Stateful cluster leader failover models and methods based on Replica State Discovery Protocol

Serhii Toliupa[1,*,†], Maksym Kotov[1,†], Serhii Buchyk[1], Juliy Boiko[2], and Serhii Shtanenko[3]

[1] *Taras Shevchenko National University of Kyiv, 60 Volodymyrska St., Kyiv, 01033, Ukraine*
[2] *Khmelnytskyi National University, 11 Instytuts'ka str., Khmelnytskyi, 29016, Ukraine*
[3] *Military Institute of Telecommunication and Information Technologies named after the Heroes of Kruty, Street of Princes of Ostrozki 45/1, Kyiv, 01011, Ukraine*

Abstract
High availability is a cornerstone of fault tolerance in production clusters. The following article delves into the novel methods and models to achieve rapid cluster leader failover based on the Replica State Discovery Protocol (RSDP). Firstly, RSDP is described and evaluated as a method of achieving consensus within homogenous multiagent distributed systems. This paper provides a novel mathematical model that describes the internal procedure of the said protocol. Additionally, diagrams and algorithm steps are provided to further simplify integration of the RSDP into modern Decentralized Coordination Networks (DCNs). Secondly, a new state reducer is developed that allows to perform a synchronized leader election process. Its mathematical model and code implementation written in JavaScript are provided and comply with an established extension interface described within the confines of RSDP's State Manager component. Evaluation and implications of the newly created leader election protocol are provided to further expand the horizons of DCN coordination. Lastly, this article explores the practical implications of the mentioned state reducer in the context of the stateful cluster leader failover. Three different approaches and models based on the proposed consensus algorithm to mitigate spontaneous critical events are modeled and assessed. Based on failure probability, failover duration, and communication overhead mathematical models, the said approaches were compared, and recommendations for their application were provided. Overall, this article is aimed at further development of RSDP and describes novel approaches towards relevant coordination issues inside the clusters with high demands for availability and fault tolerance.

Keywords
distributed computing; Decentralized Coordination Networks (DCNs); Replica State Discovery Protocol (RSDP); cluster state management models; cluster failover management models; leader election protocol based on RSDP and deterministic operations.

1. Introduction

Being tasked with a complex design of a modern distributed system leads inevitably to the myriads of convoluted architecture decisions towards achieving high availability and fault tolerance. Throughout the entire history of computer science and Internet Technology industry, the cornerstone problem is threefold: consistency, availability, and partition tolerance, renown also as CAP theorem [1-7].

The theorem in its basis promotes an assumption that service could only be two of three: either consistent and available, available and tolerant to partitioning, or consistent and tolerant to partitioning [1-7]. Though it has to be stated that business-centric approaches produced variants of the said theorem that put resource utilization, complexity, and service quality instead while going through the decision-making process.

Information Technology and Implementation (IT&I-2024), November 20-21, 2024, Kyiv, Ukraine
[*] Corresponding author.
[†] These authors contributed equally.
tolupa@i.ua (S. Toliupa); maksym_kotov@ukr.net (M. Kotov); buchyk@knu.ua (S. Buchyk); boiko_julius@ukr.net (J. Boiko); sh_sergei@ukr.net (S. Shtanenko)
0000-0002-1919-9174 (S. Toliupa); 0000-0003-1153-3198 (M. Kotov); 0000-0003-0892-3494 (S. Buchyk); 0000-0003-0603-7827 (J. Boiko); 0000-0001-9776-4653 (S. Shtanenko)

Nevertheless, the crux is the same; it is assumed that no model, method, approach, or methodology exists that could completely satisfy every property of this group. That assumption is still holding strong, since mechanisms that comply with one subset directly obstruct efforts of achieving the other [1-7].

It has been known throughout the history of research efforts into building reliable systems, that trying to achieve fault tolerance while relying on a single instance is futile. This can be attributed to the following reasons:

1. No matter how much the source code is being tested, extremely rare race conditions could still happen when dealing with external devices or even replicated systems.
2. Physical destruction of the datacenter will nullify any logical sound and fault-tolerant mechanism that was preliminarily implemented.
3. Even if we assume that software is logically consistent, a random radiation ray could flip some bits in the system and lead to a catastrophe.
4. There are always highly intelligent individuals that are interested in your system's failure and will not stop until they find a way to affect your system's stability either physically or logically. That is especially problematic during wartime or a nation-wide crisis.
5. At some point you will simply have to put the running instance under maintenance, and this will effectively stop its operation for a while.

While building cloud systems that require high availability, an architect has to eventually consider replication and clusterization techniques [8-12]. Within the context of this article, we will differentiate between these terminologies in the following way:

- Replication — is a distributed topology of a homogeneous multiagent system, where each node or instance referred to as "replica" is tasked with exactly the same purpose as the other participants of the said network. Each replica may have the same set of initial parameters, program code, logic, and ongoing state. Therefore, replicated environments provide rapid disaster recoveries since every other instance could effectively take the responsibilities of the one that failed [8-10].
- Clusterization — refers to the distributed system organization, where the overall state is still synchronized and coordinated but the purpose and the concurrent tasks on different nodes are different. The common purpose of building clustered systems is state splitting. Other examples may include hot and warm standby servers that replicate events from the main machine but still follow the orders from a leader and are usually restricted in their functionality [13-15].

Therefore, the purpose of this article is to model multiple approaches towards coordinating replicated and clustered decentralized networks. As a result of the conducted evaluation, a set of practical recommendations is proposed to simplify the decision-making process during the system design stage.

Additionally, it is the intent of this paper to develop a mathematical model for the Replica State Discovery Protocol, which serves as a framework for performing cluster-wide state synchronization and coordination. RSDP provides a basis upon which a set of logical extensions could be built to achieve various consensus effects.

Using the mentioned consensus basis, multiple leader election mechanisms were developed and modeled. Each of those mechanisms is characterized by a set of unique security, efficiency, and resilience properties, allowing for catering to the need of a specific environment. The said properties were compared based on probability and computational complexity assessments and as a result, recommendations for their application were provided.

2. Replica State Discovery Protocol

The fundamental problem in managing resilience through redundancy is coordination. Since the managed objects are by definition separated, they usually do not have any common memory location that would allow them to successfully establish a synchronization algorithm based on classical concurrency control mechanisms such as locks, mutexes, or semaphores [16-18].

There are quite a few solutions that allow for both immediate and eventually consistent consensus-achieving. An example of the first would be total order broadcast, or, in other words, complete replication of events in the original order. Eventually-consistent algorithms tend to take the process in steps to reach consensus regarding a proposed value. Examples of such algorithms include Raft, Paxos, Ring, ZooKeeper, and many others [19-21].

In that regard, the Replica State Discovery Protocol could be called one of the eventually consistent algorithms. RSDP provides not only the basis for achieving consensus but also serves as a distributed coordination framework, allowing for various extensions and handling cluster events. The difference between classical leader election or consensus-reaching protocols and RSDP is the intention and flexibility they provide. The former usually concentrate on the process of voting for a single common value. In the meantime, RSDP provides a foundation not only for single-state consensus but also for synchronizing complex state setups and merges [22].

2.1. Local Area Network Simulation based on AMQP

RSDP in its core was initially designed on the basis of the local area network simulation based on the Advanced Message Queuing Protocol. The details of its implementation, efficiency, security, implications, and resilience are outlined in a separate article. But for the purpose of theoretical context, a few words have to be said to cover potential questions regarding the reliability of RSDP and its message passing process [22].

First and foremost, the said network simulation is built on top of the message queueing protocol. In that context, AMQP stands as one of the most popular solutions in coordinating and routing complex message network topologies and is continuously gaining momentum in the field of research and engineering [23-25].

Figure 1 shows the conceptual operation basis of AMQP and its components:

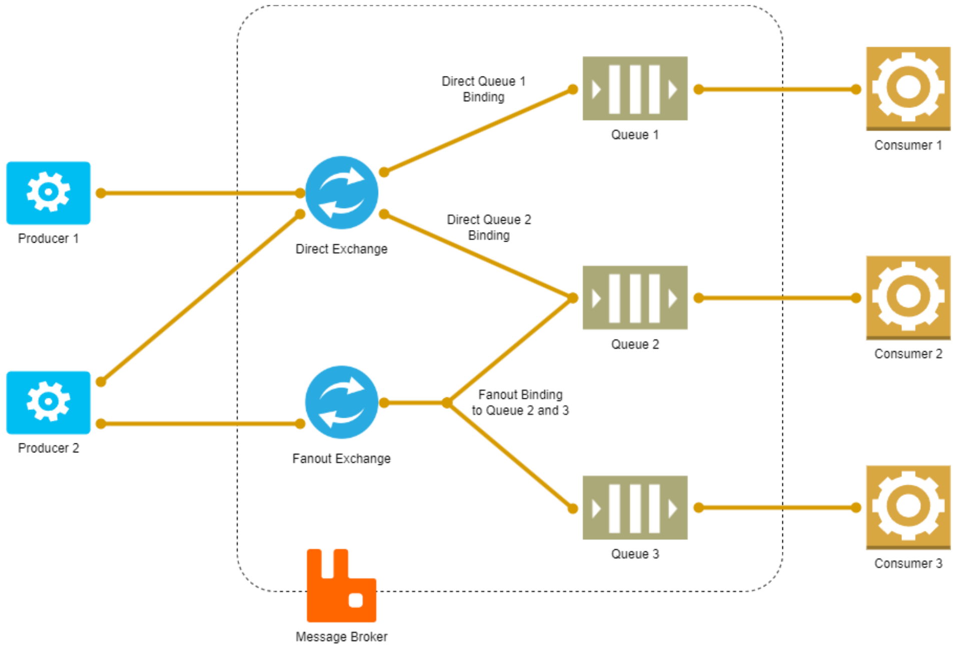


Figure 1: Advanced Message Queuing Protocol's interactions.

The fundamental idea of AMQP is the separation of client and server, which are called producer and consumer. Instead of direct communication, the message goes through the broker and its queue, thus allowing for alleviating direct dependency between clients and servers. Additionally, AMQP describes the achievement of fundamental communication properties such as resilience, durability, congestion control, security, etc. [23-25].

Local Area Network Simulation (SLAN) leverages these capabilities to establish a secure, resilient, and isolated LAN-like environment. In its basis, SLAN describes the provisioning of the two basic communication media: direct communication links and a broadcast link.

Figure 2 shows the interaction media of the SLAN:

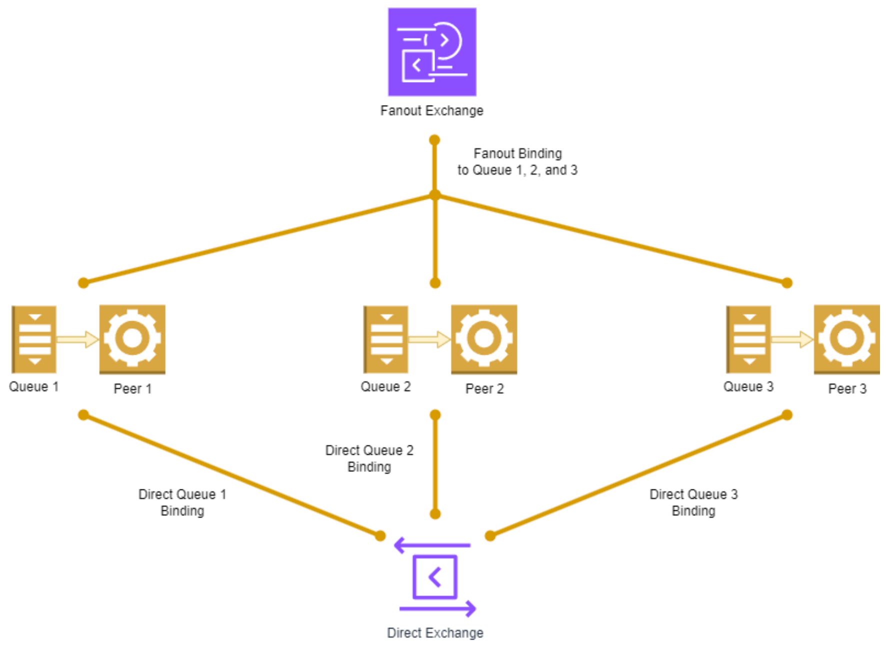


Figure 2: Local Area Network Simulation based on AMQP.

SLAN operation basis relies on the two main capabilities described within the context of AMQP: “Fanout” and “Direct” exchanges. These abstract interfaces allow to either send a message to all binded queues (e.g., broadcast the message) or send the message to a single binded queue by the routing key (direct communication routing).

AMQP defines the durability, mirroring, and quorum mechanisms for its queues and is considered to be a well-documented, tested, resilient, and flexible foundation for communication media. SLAN, and by implication, RSDP, rely on these properties to build its own abstraction layers [26, 27].

2.2. RSDP phases and consensus process

RSDP has its own dedicated article that describes every operation, state change, and consensus-oriented process in detail [22]. This section provides a succinct overview of RSDP phases with some amendments and clarifications for the operation sequences.

To provide capabilities of cluster-wide state operations, RSDP describes its lifecycle in a few distinct phases. These phases include “DEBATES”, “SHARE”, and “CLOSE”. Each of the mentioned phases is respectively responsible for the introduction of the new node, state sharing, and final state derivation. The last phase is responsible for handling the shutdown lifecycle event.

Since each distinct phase somehow interferes with the cluster state stored on each replica individually, every instance has a concurrency control mechanism based on the “*interPhaseMutex*”. That mutex prevents multiple simultaneous cluster events from interfering with each other by restricting stored state access to a single active phase [16-18].

Figure 3 shows the phases and interaction during RSDP process:

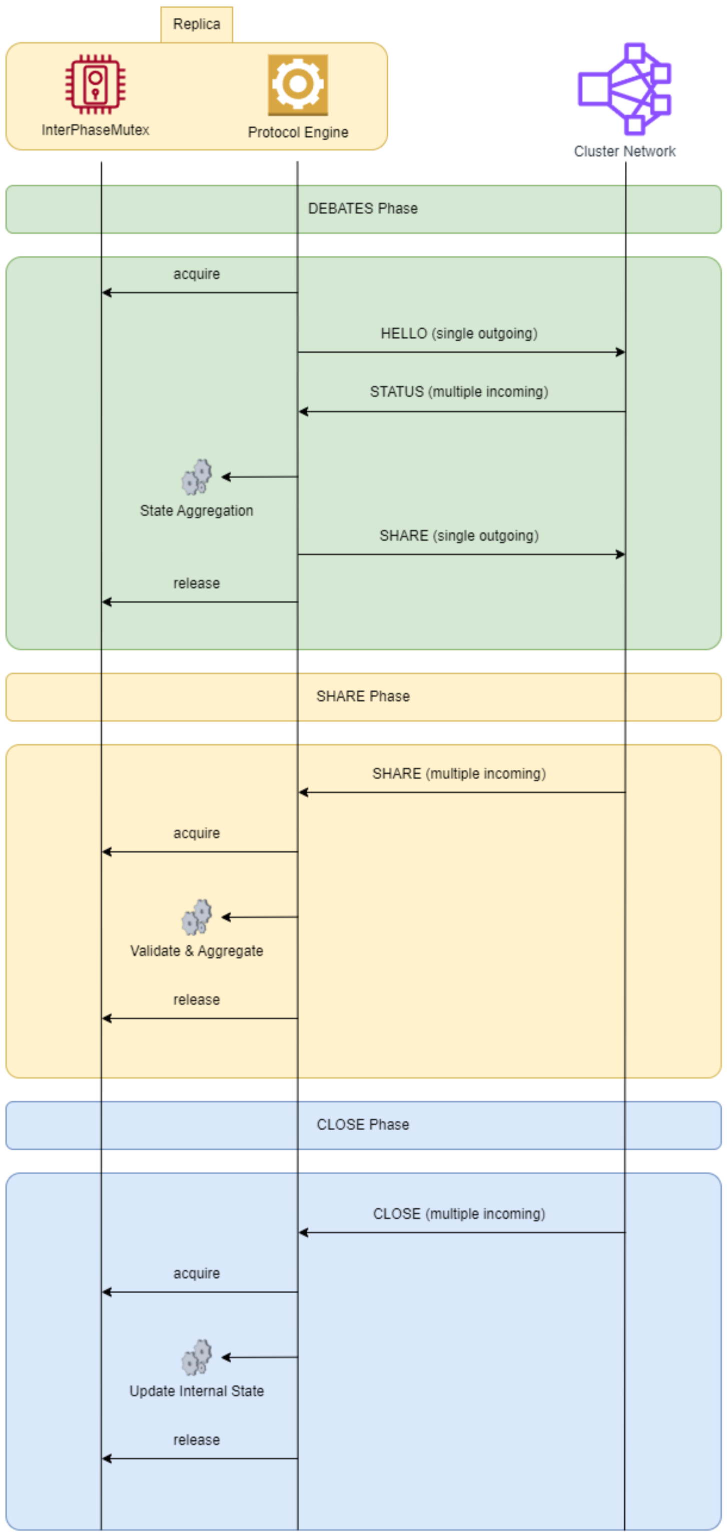


Figure 3: Replica State Discover Protocol phases.

“DEBATES” starts with an acquisition of the “$interPhaseMutex$” that prevents state mutation operations from interfering with each other. RSDP replica then sends a “HELLO” message, announcing its presence in the cluster. This message is sent through the broadcast channel and is meant to be received by all cluster members.

Upon receiving the “HELLO” message, each cluster member would then share its state with a new replica through a direct communication channel by sending the “STATUS” message. A newly created replica would then buffer answers from the cluster members and perform an aggregation of the state as dictated by the state reducers.

Upon receiving an aggregated state, a new replica would then broadcast it through the “Fanout” exchange to all the members. The final operation of the initial phase is to release the mutex and wait for any other cluster-wide events.

“SHARE” phase starts when a replica receives a “SHARE” message from anyone within the cluster. Share messages would then be buffered for a configured amount of time to avoid redundant replica reloads. After the timeout elapses, the protocol engine would acquire the mutex, and the share messages would be validated and aggregated, giving a holistic view of the cluster-wide state. Subsequently, the protocol engine would release the mutex and wait for any other occurring events in the system.

“CLOSE” phase is designed to allow for rapid failover of the replicas. When a replica goes down, it signals the others while others in the cluster would then first acquire the mutex, perform the necessary state updates, and release the mutex. No additional synchronization is necessary as the protocol assumes that every operation performed on the state is deterministic and made within the scope of a set of clean functions that do not rely on side effects.

2.3. RSDP mathematical model

Let $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$ be the set of replicas in the distributed system. The replicas communicate over a network represented as a graph $G = (\mathcal{R}, E)$, where $E \subseteq \mathcal{R} \times \mathcal{R}$ denotes the set of communication channels between replicas.

Each replica R_i maintains a local state S_i, which is an element of the global state space $\mathcal{S}$. The global state of the system is a tuple $\mathcal{S} = (S_1, S_2, \dots, S_n)$.

As was discussed in the previous sections, RSDP operates in three distinct phases: “DEBATES”, “SHARE”, and “CLOSE”. Each of the said phases performs its own set of operations and state transitions.

- “HELLO Message”: Sent by replicas to announce their presence, denoted as $m_{\text{HELLO}}^{(i)}$ from replica R_i.
- “STATUS Message”: Sent in response to a HELLO message, containing the sender's state, and is denoted as $m_{\text{STATUS}}^{(j\to i)}(S_j)$ from R_j to R_i.

During the initiation, each replica R_i broadcasts a “HELLO” message $m_{\text{HELLO}}^{(i)}$ to all other replicas. Upon receiving $m_{\text{HELLO}}^{(i)}$, a replica R_j responds with a “STATUS” message $m_{\text{STATUS}}^{(j\to i)}(S_j)$ containing its current state S_j.

“HELLO” message broadcast could be denoted then as follows:

$$\forall R_i \in \mathcal{R}, R_i \overset{broadcast}{\rightarrow} m_{\text{HELLO}}^{(i)}$$

Subsequently, “STATUS Broadcast” could be denoted as follows:

$$\forall R_j \in \mathcal{R} \setminus \{R_i\}, \text{upon receiving } m_{\text{HELLO}}^{(i)}: R_j \rightarrow m_{\text{STATUS}}^{(j\to i)}(S_j)$$

After that each replica collects “STATUS” messages and aggregates the states to form a combined state S_{agg}. The aggregation function f_{agg} combines individual states:

$$S_{\text{agg}} = f_{agg}\left(\{S_j \mid m_{\text{STATUS}}^{(j\to i)}(S_j) \text{ received}\}\right)$$

The next step for each replica is to broadcasts a “SHARE” message containing S_{agg}:

$$R_i \overset{broadcast}{\rightarrow} m_{\text{SHARE}}^{(i)}(S_{\text{agg}})$$

This operation signifies the end of the “DEBATES” phase within R_i.

The following step is performed withing the scope of the "SHARE" phase. Upon receiving "SHARE" messages, replicas update their local state using a state update function:

$$S_i \leftarrow f_{update}\left(S_i, \{S_{\text{agg}}^{(k)} \mid m_{\text{SHARE}}^{(k)}\left(S_{\text{agg}}^{(k)}\right) \text{ received}\}\right)$$

Finally, during the "CLOSE" phase, a replica R_i intending to exit broadcasts a "CLOSE" message:

$$R_i \overset{broadcast}{\rightarrow} m_{\text{CLOSE}}^{(i)}$$

Remaining replicas adjust their states to reflect the departure:

$$\forall R_j \in \mathcal{R} \setminus \{R_i\}, \quad S_j \leftarrow f_{close}(S_j, R_i)$$

Where f_{close} is a function that removes references to R_i from S_j.

2.4. RSDP formal definition and properties

Define $\mathcal{S}$ as a set of possible states for a replica. Each state S_i may include:

- Membership list $M_i \subseteq \mathcal{R}$;
- Resource utilization $U_i \in \mathbb{R}_+$;
- Other reducer and application-specific data.

Define $\mathcal{M}$ as the set of all possible messages:

$$m \in \mathcal{M} = \{m_{\text{HELLO}}, m_{\text{STATUS}}, m_{\text{SHARE}}, m_{\text{CLOSE}}\} \times \text{Payload}$$

The aggregation function (f_{agg}) combines multiple states:

$$f_{agg}(\{S_1, S_2, \ldots, S_j\}) = \bigcup_k \text{Reducer}_\text{k}(\{S_1, S_2, \ldots, S_j\})$$

This could be defined as:

- For membership lists: $M_{\text{agg}} = \bigcup_j M_j$;
- For resource utilization $U_{\text{agg}} = \frac{1}{|\{S_1, S_2, \ldots, S_j\}|} \sum_j U_j$.

The state update function f_{update} updates the local state based on received aggregated states:

$$S_i \leftarrow f_{update}\left(S_i, \{S_{\text{agg}}^1, S_{\text{agg}}^2, \ldots, S_{\text{agg}}^{(k)}\}\right)$$

This may involve:

- Updating membership lists: $M_i \leftarrow \bigcup_k M_{\text{agg}}^{(k)}$;
- Adjusting resource utilization estimates;
- Updating any other application-specific data.

To prevent race conditions, the "*interPhaseMutex*" is used during critical sections of the protocol, particularly during state updates.

Let μ_i be a mutex for replica R_i. Then, state updates are performed under the lock μ_i:

$$S_i \overset{\mu_i}{\leftarrow} f_{update}(\{S_1, S_2, \ldots, S_j\})$$

As was previously stated, RSDP is based on eventual consistency, where all replicas converge to the same state after a finite number of message exchanges.

For any two replicas R_i and R_j, their states S_i and S_j satisfy:

$$\lim_{t \to \infty} \Pr\left(S_i(t) = S_j(t)\right) = 1$$

The protocol ensures that messages are eventually delivered, and state updates occur. If a message m is sent from R_i to R_j, then m will be delivered to R_j after some finite delay δ. In case some messages will be lost, RSDP defines repeatable synchronization sessions as a contingency.

3. Leader Election Reducer for the RSDP

RSDP defines an interface for logical extensions called "State Reducers". Every operation involving storing, retrieving, validation, aggregating, and updating a state is defined in a set of preconfigured reducers. The original article of RSDP describes the benefits of such a modular approach. Additionally, it provides a thorough explanation of the interface that every reducer must follow to successfully integrate with the protocol. The list of defined methods includes [22]:

- $getCurrentState$: returns the current state of the state slice;
- $aggregateState$: receives a buffer of incoming "STATUS" messages and aggregates them;
- $normalizeState$: transforms the internal state into desired by a client format;
- $aggregateShareState$: receives a buffer of incoming "SHARE" messages and aggregates them;
- $sanitizeShareState$: verifies the validity of an aggregated state;
- $shouldReload$: receives a newly aggregated state and returns a Boolean indicating whether a client should be notified about state change;
- $updateState$: updates the internal state of a reducer;
- $aggregateCloseState$: responsible for handling "CLOSE" messages from replicas;

Out of the mentioned methods, $getCurrentState$ is the only method that is not required. This method is used to form the initial "STATUS" message and may not be applicable to abstract derived states. For example, discovery of replica members does not require $getCurrentState$ implementation since this could be derived from the sender addresses.

3.1. Mathematical model of the leader election process

Leader election reducer is an extension that performs a replicated deterministic holistic decision on the trusted entity based on incoming cluster events. Having discussed the RSDP foundation and the basis for leader election reducer, let us define an abstract description of the consensus process.

Assume c is the current replica instance number, to begin with, let's define:

- $A = \{a_1, a_2, \dots, a_n\}$: the set of all replica addresses;
- $a_{\text{self}} \in A$: the address of the current replica instance, $a_{\text{self}} = a_c$;
- $G_c \subseteq A$: the current set of replica members known to the replica;
- $L \in G_c$: the address of the current leader.

Initially:

- $G_c = \emptyset$;
- $L = \bot$ (temporarily undefined).

Method $aggregateState(statusMessageBuffer)$, as an input, accepts a list of status messages $M_{\text{status}} = \left[m^1_{\text{STATUS}}, m^2_{\text{STATUS}}, \dots, m^l_{\text{STATUS}}\right]$, where each m^i_{STATUS} contains a sender address $m^i_{\text{STATUS}}.\text{address} \in A$. During its execution it performs the following operations:

- Extract addresses: $G' = \{m^i_{\text{STATUS}}.\text{address} \mid i = 1,2, \dots, l\}$;
- Sort addresses: arrange G' in ascending order according to a total order ($\leq$) on addresses A, $G_{\text{sorted}} = \text{Sort}(G')$; This operation could also include additional sorting criteria.
- Determine leader: $L' = \max(G_{\text{sorted}})$;
- Initialize state (if G_c is $\emptyset$ or L is $\bot$): $G_c \leftarrow G_{\text{sorted}},\ L \leftarrow L'$.

As a result of its execution, the new state components are returned as the following tuple: $\text{replicaMembers} = G_{\text{sorted}}, \text{currentLeader} = L'$.

Method $normalizeState(currentLeader)$ expects an $L' \in A$ as an input and performs the following transformation:

$$\text{isLeader} = \begin{cases} \text{true} & \text{if } L' = a_{self} \\ \text{false} & \text{if } L' \neq a_{self} \end{cases}$$

Method aggregateShareState(shareMessageBuffer) expects a list of share messages $M_{share} = [m^1_{\text{SHARE}}, m^2_{\text{SHARE}}, \dots, m^l_{\text{SHARE}}]$, where each m^i_{STATUS} contains data with fields "replicaMembers" and "currentLeader". Below are multiple approaches to aggregate the state components "replicaMembers" and "currentLeader":

The first is the "Latest Source of Truth". During its execution it performs the following operations:

- Select last message: $m_{\text{last}} = m^l_{\text{SHARE}}$;
- Extract state components:
  a. $G' = m_{\text{last}}.\text{replicaMembers}$,
  b. $L' = m_{\text{last}}.\text{currentLeader}$.

As result returns $\text{replicaMembers} = G', \text{currentLeader} = L'$.

The second is the "Popular Vote Decision". This approach counts every incoming proposed state rather than relying on the latest and could be described as follows:

- Let $\mathcal{H} = \emptyset$ be a multiset of hashed state components.
- **For each** message $m_i \in M_{share}$:
  - Extract state components:
    - $G_i = m_i.\text{replicaMembers}$;
    - $L_i = m_i.\text{currentLeader}$.
  - Form state tuple:
    - $S_i = (G_i, L_i)$.
  - Compute hash of the state tuple:
    - $h_i = h(S_i)$, where h is a hash function.
  - Add h_i to $\mathcal{H}$:
    - $\mathcal{H} \leftarrow \mathcal{H} \cup \{h_i\}$.
- Identify the hash h^* with the highest frequency in $\mathcal{H}$:
  - $h^* = \arg\max(\text{frequency}(h_i, \mathcal{H}))$.
- Find $S^* = (G', L')$ such that $h(S^*) = h^*$.

As result returns $\text{replicaMembers} = G', \text{currentLeader} = L'$.

The third is the "Ranked Vote with Exponential Weights". This approach derives the leader by assigning points from each of the participants. Consider a set of share messages defined as the $M_{share} = [m^1_{\text{SHARE}}, m^2_{\text{SHARE}}, \dots, m^l_{\text{SHARE}}]$, where $\forall m_i \in M_{share}$ contains a ranked list of replica members $G_i = [a_{i,1}, a_{i,2}, \dots, a_{i,n_i}]$, where $n_i = |G_i|$ be the number of candidates in m_i, with $a_{i,1}$ being the most desired leader and a_{i,n_i} being the least desired leader.

- **For each** message $m_i \in M_{share}$.
  - **For each** candidate $a_{i,k}$ at position k in G_i compute the weight $w_{i,k} = 2^{n_i - k}$.
- Initialize a score set $C_i = \{c_{i,1}, c_{i,2}, \dots, c_{i,n_i}\}$, where $c_{i,k} = 0$ for $k = 1,2, \dots, n_i$.
- **For each** candidate $a_{i,k}$:
  - Update the candidate's score $c_{i,k} \leftarrow c_{i,k} + w_{i,k}$;

- Let $c_j \in C$ be the total score, where C is a set of total scores for each candidate, then for $a_{i,j} \in G_i$, $c_j = \sum_{i=1}^{l} \sum_{k=1}^{n_i} \delta_{a_{i,j},a_{i,k}} \cdot w_{i,k}$;
- where $\delta_{a_{i,j},a_{i,k}}$ is the Kronecker delta function:
  - $\delta_{a_{i,j},a_{i,k}} = \begin{cases} 1 & \text{if } a_{i,j} = a_{i,k} \\ 0 & \text{if } a_{i,j} \neq a_{i,k} \end{cases}$

- Determine the leader:
  - Identify the candidate L' with the highest total score $\arg\max_{c_j \in C}(c_j)$
  - In case of a tie, apply a deterministic tie-breaker, such as selecting the candidate with the highest address according to the total order $\leq$ on A.

As result returns $\text{replicaMembers} = G', \quad \text{currentLeader} = L'$.

Method $sanitizeShareState(replicaMembers, currentLeader)$ expects a set $G' \subseteq A$ and a leader $L' \in A$.

- Validate Members: $\text{areValidMembers} = (G' \neq \emptyset) \wedge (a_{\text{self}} \in G')$;
- Validate Leader: $\text{isLeaderValid} = L' \in G'$.

Output could be described then as:

- $(\text{areValidMembers} \wedge \text{isLeaderValid}) \implies \{\text{replicaMembers}: G', \text{currentLeader}: L'\}$;
- $\neg(\text{areValidMembers} \wedge \text{isLeaderValid}) \implies \emptyset$.

Method $shouldReload(replicaMembers, currentLeader)$ expects $G' \subseteq A$ and $L' \in A$. During its execution it performs the following operations:

- Compare Members: $\text{membersChanged} = (G' \neq G_c)$;
- Compare Leader: $\text{leaderChanged} = (L' \neq L)$;
- Determine Reload Necessity: $\text{shouldReload} = \text{membersChanged} \vee \text{leaderChanged}$.

Returns a Boolean that indicates whether a client should be notified about the state change.

Method $\text{updateState}(\text{replicaMembers}, \text{currentLeader})$ expects a $G' \subseteq A$ and $L' \in A$. During its execution it performs: if $(G' \neq \emptyset \wedge L' \in G')$ then $(G_c \leftarrow G', \ L \leftarrow L')$.

Method $aggregateCloseState(closeMessageBuffer)$ depends on the implementation of a leader election function and whether it is completely deterministic. If the sorting is done with an inclusion of a locally asserted context, this method is supposed to trigger the initial phase of the RSDP to achieve consistency.

Otherwise it expects a list of close messages $M_{close} = [m^1_{\text{CLOSE}}, m^2_{\text{CLOSE}}, \dots, m^p_{\text{CLOSE}}]$, where each m^i_{CLOSE} has sender address $m^i_{\text{CLOSE}}.\text{address} \in A$.

During its execution it performs the following operations:

- Extract Closing Addresses: $\text{A}_{\text{cl}} = \{m^i_{\text{CLOSE}}.\text{address} \mid i = 1,2,\dots,p\}$;
- Update Replica Members: $G_c \leftarrow G_c \setminus \text{A}_{\text{cl}}$;
- Recalculate Leader: $L \leftarrow \max(G_c)$ if $G_c \neq \emptyset$ else $\perp$.

Different approaches towards implementation of aggregateShareState(shareMessageBuffer) outlined in this section contribute to different properties of the election process. Method based on the latest source of truth is characterized by low computational intensity, but while reasonable in trusted and stable environments, is susceptible to network congestion or failures. This approach is not suitable for networks that have strict requirements for Byzantine fault tolerance.

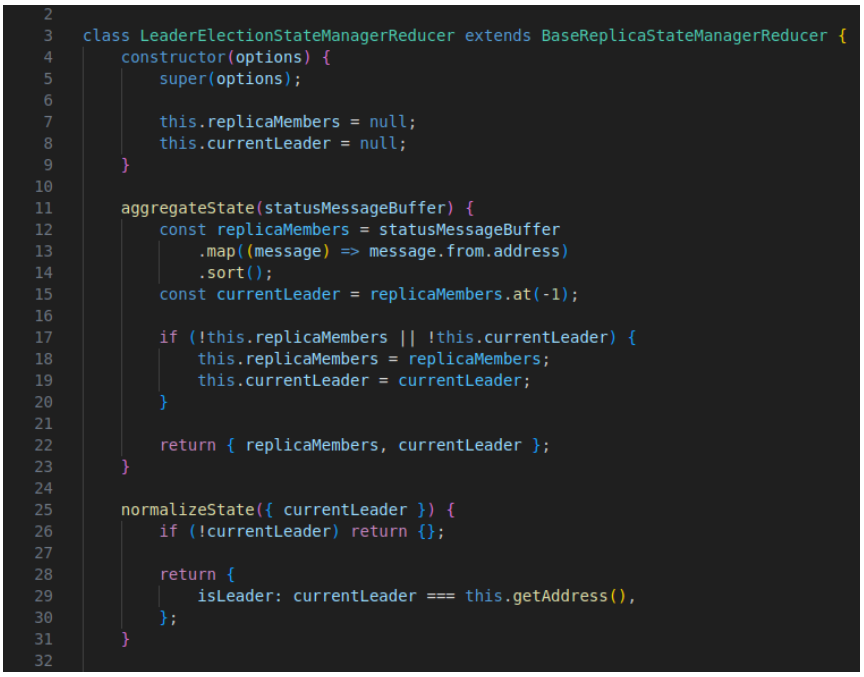
Method based on a popular vote provides higher resilience for both intentional and unintentional discrepancies during the consensus process. It is a suitable approach for systems that are beset with an unstable or restricted environment. It is additionally characterized by increased computational complexity, though it could be reduced by taking into account only scalar values to avoid hashing overhead. This approach is the most resilient among the others against intentionally hostile behavior since, to perform an action, you have to gather the majority of votes.

Finally, the last method provides a solution based on the electoral points approach. It is the most stable and resilient option among the previous three in the context of unstable connections due to its ability to downgrade votes that have lost some portion of a state and hence have a limited view of the global state set. Though it is not resilient towards intentionally hostile actions since the state set cardinality could be superficially inflated.

3.2. Implementation of the leader election reducer

The following section describes practical implementation of the leader election reducer using JavaScript and the *BaseReplicaStateManagerReducer* interface defined by the RSDP. The following algorithm assumes that every cluster member can trust the environment and implements the first approach from the previous section. Such an assumption is a common case when building coordinated replication between internal services to achieve high availability. From this point on we will refer to the leader election reducer by an abbreviation "LER".

Figure 4 shows the initial aggregation implementation logic:



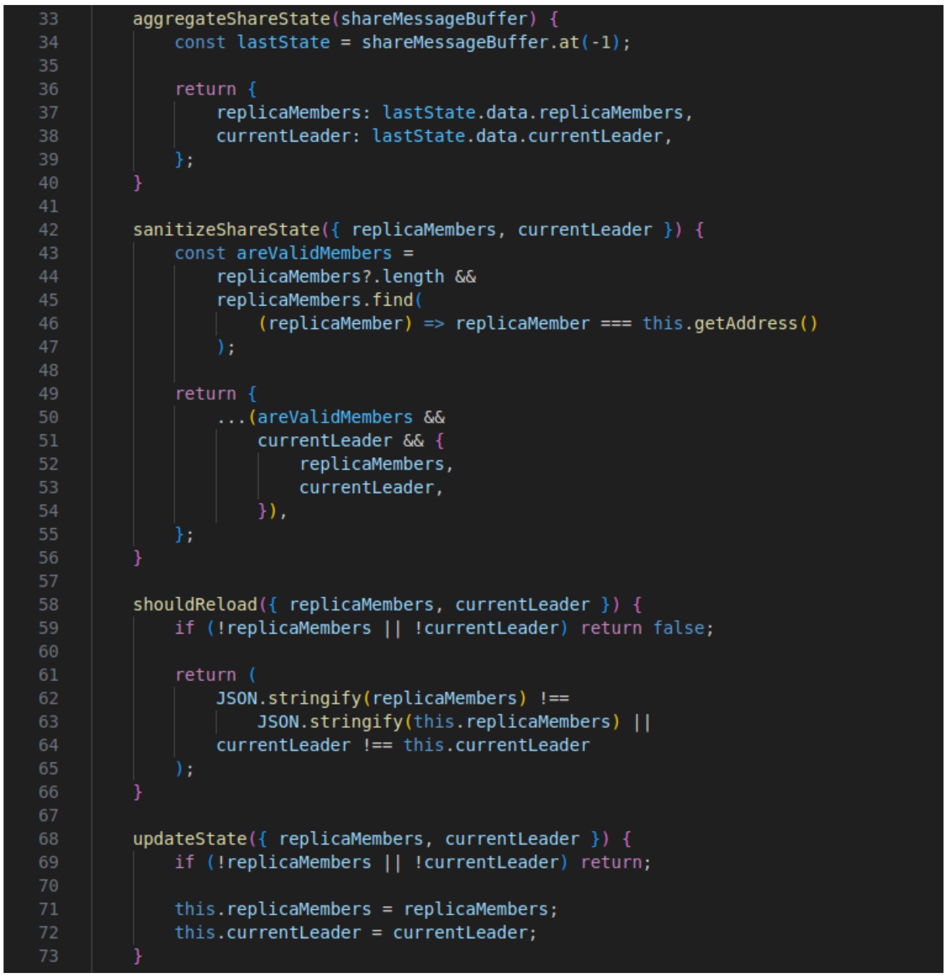
```javascript

class LeaderElectionStateManagerReducer extends BaseReplicaStateManagerReducer {
    constructor(options) {
        super(options);

        this.replicaMembers = null;
        this.currentLeader = null;
    }

    aggregateState(statusMessageBuffer) {
        const replicaMembers = statusMessageBuffer
            .map((message) => message.from.address)
            .sort();
        const currentLeader = replicaMembers.at(-1);

        if (!this.replicaMembers || !this.currentLeader) {
            this.replicaMembers = replicaMembers;
            this.currentLeader = currentLeader;
        }

        return { replicaMembers, currentLeader };
    }

    normalizeState({ currentLeader }) {
        if (!currentLeader) return {};

        return {
            isLeader: currentLeader === this.getAddress(),
        };
    }

```

Figure 4: Initial aggregation logic of the leader election reducer.

The initial state of the LER, as defined by the model, is comprised of an empty set of replica members and an undefined leader. This serves as an example of an abstract derived reducer subset since it does not have an initial *getCurrentState* to share with the cluster.

Method *aggregateState* hence relies not on the data provided by the cluster members but on the messages themselves and their metadata. Its operation is simple; the new leader is defined as the replica that has the highest address. The sorting operation here is not redundant, since to achieve determinism, every node must have the same dataset and ordering. Subsequently, *normalizeState* abstracts out the internal store and provides a simple answer to the client, whether he is a leader.

Figure 5 shows the share aggregation implementation logic:

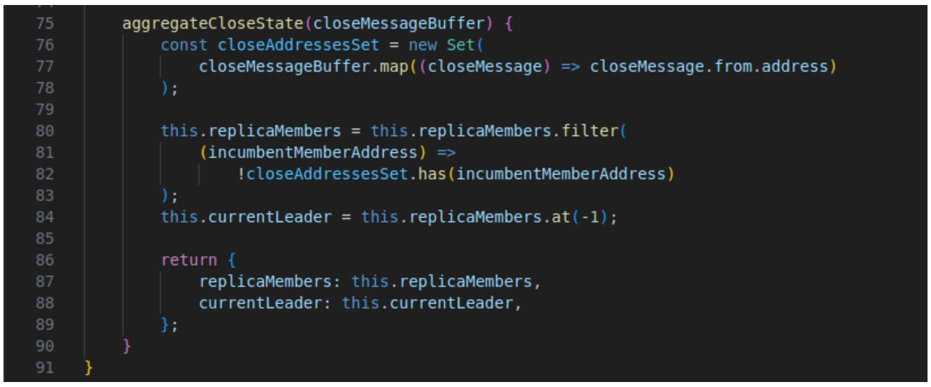


```
    aggregateShareState(shareMessageBuffer) {
        const lastState = shareMessageBuffer.at(-1);

        return {
            replicaMembers: lastState.data.replicaMembers,
            currentLeader: lastState.data.currentLeader,
        };
    }

    sanitizeShareState({ replicaMembers, currentLeader }) {
        const areValidMembers =
            replicaMembers?.length &&
            replicaMembers.find(
                (replicaMember) => replicaMember === this.getAddress()
            );

        return {
            ...(areValidMembers &&
                currentLeader && {
                    replicaMembers,
                    currentLeader,
                }),
        };
    }

    shouldReload({ replicaMembers, currentLeader }) {
        if (!replicaMembers || !currentLeader) return false;

        return (
            JSON.stringify(replicaMembers) !==
                JSON.stringify(this.replicaMembers) ||
            currentLeader !== this.currentLeader
        );
    }

    updateState({ replicaMembers, currentLeader }) {
        if (!replicaMembers || !currentLeader) return;

        this.replicaMembers = replicaMembers;
        this.currentLeader = currentLeader;
    }
```

Figure 5: State share aggregation logic of the leader election reducer.

The *sanitizeShareState* method is responsible for verifying the consistency of the data received from the aggregated state. A leader is valid if it is in a set of known members, and the members are valid if it is not an empty set. Then, the *shouldReload* method decides whether the state has changed and whether the client should be notified. At last, *updateState* simply sets a new state if it was provided.

Figure 6 shows the close aggregation implementation logic:



```
    aggregateCloseState(closeMessageBuffer) {
        const closeAddressesSet = new Set(
            closeMessageBuffer.map((closeMessage) => closeMessage.from.address)
        );

        this.replicaMembers = this.replicaMembers.filter(
            (incumbentMemberAddress) =>
                !closeAddressesSet.has(incumbentMemberAddress)
        );
        this.currentLeader = this.replicaMembers.at(-1);

        return {
            replicaMembers: this.replicaMembers,
            currentLeader: this.currentLeader,
        };
    }
}
```

Figure 6: Replica close aggregation logic of the leader election reducer.

The $aggregateCloseState$ method is invoked during the "CLOSE" phase of the cluster subset. Its primary goal is to deterministically determine a new set of cluster members and a leader. The leader election logic here is the same as during the initial aggregation. RSDP continuously resynchronizes the state of the cluster, so any discrepancies caused by the lost messages will eventually be resolved due to the principle of eventual consistency.

To conclude, RSDP provides a well-defined model for an arbitrary logical extension. The developed algorithm assumes that the environment does not require "Byzantine Fault Tolerance". Farther research could lead to the different invariants of this protocol that could potentially be applicable in decentralized environments.

3.3. Leader election reducer duration and failure probability

Failure probability and consensus duration are two of the most important metrics for the consensus-achieving algorithm. The following section provides mathematical models that describe these properties. We will start with a model of time required to reach a consensus. The following assumptions are made:

- Let n be the total number of replicas in the system.
- Let d be the maximum one-way network delay between any two replicas.
- Let t_p be the maximum time a replica takes to process a message.
- Phases to achieve initial consensus include "DEBATES" and "SHARE".
- All message delays and processing times are bounded and known.
- The SLAN layer provides guarantees of message delivery.
- The probability of the communication media coordinator failure is negligent.

During the "DEBATES" phase each replica sends a "HELLO" message to all other replicas where time for a "HELLO" message to reach other replicas: d. After receiving a "HELLO" message, each replica processes it in time $(\mathrm{n}-1)\mathrm{t_p}$ and sends back a "STATUS" message where time for a "STATUS" message to reach the original sender is d.

For a replica to receive "STATUS" messages from all others, the time is $\mathrm{d}+(\mathrm{n}-1)\mathrm{t_p}+\mathrm{d}=2\mathrm{d}+(\mathrm{n}-1)\mathrm{t_p}$ and since there are $n-1$ replicas sending "STATUS" messages, processing them takes $(n-1)t_p$.

Subsequently, the total "DEBATES" phase time (T_{DEBATES}) is:

$$\mathrm{T_{DEBATES}}=2\mathrm{d}+(\mathrm{n}-1)\mathrm{t_p}+(\mathrm{n}-1)\mathrm{t_p}=2\mathrm{d}+2\mathrm{t_p}(\mathrm{n}-1) \tag{1}$$

During the "SHARE" phase, after aggregating the received "STATUS" messages, each replica broadcasts a "SHARE" message to all others. Time for "SHARE" message to reach other replicas is d. After that each replica processes incoming "SHARE" messages from $n-1$ replicas in time $(n-1)t_p$.

Then the total "SHARE" phase time (T_{SHARE}) is:

$$\mathrm{T_{SHARE}}=\mathrm{d}+(\mathrm{n}-1)\mathrm{t_p} \tag{2}$$

Hence, the total time to reach consensus ($T_{\mathrm{consensus}}$) is:

$$\begin{aligned}\mathrm{T_{consensus}}&=\mathrm{T_{DEBATES}}+\mathrm{T_{SHARE}}=\left(2\mathrm{d}+2\mathrm{t_p}(\mathrm{n}-1)\right)+\left(\mathrm{d}+(\mathrm{n}-1)\mathrm{t_p}\right)\\&=3\mathrm{d}+3\mathrm{t_p}(\mathrm{n}-1)=3\left(d+\mathrm{t_p}(\mathrm{n}-1)\right)\end{aligned} \tag{3}$$

It is obvious then that the consensus achieving time is linearly dependent on the amount of cluster members. Additionally, network delay d and processing time t_p are critical factors, but since the protocol is built on top of deterministic principles and clean functions, t_p should be negligent.

As for the failure probability, the following assumptions are made:

- Let p_l be the probability that a message is lost.
- Let p_f be the probability that a replica fails during the consensus process.
- Message losses and replica failures are independent.
- Each replica must receive "HELLO", "STATUS", and "SHARE" messages from all the replicas.
- A replica successfully participates if it can send and receive all required messages.

The probability that a single message is successfully transmitted is:

$$\mathrm{P}_{\mathrm{msg}} = 1 - \mathrm{p}_{\mathrm{l}} \tag{4}$$

During the consensus achieving stages, each replica must send $n - 1$ "HELLO" and $n - 1$ "SHARE" messages. Consequently, every replica expects to receive $n - 1$ "HELLO", $n - 1$ "STATUS", $n - 1$ "SHARE" messages. Then the total messages received per replica could be represented as:

$$\mathrm{M}_{\mathrm{total}} = 3(\mathrm{n} - 1) = 3\mathrm{n} - 3 \tag{5}$$

Having the total amount of required messages to successfully achieve consensus, the probability that a replica successfully sends and receives all messages:

$$\mathrm{P}_{\mathrm{replica}} = (1 - \mathrm{p}_{\mathrm{f}}) \times \left(\mathrm{P}_{\mathrm{msg}}\right)^{\mathrm{M}_{\mathrm{total}}} \tag{6}$$

Consequently, the probability that all replicas will successfully participate is:

$$\mathrm{P}_{\mathrm{all\ replicas}} = \left(\mathrm{P}_{\mathrm{replica}}\right)^{\mathrm{n}} = [(1 - \mathrm{p}_{\mathrm{f}}) \times (1 - \mathrm{p}_{\mathrm{l}})^{3\mathrm{n}-3}]^{\mathrm{n}} \tag{7}$$

Then the probability of consensus failure for the first election method:

$$\mathrm{P}_{\mathrm{last\ state\ failure}} = 1 - [(1 - \mathrm{p}_{\mathrm{f}}) \times (1 - \mathrm{p}_{\mathrm{l}})^{3\mathrm{n}-3}]^{\mathrm{n}} \tag{8}$$

The probability of failure is dependent on key characteristics of the network and the underlying infrastructure. It is obvious that such an approach is suitable only in cases of stable network connections. SLAN layer provides delivery recovery mechanisms but does not solve every issue related to the message loss. Since the "Popular Vote Decision" and "Ranked Vote with Exponential Weights" do not require successful participation of every node, the probability could be reevaluated and considered in the following way:

Let q be the minimum number of replicas required for consensus (the quorum). For a simple majority:

$$\mathrm{q} = \left\lceil \frac{n}{2} \right\rceil \tag{9}$$

Consequently, the probability that at least q replicas successfully participate is:

$$\mathrm{P}_{\mathrm{quorum\ consensus}} = \sum_{\mathrm{k}=\mathrm{q}}^{\mathrm{n}} \binom{\mathrm{n}}{\mathrm{k}} \left(\mathrm{P}_{\mathrm{replica}}\right)^{\mathrm{k}} \left(1 - \mathrm{P}_{\mathrm{replica}}\right)^{\mathrm{n}-\mathrm{k}} \tag{10}$$

Then the probability of consensus failure:

$$\mathrm{P}_{\mathrm{vote\ failure}} = 1 - \mathrm{P}_{\mathrm{quorum\ consensus}} \tag{11}$$

Evidently, vote-based approaches are significantly more resilient than the method based on the last state decision. In such systems, it is possible to withstand partial failure of participating nodes during the consensus process.

4. Stateful Cluster Failover Models

A stateful cluster in the context of this article is a distributed system where each node has its own subset of the system state. The subset might be either a unique unknown portion for every other cluster member or, as a more common case, a subset of another node's state. Stateful clusters often establish a leader-follower model to achieve high consistency [13-15].

While the entire cluster follows a single leader, it becomes a single point of failure. The principles of fault tolerance in that context require establishing a failover mechanism as a contingency. During its passive phase called "monitoring", the active cluster nodes should be probed and tested to detect any issues promptly. As soon as the critical event on the leader node is detected, the mechanism switches to the active phase of achieving consensus. The entire network has to agree upon a new leader of a cluster to continue its operation [28-30].

The following list includes common definitions used to model every subsequent failover method and their properties:

- N: Number of instances in the cluster.
- M: Number of external observers (Method 2 & 3).
- h: Health-check interval between instances (Method 1).
- h_o: Health-check interval by observers (Methods 2 and 3).
- T_d: Failure detection time.
- T_p: Time to perform the failover procedure.
- T_c: Consensus achieving time (an independent parameter).
- p_i: Probability of an instance failing during the observation window.
- p_o: Probability of an observer failing during the observation window.
- $P_{\text{consensus}}$: Probability of failure in achieving consensus.
- T_{obs}: Total observation time.
- C_m: Average size of a message (in bytes)

Each node/observer sends $(N-1)\ or\ (M-1)$ messages three times during consensus. In the following subsection, each failover topology will be described in terms of failover delay, total failure probability, and communication overhead. The "CLOSE" phase is not considered here since it exists as an optimization phase to avoid going through the entire consensus cycle every time a node leaves the cluster.

4.1. Self-regulated mutual health evaluation

We will first evaluate a model based on a single logical plane. Each node in such a cluster is responsible for operational execution, monitoring, and governance. In such topology, every node must have a communication link with every other in the system to successfully achieve consensus and monitor other instances [31, 32].

The cluster could be preconfigured to initiate health probes in a specified interval but with different initial timestamp shifts based on a node position in the network. This allows to efficiently utilize the repetitive status probes and decrease failure detection time. Additionally, since every node conducts the monitoring, the network can tolerate to up to $N-1$ failed nodes, where N is a node set cardinality.

In that regard, LER provides all the necessary data needed to establish successful monitoring and election in an automated way. The capabilities of LER already provided data for the dynamic node discovery. Hence, every node has a list of the cluster members that they must probe. LER automatically adjusts the states of nodes in a cluster as soon as some subset leaves, but the new leader election cycle could also be triggered in case the nodes suffered critical events.

Figure 7 shows the topology of a self-regulated cluster of nodes:

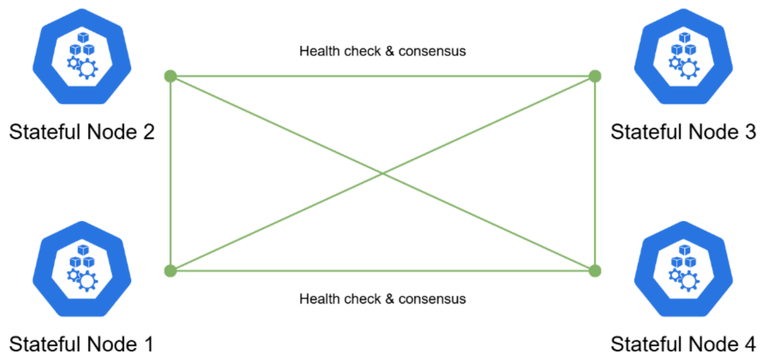


Figure 7: Self-regulated mutual health evaluation.

Each instance performs health checks on every other instance at intervals of h. The expected time for the first instance to detect a failure is the minimum time any instance takes to detect it. Assuming health checks and uniformly distributed, the expected detection time is:

$$\mathrm{E}[\mathrm{T_{d1}}] = \frac{\mathrm{h}}{2\mathrm{N}} \tag{12}$$

After detecting failure, instances need to reach consensus. The consensus achieving time (T_c) is considered an independent parameter to simplify the model and concentrate directly on the factors directly tied to the topology. The total time from failure occurrence T_{f1} to failover completion is the sum of detection time, consensus time, and failover procedure time:

$$\mathrm{T_{f1}} = \mathrm{E}[\mathrm{T_{d1}}] + \mathrm{T_c} + \mathrm{T_p} = \frac{\mathrm{h}}{2\mathrm{N}} + \mathrm{T_c} + \mathrm{T_p} \tag{13}$$

The probability that all instances fail simultaneously ($P_{\text{all instances}}$) could be represented simply as an exponent of a single instance failure:

$$\mathrm{P_{all\ instances}} = p_i^N \tag{14}$$

Then the total failure probability P_{f1} would be described in terms of consensus ($\mathrm{P_{consensus}}$) and all instances ($\mathrm{P_{all\ instances}}$) failure probabilities:

$$\mathrm{P_{f1}} = \mathrm{P_{consensus}} + \mathrm{P_{all\ instances}} - (\mathrm{P_{consensus}} \times \mathrm{P_{all\ instances}}) \tag{15}$$

Then each instance sends health checks to $N - 1$ other instances. During consensus, each instance sends $N - 1$ messages three times. The overall amount of the messages sent during consensus is:

$$\mathrm{M_{consensus}} = \mathrm{N} \times (\mathrm{N} - 1) \times 3 \tag{16}$$

The total number of messages exchanged during the observation period (T_{obs}) is:

$$\mathrm{M_{total1}} = \left(\frac{\mathrm{T_{obs}}}{\mathrm{h}} \times \mathrm{M_{health}}\right) + \mathrm{M_{consensus}} \tag{17}$$

Total communication overhead in bytes:

$$\mathrm{Overhead_1} = \mathrm{M_{total1}} \times \mathrm{C_m} \tag{18}$$

By substituting the logic and models that pertain to the consensus mechanism, we've simplified the models that are tied directly to the system failover properties. But to achieve a holistic view, consensus time and maintenance time models must be included.

4.2. Centralized observer health monitoring

The topology with a centralized observer includes a set of stateful worker nodes in a cluster and a single coordinating machine that manages the entire network. This topology introduces the division between operational and control planes and thus fosters the single responsibility principle in the system [33, 34].

Figure 8 shows the topology of a cluster monitored and coordinated by a single observer:

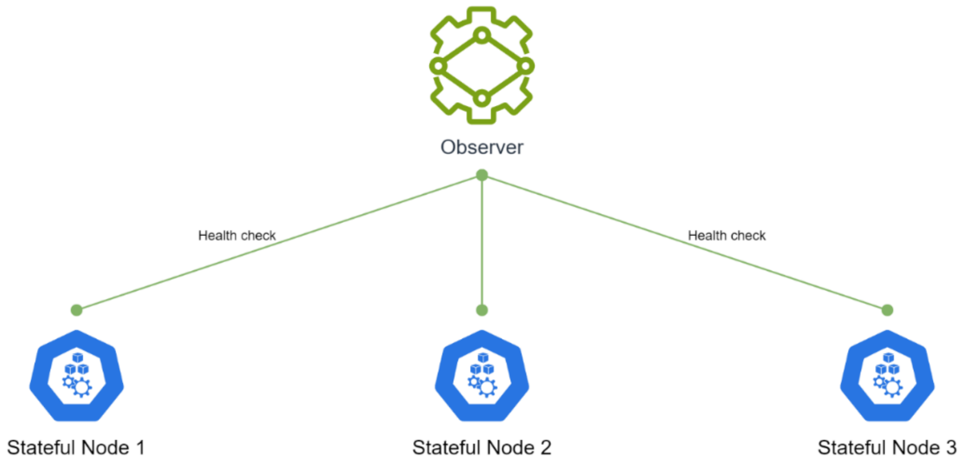


Figure 8: Centralized observer health monitoring.

Let us consider failure detection time T_{d2}. Assume that the observer performs health checks on all instances at intervals of h_o. Then the expected time to detect a failure is half the health-check interval:

$$E[T_{d2}] = \frac{h_o}{2} \tag{19}$$

Since there is no consensus process among instances, the total failover time is:

$$T_{f2} = E[T_{d2}] + T_p = \frac{h_o}{2} + T_p \tag{20}$$

The system relies on a single observer; its failure directly impacts the system's ability to perform failover. Probability of all Instances failing $\left(P_{\text{all\_instances}}\right)$ is the same as in the first method.

The total failure probability includes the observer failure and all instances failing:

$$P_{f2} = p_o + P_{\text{all instances}} - (p_o \times P_{\text{all instances}}) \tag{21}$$

Moving on to the communication overhead model, the observer sends health checks to all N instances. Messages per health-check interval h_o:

$$M_{\text{health}} = N \tag{22}$$

Then the total messages over observation time T_{obs}:

$$M_{\text{total2}} = \frac{T_{\text{obs}}}{h_o} \times M_{\text{health}} \tag{23}$$

Consequently, the total communication overhead in bytes:

$$\text{Overhead}_2 = M_{\text{total2}} \times C_m \tag{24}$$

This model imposes smaller communication overhead and reduces coordination complexity but suffers from a single point of failure.

4.3. Distributed observer health assessment

The following discussed topology is also comprised of two distinct execution plains. In such networks, a consensus algorithm is used between observers themselves and decides on the responsible node that must coordinate the workers plane [35-37].

Figure 9 shows the topology of a cluster monitored and coordinated by a group of observers:

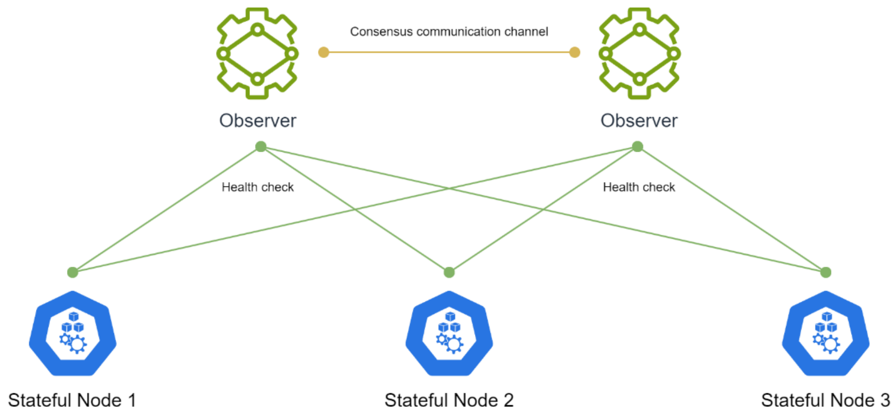


Figure 9: Distributed observer health assessment.

Firstly, let's consider failure detection time T_{d3}. With M observers, the expected time to detect a failure is:

$$\mathrm{E}[\mathrm{T}_{d3}] = \frac{\mathrm{h}_o}{2\mathrm{M}} \tag{25}$$

Observers need to reach consensus after detecting a failure that takes T_c. Then the total time from failure occurrence to failover completion:

$$\mathrm{T}_{f3} = \mathrm{E}[\mathrm{T}_{d3}] + \mathrm{T}_c + \mathrm{T}_p = \frac{\mathrm{h}_o}{2\mathrm{M}} + \mathrm{T}_c + \mathrm{T}_p \tag{26}$$

The probability that all observers fail simultaneously is $P_{\text{all\_observers}} = p_o^M$. Given that the of all instances failing ($P_{\text{all\_observers}}$) is the same as in the previous methods, the total failure probability (P_{f3}) is:

$$\mathrm{P}_{f3} = \mathrm{P}_{\text{consensus}} + \mathrm{P}_{\text{all observers}} + \mathrm{P}_{\text{all instances}} - (\mathrm{P}_{\text{consensus}} \times \mathrm{P}_{\text{all observers}} \times \mathrm{P}_{\text{all instances}}) \tag{27}$$

Each of M observers sends health checks to all N instances. Then the $\mathrm{M}_{\text{health}}$ is:

$$\mathrm{M}_{\text{health}} = \mathrm{M} \times \mathrm{N} \tag{28}$$

Additionally, each observer sends $M - 1$ messages three times during consensus:

$$\mathrm{M}_{\text{consensus}} = \mathrm{M} \times (\mathrm{M} - 1) \times 3 \tag{29}$$

Then the total messages during T_{obs} is:

$$\mathrm{M}_{\text{total3}} = \left(\frac{\mathrm{T}_{\text{obs}}}{\mathrm{h}_o} \times \mathrm{M}_{\text{health}}\right) + \mathrm{M}_{\text{consensus}} \tag{30}$$

Consequently, the communication overhead (Overhead_3):

$$\text{Overhead}_3 = \mathrm{M}_{\text{total3}} \times \mathrm{C}_m \tag{31}$$

This model provides a balance between communication overhead, complexity, separation of concerns and fault tolerance.

5. Conclusions

Achieving consensus within the confines of a Decentralized Coordination Network based on homogenous multiagent system is a critical process that is gaining momentum in the research field due to the ever-growing sizes of distributed systems and requirements towards high availability. Within the context of this article, a novel leader election protocol was created and modeled based on the Replica State Discovery Protocol.

One of the primary goals of this article is to introduce a leader election state reducer as a logical extension of RSDP to address the rapid failover problem. As a result, multiple viable approaches were proposed towards building such a reducer that are based on different properties of common state aggregation. The first proposed method of leader election is based upon the supposition that the network is controlled, and fault tolerance against malicious action during consensus is not expected. That is a reasonable expectation since RSDP is built on top of LAN simulation, which in turn is based on AMQP provider. Such providers often come with a set of authentication and authorization mechanisms of their own. This method is characterized by its low computational overhead and finality characteristics in case of an extremely dynamic network.

The following two methods are based on quorum approach towards handling the leader election process. The method based on a popular vote is the most suitable approach to ensure resilience against both intentional and accidental failures during consensus interactions. Popular vote is a common solution in networks that require Byzantine fault tolerance.

The third proposed leader election method based on RSDP, in its foundation relies on the weighted election algorithm. The approach is characterized by a greater degree of resilience in highly congested and unreliable networks where random packet losses occur frequently due to its ability of partial inclusion.

Given the mathematical models and graphs for the three different cluster failover models and approaches, it is fair to assume that none of those could be called objectively superior in every plain of comparison. Method involving self-regulated mutual health evaluation is most suitable when high additional infrastructure incurrence and the critical event detection time are the most influential metrics of the successful system operation. Though it is worth noting that the communication overhead grows exponentially with the number of cluster members.

The second method, based on centralized observer health monitoring, is an appropriate solution only in cases where higher infrastructure and communication overhead costs are a primary decision factor. Since the entire stability of the system depends on a single centralized external observer, the very same observer becomes a single point of failure, which could lead to a disaster when high availability is a hard requirement.

Lastly, a method based on distributed observer health assessment serves as a trade-off between high availability, infrastructure cost incurrence, communication overhead, and failover delay by offloading the decision-making process to the parallel distributed layer of coordination. Such an approach is mostly suitable for current cloud infrastructure demands due to its flexibility and clearly established separation of concerns.

Overall, this article aims to inspire a surge of further research in the complex, exciting, and extremely relevant field of distributed computing and management. The implications and results of this research allow to bolster the security of modern critical infrastructure by effectively describing novel ways of achieving resilience through redundancy and the distribution of responsibility. Provided mathematical and graphical models are provided to help in the complex decision-making process and reduce possible risks when choosing an appropriate model and approach for rapid cluster failover.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] S. Gilbert, N. Lynch, The CAP theorem, Computer, vol. 45, no. 2, pp. 30-36, Feb. 2012. doi: 10.1109/MC.2011.389.

[2] M. Kleppmann, A critique of the CAP theorem, arXiv:1509.05393v2 [cs.DC], 2015. doi: 10.48550/arXiv.1509.05393.

[3] E. Brewer, A certain freedom: thoughts on the CAP theorem, PODC '10: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, p. 335, 2010. doi: 10.1145/1835698.1835701.

[4] E. A. Lee, S. Bateni, S. Lin, M. Lohstroh, C. Menard, Quantifying and generalizing the CAP theorem, arXiv:2109.07771v1 [cs.DC], 2021. doi: 10.48550/arXiv.2109.07771.

[5] F. D. Muñoz-Escoí, et al, CAP theorem: revision of its related consistency models, The Computer Journal, vol. 62, no. 6, pp. 943-960, June 2019. doi: 10.1093/comjnl/bxy142.

[6] A. Lewis-Pye, T. Roughgarden, Resource pools and the CAP theorem, arXiv:2006.10698v1 [cs.DC], 2020. doi: 10.48550/arXiv.2006.10698.

[7] L. Frank, R. U. Pedersen, C. H. Frank, N. J. Larsson, The CAP theorem versus databases with relaxed ACID properties, Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication (ICUIMC '14), article no. 78, pp. 1-7, Jan. 2014. doi: 10.1145/2557977.2557981.

[8] Fault-tolerance by replication in distributed systems, Reliable Software Technologies — Ada-Europe '96 (Ada-Europe 1996), conference paper, pp. 38-57, Jan. 2005.

[9] K. P. Birman, T. A. Joseph, Exploiting replication in distributed systems, NASA Contractor Report CR-186410, Jan. 1989. doi: NASA-CR-186410.

[10] B. Ciciani, D. M. Dias, P. S. Yu, Analysis of replication in distributed database systems, IEEE Transactions on Knowledge and Data Engineering, vol. 2, pp. 247-261, Jun. 1990. doi: 10.1109/69.54723.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12), pp. 1-14, University of California, Berkeley, 2012.

[12] F. Cristian, Understanding fault-tolerant distributed systems, Communications of the ACM, vol. 34, no. 2, pp. 56-78, Feb. 1991. doi: 10.1145/102792.102801.

[13] A. Luntovskyy, B. Shubyn, T. Maksymuk, M. Klymash, Highly-distributed systems: What is inside?, Proceedings of the 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, Oct. 2020. doi: 10.1109/PICST51311.2020.9467890.

[14] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum, Locality-aware request distribution in cluster-based network servers, ACM SIGOPS Operating Systems Review, vol. 32, no. 5, pp. 205-216, Oct. 1998. doi: 10.1145/384265.291048.

[15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, EuroSys '15: Proceedings of the Tenth European Conference on Computer Systems, article no. 18, pp. 1-17, Apr. 2015. doi: 10.1145/2741948.2741964.

[16] A. Holub, "The mutex and lock management," in Taming Java Threads, Apress, Berkeley, CA, 2000. doi:10.1007/978-1-4302-1129-7_3.

[17] M. Walmsley, "Semaphores," in Multi-Threaded Programming in C++, Springer, London, 2000. doi:10.1007/978-1-4471-0725-5_5.

[18] S. Plagnol, "Beyond mutexes, semaphores, and critical sections," in Embedded Real Time Software and Systems (ERTS2012), Toulouse, France, Feb. 2012. ⟨hal-02263445⟩.

[19] M. Petrescu, "Replication in Raft vs Apache Zookeeper," in Soft Computing Applications (SOFA 2020), Advances in Intelligent Systems and Computing, vol. 1438, Springer, 2023, pp. 426–435. doi:10.1007/978-3-030-55556-4_44.

[20] H. Howard, R. Mortier, "Paxos vs Raft: Have we reached consensus on distributed consensus?," Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), Article No. 8, pp. 1–9, April 2020. doi:10.1145/3380787.3393681.

[21] F. Junqueira, B. Reed, ZooKeeper: Distributed Process Coordination, O'Reilly Media, Inc., 2013.

[22] M. Kotov, S. Toliupa, V. Nakonechnyi, "Replica State Discovery Protocol Based on Advanced Message Queuing Protocol," Cybersecurity: Education, Science, Technique, vol. 3, no. 23, 2024. doi:10.28925/2663-4023.2024.23.156171.

[23] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, Oct. 2017, pp. 426-435. doi:10.1109/SysEng.2017.8088251.

[24] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, "Performance evaluation of RESTful web services and AMQP protocol," in 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), Da Nang, Vietnam, Jul. 2013. doi:10.1109/ICUFN.2013.6614932.

[25] N. Q. Uy and V. H. Nam, "A comparison of AMQP and MQTT protocols for Internet of Things," in 2019 6th NAFOSTED Conference on Information and Computer Science (NICS), Hanoi, Vietnam, Dec. 2019. doi:10.1109/NICS48868.2019.9023812.

[26] A. Prajapati, "AMQP and beyond," in 2021 International Conference on Smart Applications, Communications and Networking (SmartNets), Glasgow, United Kingdom, Sep. 2021. doi:10.1109/SmartNets50376.2021.9555419.

[27] I. N. McAteer, M. I. Malik, Z. Baig, and P. Hannay, "Security vulnerabilities and cyber threat analysis of the AMQP protocol for the Internet of Things," in Australian Information Security Management Conference, 2017. ISBN: 978-0-6481270-8-6.

[28] A. Stanik, M. Höger, and O. Kao, "Failover pattern with a self-healing mechanism for high availability cloud solutions," in 2013 International Conference on Cloud Computing and Big Data, Fuzhou, China, Dec. 2013. doi:10.1109/CLOUDCOM-ASIA.2013.63.

[29] W. Lin, H. Jiang, N. Zhao, and J. Zhang, "An optimized multi-Paxos protocol with centralized failover mechanism for cloud storage applications," in Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2018), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 268, Feb. 2019, pp. 610–625. doi:10.1007/978-3-030-30146-8_41.

[30] P. Somasekaram, R. Calinescu, and R. Buyya, "High-availability clusters: A taxonomy, survey, and future directions," Journal of Systems and Software, vol. 187, May 2022, 111208. doi:10.1016/j.jss.2021.111208.

[31] C. K. High-availability (HA) PostgreSQL Cluster with Patroni. Medium, Jan 14, 2024. URL: https://medium.com/@chriskevin_80184/high-availability-ha-postgresql-cluster-with-patroni-1af7a528c6be.

[32] Percona Distribution for PostgreSQL, "High availability." Percona Documentation, version 15.8. URL: https://docs.percona.com/postgresql/15/solutions/high-availability.html.

[33] "Introduction to pg_auto_failover." pg_auto_failover Documentation. URL: https://pg-auto-failover.readthedocs.io/en/main/intro.html.

[34] L. Fittl, "Introducing pg_auto_failover: Open source extension for automated failover and high-availability in PostgreSQL." Microsoft Azure Blog, May 6, 2019. URL: https://opensource.microsoft.com/blog/2019/05/06/introducing-pg_auto_failover-postgresql-open-source-extension-automated-failover-high-availability/.

[35] A.E. Nocentino, B. Weissman, "Kubernetes Architecture," in SQL Server on Kubernetes, Apress, Berkeley, CA, 2021. doi:10.1007/978-1-4842-7192-6_3.

[36] Kubernetes Documentation, "Cluster Architecture," Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/architecture/.

[37] L. Larsson, H. Gustafsson, C. Klein, and E. Elmroth, "Decentralized Kubernetes Federation Control Plane," 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 2020, pp. 354-359. doi:10.1109/UCC48980.2020.00056.