LR_{CP(1)}$ it prepares the acknowledgement $ACK_{CP(1)}$ and initiates the symmetrically-opposite lease L_{PC} for node C. It piggybacks both the symmetrically-opposite first lease request $LR_{PC(1)}$ and its current neighbor set N_P with the ack message.

Node P also records the arbitrator set for its relationship with C: $\mathcal{A}_{PC} = P \cup N_P \cup C \cup N_C$, and marks the arbitrator-set as *dormant*. Dormant means that if the lease L_{PC} times out, instead of involving the arbitrator group $\mathcal{A}_{PC}$, node P will clean-up the lease and neither send further lease requests to C nor response to any of C's future lease-requests.

Upon receiving the incoming ack $ACK_{CP(1)}$, node C gets the new lease request from P, $LR_{PC(1)}$ and P's neighborhood set N_P. At this point node C continues the Symmetric Monitoring according to Rule 1 in Sec. III. C sends back $ACK_{PC(1)}$ to P. Besides, C also prepares the arbitrator set $\mathcal{A}_{CP} = C \cup \{N_C\} \cup P \cup \{N_P\}$, and marks this as *dormant*.

If at least a single lease request with any of its neighbors times out, node C discards all the established leases L_{C*}, backs off and retries again starting from Phase 1. If a neighbor (say node P) has already established a lease L_{PC}, that lease will also automatically time out at P (since node C will not reply back eventually). At node P, the arbitrator-group $\mathcal{A}_{PC}$ has still been marked as dormant. Therefore, instead of involving the arbitrator-group, Node P merely deletes the timed-out lease L_{PC}, i.e., it does not attempt to renew this lease in the future. Otherwise, if C and each of its neighbors P successfully establish symmetric leases in this 1^{st} session $L_{C*(1)}$, then C moves to Phase 4.

Phase 4 (Wrap-up): Node C sends the 2^{nd} lease request $LR_{C*(2)}$ to all of its neighbors. Once $LR_{CP(2)}$ is received, the neighbor P can safely assume that C has successfully established the Symmetric Monitoring (Rule 1) with the members of N_C, hence safely marks the arbitrator group $\mathcal{A}_{PC}$ as *active*. Therefore, from now on whenever the lease L_{PC} times out, P will consult with the $\mathcal{A}_{PC}$ for failure decision.

Once node C establishes the 2^{nd} leasing session with all of its neighbors and receives all the corresponding acks $(ACK_{C*(2)})$, it considers itself as an active ring member and marks the corresponding arbitrator-groups ($\mathcal{A}_{C*}$) as *active*. At this point, the failure detection and decision algorithms specified in Section III can be used between P and C.

VI. Theoretical Analysis

We analyze correctness of: A) our decentralized (and thus the original centralized) arbitrator approach, B) our node join protocol, and C) overheads.

A. Decentralized Arbitrators

We first analyze our fully-decentralized failure detector from Sec. IV. Unless noted otherwise, our theorems below also apply to the original Service Fabric failure detector (centralized) from Section III and [28]. This is a side-contribution of our paper, because the original Service Fabric paper [28] (and its based-on white papers/patents [24], [29]) did not come associated with rigorous analysis—we thus prove their previously-held hypotheses. Our first theorem is specific to our new decentralized arbitrator.

Theorem 1 (Maximum Inconsistency Interval Between Arbitrator-groups $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$)**.** *Consider neighbors P and Q. Say P successfully upgrades the arbitrator-group $\mathcal{A}_{PQ}$ at absolute time T. If the lease L_{PQ} stays established for long enough into the future, then: Q will reflect the upgrade by time $T + 2T_l$.*

Proof. To upgrade the arbitrator-group $\mathcal{A}_{PQ}$, node P follows a two-step process (Fig. 4). Let upgrade time T occur in the m^{th} leasing session $L_{PQ(m)}$. P sends an explicit confirmation $ArbUpgrd(N_{P(ver*)})$ to node Q. However, this explicit message might get lost in the network. Yet, because $ArbUpgrd(N_{P(ver*)})$ will be piggybacked with the next subsequent request, if P and Q's mutual leases stay correct for long enough into the future, this next lease request for $L_{PQ(m+1)}$ will succeed and will communicate $ArbUpgrd(N_{P(ver*)})$ to Q. Hence Q will upgrade its view of the arbitrator-group $\mathcal{A}_{QP}$. Therefore, Q knows about the upgrade by time $T + 2T_l$. □

The following results hold for both the new decentralized version and the centralized version of arbitrators.

Next we analyze failure detection times. For a given node Q, define its *failure detection time* (T_{FD}) as the time between Q's failure occurring and *all* of Q's monitors suspecting Q.

Theorem 2 (Failure Detection Time)**.** *When a node Q fails and at least one of its monitors is alive, then: T_{FD} is bounded from both below and above, as: $T_l \leq T_{FD} < 2T_l$.*

Proof. A pair of ring neighbors, node Q and node P maintains the Symmetric Monitoring (Rule 1). Let Q be the failing node and P be the alive monitor. Further, let Q fail during P's n^{th} lease period for Q, and the failure occurs β time units after the start of that lease period.

1) Node P initiates the n^{th} leasing session $L_{PQ(n)}$ and sends the leasing request $LR_{PQ(n)}$.
2) Node Q receives the lease request and replies back with $ACK_{PQ(n)}$. After sending this response, Q crashes.
3) Node P receives the $ACK_{PQ(n)}$ in time and marks the n^{th} leasing session as established. Once the current session completes (which takes T_l time units since the lease request), P initiates the $(n+1)^{th}$ leasing session.
4) However, as node Q has been crashed, node P's $(n+1)^{th}$ session with Q will timeout after a further T_l time units.

Starting from the point when node Q crashes, the time left at the n^{th} leasing session $L_{PQ(n)}$ is $(T_l - \beta)$. Node P detects the failure at the end of the $(n+1)^{th}$ leasing session. Therefore, the failure detection time $T_{FD} = (T_l - \beta) + T_l$. But since $\beta \in (0, T_l]$, hence we have $T_l \leq T_{FD} < 2T_l$. □

Next we analyze how long it takes for two neighbors to mutually suspect each other.

Theorem 3 (Symmetric Lease Timeout)**.** *The Symmetric Monitoring (Rule 1) between a pair of nodes P and Q consists of two independent leases: L_{PQ} and L_{QP} respectively. Without loss of generality, assume L_{PQ} times out first. From that point onwards, if it takes T_{LT} time unit to time out L_{QP}, then: $T_l \leq T_{LT} < 2T_l$.*

Proof. The run consists of the following steps:

1) Node P detects the time out of L_{PQ} at absolute time T and starts rejecting any future lease requests from Q (Rule 1).
2) At Q, let $L_{QP(m)}$ be the ongoing leasing session that starts just before T and receives the corresponding $ACK_{QP(m)}$.
3) Starting from T, the time left to finish the ongoing m^{th} leasing session is T_b $(< T_l)$.
4) As the m^{th} session was a success, Q initiates the next session and sends the lease request $LR_{QP(m+1)}$ to P.
5) However, since node P is already rejecting all of node Q's future leasing requests, this $(m+1)^{th}$ leasing session at node Q (for P) will timeout after a further T_l time units and node Q will finally mark the lease L_{QP} as timed out.

Therefore, starting from the moment when P first detects the lease timeout of L_{PQ}, Q will also detect the timeout of L_{QP} no more than $(T_b + T_l)$ time units. As $T_b \in [0, T_l)$, hence the lease timeout interval T_{LT} is bounded by $T_l \leq T_{LT} < 2T_l$. □

Corollary 1 (Mutual Arbitration Requests)**.** *Given two neighbors P and Q, if P suspects Q and sends an arbitration request, then: within another $T_{arbReqInt}$ time units, Q will also send an arbitration request suspecting P. $T_l \leq T_{arbReqInt} < 2T_l$.*

As soon as the lease L_{PQ} times out, P immediately sends an arbitration request suspecting Q (Rule 2). However, according to Theorem 3, L_{QP} also times out within T_{LT} time units and Q immediately sends an arbitration request suspecting P. The interval $(T_{arbReqInt})$ between the two arbitration requests coincides with T_{LT}. Hence $T_l \leq T_{arbReqInt} < 2T_l$.

We now analyze how long a suspected node can survive.

Theorem 4 (Maximum Lifespan of a Suspected Node)**.** *Let Q be suspected by a monitor P, and thus Q is forced to leave the system. Starting from the moment when node Q is first suspected by node P, the suspected node can stay in the system no later than $T_{maxLifespan} < (2T_l + T_a)$ time units.*

Proof. As soon as the lease L_{PQ} times out, P suspects Q and makes an arbitration request. Due to the First Come First Serve (FCFS) policy (Algo. 1), the arbitrator accepts that request. Q *may* also send an arbitration request no later than $2T_l$ (Corollary 1) time units after, and awaits the response. The arbitrator will reject the request due to FCFS. Two cases arise:

1) If node Q successfully receives the rejection it will immediately leave the system.
2) If the arbitration request times out (after T_a time units), node Q will leave the system (Rule 2).

Starting from the failure detection at node P, the suspected node (node Q) can stay in the system for at most $(T_{arbReqInt} + T_a)$ time units. Therefore, the maximum possible time that node Q can stay in the system is bounded from above as: $T_{maxLifespan} < (2T_l + T_a)$. □

Corollary 2 (Arbitrator can safely remove any entry older than $T_{arb} = (2T_l + T_a)$ from the *Recently-Failed* list)**.** *A pair of ring neighbors, node P and Q is maintaining the Symmetric Monitoring (Rule 1). Without loss of generality, node P suspects Q first and sends an arbitration request.*

Once node P suspects Q, starting from that time, node Q can stay in the system for at most $T_{maxLifespan} = 2T_l + T_a$ time units (Theorem 4). Therefore, it is guaranteed that if the arbitrator receives P's request at absolute time T, at $T + T_{maxLifespan}$ time units node Q must leave the system. Hence, the arbitrator can safely remove entries from the Recently-Failed list that are older than $T_{arb} = 2T_l + T_a$.

Corollary 3 (Safe time to Start Failure Recovery)**.** *Given two neighbors P and Q, if P suspects Q (failure detection) and the arbitrator agrees with it (failure decision), then: node P should wait for $(2T_l + T_a)$ time-span to safely declare node Q as dead, and start any recovery actions.*

As Theorem 4 points, if node Q is suspected by node P at time T, counting from that time, node Q can stay in the system for at most $T_{maxLifespan} = (2T_l + T_a)$ time units. Therefore, node P should wait for at least that time units before finally considering node Q as dead.

Finally, we can state that our protocol (as well as the centralized protocol of Section III and [28]) satisfy the time-based consistency that we outlined in Section I:

Theorem 5 (Time-based Consistency)**.** *The decentralized (and centralized) arbitrator-based failure detection protocol maintains time-based consistency, as defined in Section I. Concretely, after a node Q fails, within another $(T_a + 4T_l)$ time units two conditions are true: i) Q leaves the system, and ii) all of Q's monitors know about Q's failure.*

Proof. When Q crashes, an alive monitor node P will detect it within T_{FD} time units where $T_l \leq T_{FD} < 2T_l$ (Theorem 2). However, node P has to wait for another $(2T_l + T_a)$ time units to safely mark node Q as dead (Theorem 4). Therefore, $(T_a + 4T_l)$ after node Q's failure, Q has left the system and all its monitors can start recovery actions. □

B. Node Join Protocol Correctness

Next we analyze the Node Join protocol of Section V.

Theorem 6 (Correctness of Single Node Join under Failures)**.** *The Node Join Protocol for a given joining node maintains consistent membership lists in spite of failures.*

Proof. If any of the monitors of a joining node C fails at any point before the specific instant of time that C has successfully established the second leasing session with *all* of its monitors, then C will time out in one of the above phases, and gracefully leave the system. Subsequently, monitors will also release their locks after the $3T_l$ timespan. If on the other hand, C fails after it has finished Phase 4, our normal failure detection mechanism (Sec. IV combined with Sec. III) will detect C's failure. □

Theorem 7 (Correctness under Multiple Node Joins)**.** *The Node Join Protocol Maintains consistent membership lists in spite of multiple nodes joining simultaneously.*

Proof. Because a joining node C first acquires locks on its potential monitors, no other joining nodes share the same monitors (neighbors) as C. Thus, our protocol allows simultaneous joining nodes if and only if their neighbor sets (monitor sets) are disjoint. Together with Theorem 6, this maintains consistency. □

Theorem 8 (Time-bounded Node Join)**.** *When there are no failures or dropped messages, then: a new joining node, after Phase 1, finishes joining in another $3 \cdot T_l$ time units.*

Proof. Each of the remaining Phases 2-4 take T_l time units. Hence node joining finishes in $3 \cdot T_l$ time units. □

C. Overhead and Downsides

Leasing Message Overhead: The leasing messages comprise of both lease-requests from any node P to Q $(LR_{PQ(*)})$ and the corresponding acks $(ACK_{PQ(*)})$. The number of leasing messages per second, per node, is calculated as:

$$2 \times (2k) \times \frac{1}{T_l} \tag{2}$$

This equation shows that the leasing message overhead per node scales with system size. Additionally, T_l represents a tradeoff between overhead and detection time: selecting a smaller T_l ensures faster failure detection, but also means frequent lease renewals and more stabilization messages.

Partitioning Behavior: Finally, in the interest of completeness of analysis, we observe that arbitrator-based approaches cannot avoid the well-known partitioning problem inherent to group membership systems. Both the centralized and decentralized arbitration schemes (Sec. III, IV) are susceptible to collapse when the network is partitioned. In the centralized version, in the worst case, if none of the partitions has a quorum number of arbitrators, nodes detect failures of their neighbors in the other partitions, but are unable to obtain a quorum number of arbitrator responses, and therefore leave the system. These forced departures cascade, and eventually everyone leaves the system. The decentralized version also suffers from the same forced departure + cascading failure behavior.

VII. Evaluation

We implemented the decentralized arbitrator-based failure detector in both: 1) a simulator, and 2) a real Java implementation, which we run on the Emulab cluster [16].

A. Impact of Arbitrator Failure (Simulation)

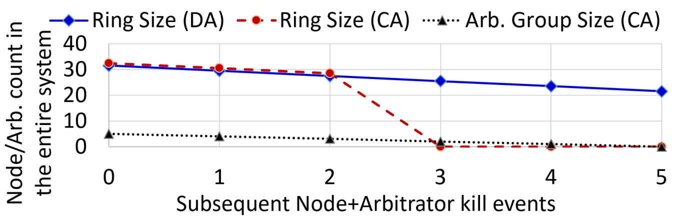

Fig. 7. **Impact of Arbitrator Failure.** *X axis represents the arbitrator + regular node kill event. Y axis shows the total number of arbitrator/regular node in the system.*

Fig. 7 shows, via simulation, the impact of an arbitrator crash for two different systems: *CA* which uses the Centralized arbitrator from Service Fabric [28], and *DA* which is our scheme. The ring consists of 32 nodes, and the CA variant uses a set of 5 arbitrators. The DA uses 6 monitors per node.

The X-axis shows *crash events*–at each event, 2 nodes are killed: in CA, a member-node and an arbitrator-node; in DA, two random nodes. The Y-axis shows the active node count.

We observe that the original CA scheme is tolerant only up to 2 arbitrator failures. With 3 or more arbitrator failures, there are insufficient number of arbitrators left to make failure decisions, and as a result all nodes voluntarily leave the system, and the ring size (system size) drops to 0 quickly. In comparison, our DA scheme maintains a stable system size, which drops only by the number of crashed nodes.

In summary, we conclude that the distributed arbitrator based scheme is more resilient to arbitrator failure.

B. Failure Detection Accuracy (Emulab)

Next we measure whether failing nodes affect detection accuracy of otherwise healthy nodes. The main concern here is detection *cascades*, wherein failing or leaving nodes cause other nodes in the neighborhood to also voluntarily leave, due to the timeouts involved in leasing and/or arbitration. Furthermore, because our algorithm is decentralized, it is plausible to assume that such cascading failures may be exacerbated compared to the centralized version. This experiment implicitly measures the effect of this behavior.

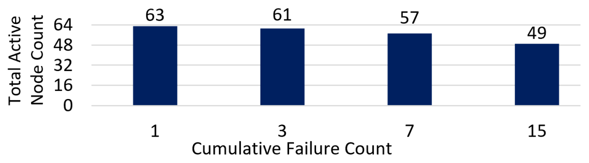

Fig. 8. **Failure Detection Accuracy.** *X axis shows the number of nodes failed simultaneously. Y axis shows the total active node count in the system.*

We deploy a 64-member group in an EmuLab cluster with 5 d710 nodes [13]. Each node consists of one 2.4 GHz 64-bit Quad Core Xeon processor, 8MB L3 cache, 12 GB DDR2 Ram. The 64 nodes are in a ring, with 3 successors and 3 predecessors (thus a total of 6 monitors each).

In Fig. 8 we progressively kill a group of randomly-selected nodes. The x-axis shows these simultaneous failure events. Thus this plot is a timeline plot (without showing the time). The y-axis shows the number of alive nodes left after the detections+decisions have stabilized, after each failure event.

Fig. 8 shows that, at the very first event when only one node failed, the total alive node count becomes 63. This indicates that only the failed node leaves, and there are no additional or cascading departures. As the reader can observe, for each subsequent failure event, the marginal reduction in system size is identical to the number of failing nodes.

In summary, this experiment validates the desirable behavior that failures are detected accurately, and do not force further departures of nodes due to cascades.

C. Arbitrator Message Overhead (Simulation)

One of the goals of the decentralized scheme was to reduce load on the arbitrators. This is important as it keeps arbitrators fast and responding timely, and reduces risk of arbitration requestors timing out.

We run a simulation with a ring consists of total 1000 nodes. First we run the centralized scheme with $\alpha = 9$ arbitrator nodes. Neighbor count in each direction of the ring (clockwise and anticlockwise) is $k = 3$. $f = 100$ randomly selected nodes were failed sequentially. The interval between two consecutive failures was sufficient to bring the stability back into the system.

Each failure causes $2k$ arbitration requests at each of the α arbitrator nodes, creating an overhead of $(2 \cdot k \cdot f \cdot \alpha)$. This implies a total of 600 messages per arbitrator node.

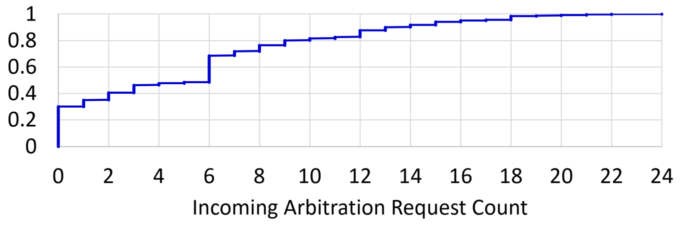

Fig. 9. **Incoming Arbitration Requests: CDF.** *Our decentralized scheme distributes arbitration requests among* 1000 *nodes. The maximum number of arbitration requests received by a node is* 24.

In comparison, Fig. 9 shows that our decentralized scheme imposes a much lower overhead, and creates no bottlenecks. In this scheme, the arbitrator group size varies from $(2 + 2k)$ to $(1 + 3k)$ (depending on how far the monitoring nodes are from each other in the ring). The figure shows that 80% of the nodes receive fewer than 10 arbitration requests. The average is 5.4 arbitration requests per node, and the worst case is 24.

We conclude that compared to the centralized scheme, our decentralized approach reduces worst-case arbitration message overhead by at least two orders of magnitude.

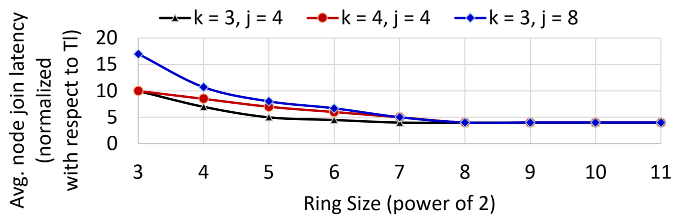

Fig. 10. **Node Join Latency.** k = neighbor count in each direction. j = total number of nodes tried to join simultaneously. The node joining time increases if any of the k or j increases. As the ring grows, the node joining time decreases.

D. Node Join Latency (Simulation)

We use our simulation to measure the speed of our Node Join algorithm from Section V. Fig. 10 shows the node join time (normalized with respect to T_l) for different ring sizes (x-axis is logarithmic in ring size) and different values of k, and j the total number of nodes attempting to join simultaneously. Nodes attempt to join at random positions in the ring. We observe that:

1) Increasing ring size reduces latency because it spreads out the joining load more. The minimum node join time $4T_l$ arises from the 4-phase node join procedure.
2) Latency increases with the number of monitors $(2k)$ (fixing ring size and j) because joining nodes need to coordinate with more ring neighbors (Phase 2 of the node join protocol described in Sec. V).
3) Increasing the number of simultaneous joiners increases latency because of the locking involved in the joining process, which causes contention and some waiting.

In summary, we conclude that node join latencies scale very well and decrease with ring size, and are able to accommodate simultaneously joining nodes.

E. Failure Detection Time (Emulab)

Fig. 11 depicts the failure detection time (normalized with respect to T_l). We use the same EmuLab deployment described above, and vary T_l from 32 ms to 1.024 s.

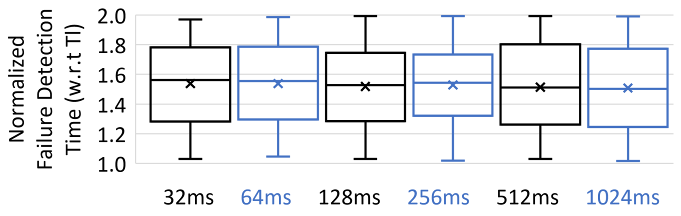

Fig. 11. **Failure Detection Time vs. Leasing Interval T_l.** *Candlestick plots show the* 1^{st}, 2^{nd} *and* 3^{rd} *quartiles and the average (X's on plot). Y-axis normalized w.r.t.* T_l.

The failure detection time T_{FD} is bounded according to our proved result in Theorem 2. That is, $T_l \leq T_{FD} < 2T_l$. On average, it takes around $(1.5 \cdot T_l)$ time to detect a failure.

VIII. Related Work

The key component of a membership protocol is the failure detector. The formal characterization of the properties of failure detectors was first offered by Chandra and Toueg [11] where they also showed that it is impossible for a failure detector algorithm to deterministically achieve both completeness and accuracy over an asynchronous unreliable network.

Chandra and Toueg's impossibility result [11] says that one cannot design a failure detector in an asynchronous network that both detects all failures (completeness) and makes no mistakes (accuracy). Subsequent failure detectors [1], [3], [7], [12], [19], [20], [49] choose to satisfy completeness because of the need for correct failure recovery, and they attempt to optimize accuracy (reduce false positives). Reliable failure detectors include Falcon [32] with sub-second detection times, but the paper states that this protocol does not scale.

Virtual Synchrony and similar approaches [5], [10], [22] offer totally-ordered and consistent membership view. However, members of this protocol family suffer from scalability limitations.

Among weakly consistent protocol are gossip-style heartbeating [48], [49] and SWIM [14]. SWIM uses random pinging for failure detection, and piggybacks failure notifications atop such pings and acks. Such probabilistic membership approaches are used in Cassandra [25], Akka [2], ScyllaDB [43], Serf [44], Redis Cluster [40], Orleans [37], Uber's Ringpop [41], Netflix's Dynomite [36], Amazon Dynamo [15], etc.

A widely used approach to achieve the consistent membership is to store the membership list in an auxiliary service such as Chubby [8], Etcd [17], ZooKeeper [26] etc. Offloading is attractive but increases the dependence on a small set of nodes. Under congestion or failure of a quorum of these special nodes, the membership service is completely unavailable. In comparison, in our system, even under an arbitrary number of failures, membership lists remain available.

Rapid [46] is an interesting protocol that can detect partitions (i.e., cuts). We believe Rapid can be orthogonally combined with our decentralized arbitrator-based failure detector.

IX. Summary

We have presented the design of a new fully-decentralized membership protocol that maintains strong time-based consistency of membership lists. Where past work relied on a central group of arbitrators to referee decisions and conflicts on failure detections, our approach fully decentralizes this arbitrator set. Via formal analysis, we proved important correctness and consistency properties of our scheme, and some of these results prove previously-held hypotheses about the centralized arbitrator scheme. Via both simulation and cluster deployment, we showed that our decentralized membership protocol: 1) minimizes forced departures of healthy nodes, 2) avoids failure cascades, 3) significantly reduces arbitration message overhead vs. centralized scheme, 4) incurs latency that decreases with system size, and 5) detects failures quickly.

X. Acknowledgments

This work was supported in part by a gift from Microsoft, as well as in part by the following grants: NSF CNS 1908888 and NSF RI 1909577.

References

[1] Aguilera, M. K., Chen, W., and Toueg, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Distributed Algorithms, 11th International Workshop, Saarbrücken, Germany* (September 1997), pp. 126–140.

[2] Akka. http://akka.io/.

[3] Almeida, C., and Verissimo, P. Timing failure detection and real-time group communication in quasi-synchronous systems. In *Proc. of the Eighth Euromicro Workshop on Real-Time Systems* (June 1996), pp. 230–235.

[4] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. 19th IEEE International Conference on Distributed Computing Systems* (June 1999), pp. 262–272.

[5] Birman, K., and Joseph, T. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 123–138.

[6] Birman, K. P. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec. 1993), 37–53.

[7] Bollo, Narzul, L., Raynal, and Tronel. Probabilistic analysis of a group failure detection protocol. In *1999 Proc. Fourth International Workshop on Object-Oriented Real-Time Dependable Systems* (Jan 1999), pp. 156–162.

[8] Burrows, M. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 335–350.

[9] Caesar, M., Castro, M., Nightingale, E. B., O'Shea, G., and Rowstron, A. Virtual ring routing: Network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev. 36*, 4 (Aug. 2006), 351–362.

[10] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proc. of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), USENIX Association, pp. 173–186.

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

[12] Chen, W., Toueg, S., and Aguilera, M. K. On the quality of service of failure detectors. *IEEE Transactions on Computers 51*, 1 (Jan 2002), 13–32.

[13] Emulab: D710. https://wiki.emulab.net/wiki/d710.

[14] Das, A., Gupta, I., and Motivala, A. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proc. International Conference on Dependable Systems and Networks* (2002), pp. 303–312.

[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 205–220.

[16] Emulab. https://www.emulab.net/.

[17] Etcd. https://coreos.com/etcd/.

[18] Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec. 30*, 2 (May 2001), 115–126.

[19] Fakhouri, S. A., Goldszmidt, G., and Gupta, I. Gulfstream - a system for dynamic topology management in multi-domain server farms. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science* (Oct 2001), pp. 55–62.

[20] Fetzer, C., and Cristian, F. Fail-awareness in timed asynchronous systems. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (1996), ACM, pp. 314–321.

[21] Ganesh, A. J., Kermarrec, A., and Massoulie, L. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers 52*, 2 (Feb 2003), 139–149.

[22] Gupta, I., Birman, K., and van Renesse R. Fighting fire with fire: Using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering Int. Journal 18*, 3 (2002), 165–184.

[23] Hampel, T., Bopp, T., and Hinn, R. A peer-to-peer architecture for massive multiplayer online games. In *Proc. of 5th ACM SIGCOMM Workshop on Network and System Support for Games* (2006).

[24] Hasha, R., Xun, L., Kakivaya, G., and Malkhi, D. Allocating and reclaiming resources within a rendezvous federation, 2008. US Patent 11,752,198.

[25] Hayashibara, N., Defago, X., Yared, R., and Katayama, T. The ϕ accrual failure detector. In *Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 66–78.

[26] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX Association, pp. 11–11.

[27] What is the IBM SP2? https://www.cac.cornell.edu/slantz/what_is_sp2.html.

[28] Kakivaya, G., and et al. Service fabric: A distributed platform for building microservices in the cloud. In *Proc. of the Thirteenth EuroSys Conference* (2018), ACM, pp. 33:1–33:15.

[29] Kakivaya, G., Hasha, R., Xun, L., and Malkhi, D. Maintaining routing consistency within a rendezvous federation, 2008. US Patent 11,549,332.

[30] Knutsson, B., Lu, H., Xu, W., and Hopkins, B. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM* (March 2004), vol. 1, p. 107.

[31] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *Operating Systems Review 44*, 2 (2010), 35–40.

[32] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M. Detecting failures in distributed systems with the falcon spy network. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 279–294.

[33] Marti, S., Giuli, T. J., Lai, K., and Baker, M. Mitigating routing misbehavior in mobile ad hoc networks. In *Proc. of the 6th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2000), MobiCom '00, ACM, pp. 255–265.

[34] Mesh Systems. http://www.mesh-systems.com/.

[35] MongoDB. https://www.mongodb.org/.

[36] Netflix Dynomite. https://github.com/Netflix/dynomite.

[37] Newell, A., Kliot, G., Menache, I., Gopalan, A., Akiyama, S., and Silberstein, M. Optimizing distributed actor systems for dynamic interactive services. In *Proc. of the Eleventh European Conference on Computer Systems* (2016), pp. 38:1–38:15.

[38] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. In *Proc. of the Sixteenth ACM Symposium on Operating Systems Principles* (1997), pp. 288–301.

[39] Quorum Business Solutions. https://www.qbsol.com/.

[40] Redis. https://redis.io/.

[41] UBER Ringpop. ringpup-184: Leader election on top of a weakly consistent system. https://github.com/uber/ringpop-go/issues/184.

[42] Rodrigues, R., and Druschel, P. Peer-to-peer systems. *Commun. ACM 53*, 10 (Oct. 2010), 72–82.

[43] ScyllaDB. https://www.scylladb.com/.

[44] Serf: Decentralized Cluster Membership, Failure Detection, and Orchestration. https://www.serf.io/.

[45] Service fabric github. https://github.com/Microsoft/service-fabric/.

[46] Suresh, L., Malkhi, D., Gopalan, P., Carreiro, I. P., and Lokhandwala, Z. Stable and consistent membership at scale with rapid. In *2018 USENIX Annual Technical Conference* (July 2018), pp. 387–400.

[47] Talk Talk TV. http://www.talktalk.co.uk/.

[48] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 164–206.

[49] van Renesse, R., Minsky, Y., and Hayden, M. A gossip-style failure detection service. In *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (1998), Middleware '98, Springer-Verlag, pp. 55–70.

[50] Woo, A., Tong, T., and Culler, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. of the 1st International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2003), SenSys '03, ACM, pp. 14–27.