Partible State Replication for Industrial Controller Redundancy

Bjarne Johansson*†, Olof Holmgren*, Thomas Nolte†, Alessandro V. Papadopoulos†
* ABB Process Automation, Process Control Platform, Västerås, Sweden, {bjarne.johansson, olof.holmgren}@se.abb.com
† Mälardalen University, Västerås, Sweden, {thomas.nolte,alesandro.papadopoulos}@mdu.se

***Abstract*—Distributed control systems are part of the often invisible backbone of modern society that provides utility services like water and electricity. Their uninterrupted operation is vital, and unplanned stops due to failure can be expensive. Critical devices, like controllers, are often duplicated to minimize the service stop probability, with a secondary controller acting as a backup to the primary. A seamless takeover requires that the backup has the primary's latest state, i.e., the primary has to replicate its state to the backup. While this method ensures high availability, it can be costly due to hardware doubling. This work proposes a state replication solution that doesn't require the backup to store the primary state, separating state storage from the backup function. Our replication approach allows for more flexible controller redundancy deployments since one controller can be a backup for multiple primaries without being saturated by state replication data. Our main contribution is the partible state replication approach, realized with a distributed architecture utilizing a consensus algorithm. A partial connectivity-tolerant consensus algorithm is also an additional contribution.**

I. INTRODUCTION

Distributed Control Systems (DCS) are the backbone of many large-scale automation solutions, especially in critical domains where unplanned downtime can have significant financial and operational repercussions. Central to these systems are controllers with redundancy mechanisms that minimize the risk of unplanned downtime. Commonly, this redundancy is achieved through hardware duplication, where one controller operates as the active primary and another as a standby backup, ready to take over in case of a primary failure. In a DCS setting, controllers are often termed Distributed Controller Nodes (DCN), a term interchangeable with 'controller' in this paper.

With the advent of Industry 4.0, there's been a notable transition from specialized fieldbuses to more flexible networked solutions, enhancing system interconnectivity. Networked-based architecture allows flexible redundancy schemes, such as one backup for many primaries, a redundancy pattern that increases fault tolerance with reduced hardware footprint. However, seamless backup takeover requires state replication from the primaries, a task the backup's bandwidth could limit.

Central to the DCS is the DCN-driven control application, which manages the physical process's state. The application samples the process state by reading values from input I/O connected to sensors and determines appropriate actions based on these samples. These actions then dictate the output values sent to the output I/O, interfacing with the real-world process. Fig. 1 shows this sequence—often described as 'copy-in, execute, and copy-out.'

This work is funded by The Knowledge Foundation (KKS), project ARRAY and SACSys, and The Swedish Foundation for Strategic Research (SSF).

As mentioned, a primary DCN replicates the redundant DCN control application state to the backup. The application state data size depends on the application and can vary between a few bytes to many megabytes.

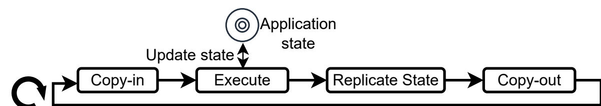


Fig. 1: Typical control application task execution steps.

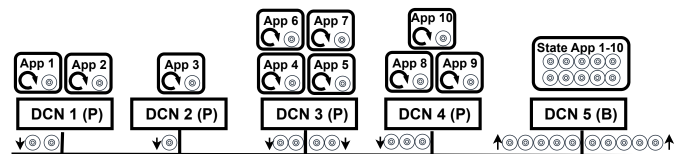


Fig. 2: Naive state replication approach - the backup stores every application's state.

The latest state is needed to resume the operation of an application seamlessly. Fig. 2 depicts a redundancy deployment with multiple primaries and one backup using a naive state replication where all applications replicate their state to a single backup (DCN 5). The network capacity of the backup in terms of bandwidth becomes a potential bottleneck in such a deployment.

The required bandwidth is the replicated state data size multiplied by replication frequency. If A is the set of all redundant applications, sds_i is the size of the state data, and rt_i is the allowed state replication time for application i and $i \in A$. Then, Eq. 1 shows the total bandwidth required, bw, to replicate the states. If bw is larger than the bandwidth the backup provides, the puzzle is not solvable with the naive approach.

$$bw = \sum_{i=1}^{|A|} sds_i \left(\frac{1}{rt_i} \right) \tag{1}$$

This paper aims to answer the question of how the aggregate state replication data sent to the backup DCN can be reduced. By reducing this traffic, the risk of redundancy arrangements being limited due to backup network resources will also

decrease. To address this challenge, the paper proposes a new method called Partible State Replication (PSR), which separates the backup role from the recipient of primary application states. With PSR, states can be replicated to any DCN, serving as our direct contribution. Further enhancing fault tolerance, we decentralize storage allocation handling for PSR, leading to our secondary contribution: a consensus protocol targeting a DCN cluster.

The paper is organized as follows: Sec. II reviews related work. Sec. III provides an overview of PSR, elaborated upon in Sec. IV. The consensus protocol is detailed in Sec. V, while Sec. VI presents our implementation and evaluation findings. Finally, we conclude and discuss future work in Sec. VII.

II. Related work

Passive standby redundancy is the prevailing DCN redundancy mechanism [1], [2], [3]. Prior research has explored diverse DCN redundancy concepts, including cloud-hosted redundant controllers, orchestrator utilization, and architectures centered on forming redundant solutions from non-redundant Commercial Off-The-Shelf (COTS) Programmable Logic Controllers (PLC) [4], [5], [6].

Achieving standby redundancy via hardware duplication is costly, especially with high-end DCNs designed for redundancy [6]. In contrast to the related work mentioned above, we propose a partible synchronization between primary and backup to enable a cost-effective redundancy. This partible approach entails segregating state storage from the backup role, further detailed in Sec. III.

The data replication research landscape is vast; examples include deduplication and placement strategies [7], [8], [9]. Our contribution is a placement-enabling architecture aimed at reducing the network resource load on backup nodes. Exploring optimal placements for redundant DCN applications remains an avenue for future research. Like our work, Bakhshi et al. [10] provide a distributed persistent state storage architecture for containerized applications. However, their solution replicates the states to all nodes, likely increasing bandwidth demand.

PSR is a decentralized distributed system. Common in fault-tolerant distributed systems is active replication using Replicated State Machines (RSM) synchronized using a replicated request log [11], [12]. Consensus protocols, like the well-known Paxos, ensure ordered delivery of requests to the RSMs [13], [14]. While influential, Paxos is intricate; hence, Raft offers a simpler alternative [15]. Raft divides time into terms, each with a dedicated leader. Another quite well-known consensus protocol is Viewstamped Replication (VSR), which employs views comparable to Raft's terms [16], [17], [18].

Omni-Paxos, a variant of Paxos, addresses a shortcoming in protocols like Raft, Paxos, and VSR, which can lose progression under partial connectivity scenarios [19]. An example of partial connectivity is a three-peer system where only one peer connects to all others, inhibiting direct communication between the two remaining peers. This situation can hinder progress in VSR and Raft. Omni-Paxos resolves this by implementing Quorum-Connected (QC) as a criterion for leader election. QC means a connection to a quorum of peers.

ZooKeeper Atomic Broadcast (ZAB) is a replication protocol that prioritizes performance by relaxing the guaranteed order slightly [20], [21], [16].

PSR and the above protocols assume fail-stop semantics; Castro et al. propose a practical version of a Byzantine fault-tolerant protocol [22].

Industrial control systems—especially those necessitating redundancy—prioritize high dependability [6]. Solutions tolerant to partial connectivity are more likely to show higher availability. Also, as argued by Ongaro et al. [15], an algorithm where there is one dedicated leader, and that leader is the most up-to-date partaker, is easier to understand. Hence, with inspiration from the abovementioned protocols, we propose a consensus protocol that, like Omni-Paxos, is partial connectivity tolerant but built upon a VSR foundation instead of Paxos. VSR, like Raft, ensures that the leader is up-to-date with the latest entries after synchronization, and VSR deterministically elects a leader and ensures that this is the only leader. We call the proposed protocol Viewstamped Replication - Quorum Connected (VSR-QC), further described in Sec. V.

III. Partible State Replication

This section provides a high-level introduction and overview of PSR, the problem addressed with PSR, the assumptions, and requirements.

Overview: In the naive state replication method, the backup is required to manage the aggregate bandwidth necessary for synchronizing the state of every application for which it serves as the backup DCN, as detailed in Eq. 1. PSR reduces the state replication bandwidth required from a backup by distributing the replicated state storage amongst the DCNs in the DCN cluster. The DCN cluster is the set of DCNs that forms the resource pool available for state replication. Fig. 3 demonstrates the distributed state storage facilitated by PSR, where DCNs 1-4 function as primary DCNs, managing the primary instances of the applications, while DCN 5 acts as a backup for all these primaries.

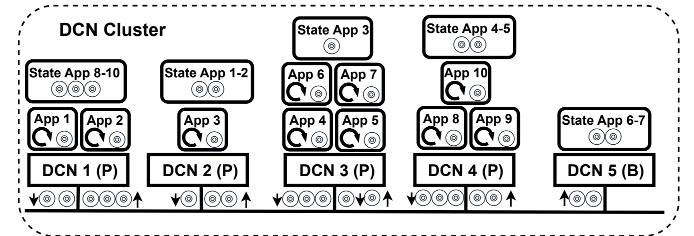


Fig. 3: An example of a PSR using DCN cluster of five DCNs. DCN 1-4 host primary applications, and DCN 5 is backup for DCN 1-4. All primary applications replicate their state somewhere, but not all to the same DCN.

Assumptions: This work does not cover the allocation of applications to the DCNs, nor the allocations of DCNs to

a DCN cluster. We assume applications reside in persistent storage and start upon DCN startup. Additionally, the backup is assumed to have adequate resources to maintain applications on warm standby. I.e., allow the backup application instance to detect a failure of the primary instance and resume the primary state. The system operates under a non-Byzantine failure-recovery semantic.

Requirements: Each primary application instance must have a designated location for state storage, and each backup instance needs to be able to access this storage. Efficient state fetching and storing are crucial, especially in applications with short cycle times. "Short" is relative, but shorter is better for faster control loops, with 500 ms being a common minimum [4].

PSR must avoid central mechanisms for pairing application state storage. The cluster should operate independently and recover from faults without a central server, enhancing fault tolerance.

PSR must provide the capability to add (register) and remove (deregister) applications for state storage, i.e., provide dynamic properties. Active applications request and consume available storage; removed applications return storage. Similarly, DCNs register their storage capability upon activation and update it upon change, ensuring they don't become over-allocated. In other words, DCNs report their available capacity, applications declare their resource needs when registering, and PSR tries to find a matching storage for each application.

IV. Architecture

This section outlines the PSR architecture, detailing its internal components and their interactions in key use cases. Although we refer to DCN, this term is interchangeable with any computing device. The focus is on storage and state replication, but the described principles and mechanisms can be applied to other scenarios, like allocating application execution based on available computational resources.

A. Components

The PSR architecture comprises three main components, as depicted in Fig. 4: (i) Application Redundancy Functions (ARF), (ii) Partible State Replication Manager (PSRM), and (ii) Cluster Consensus Manager (CCM). Sec. V provides a detailed discussion of the CCM. For the context of this section, it suffices to understand that the CCM provides consistent replication across all DCNs via a consensus algorithm.

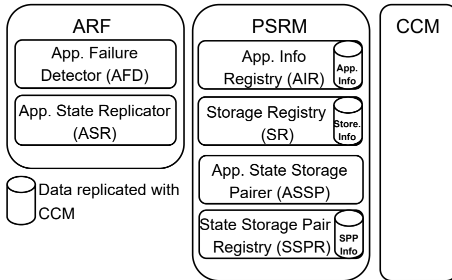
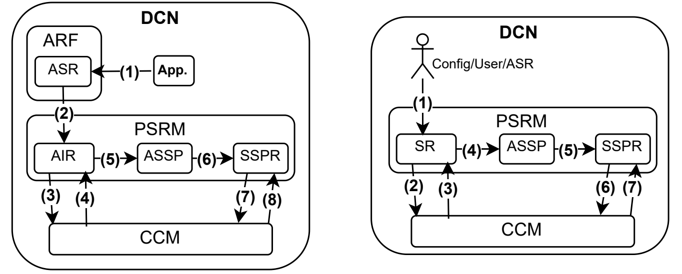


Fig. 4: High-level architecture view of the PSR constituting components.



(a) Application registering (or deregistering) (adding or removing).

(b) Storage (DCN) registering (or deregistering) (adding or removing).

Fig. 5: Component interaction when (a) adding/removing an application or (b) storage when a DCN startup.

The ARF offers redundancy functions essential for a redundant application, including failure detection and state replication. It comprises two sub-components: the Application Failure Detector (AFD) and the Application State Replicator (ASR). AFD manages failure detection, where the backup's AFD monitors the primary and alerts in case of failure. ASR is responsible for state replication, transferring the application state to the designated storage, and enabling the backup application to fetch the state to resume with the primary's latest state if needed.

The PSRM consists of four sub-components: the Application Info Registry (AIR), the Storage Registry (SR), the Application State Storage Pairer (ASSP), and the State Storage Pair Registry (SSPR).

The AIR's responsibility is threefold. The first is to gather the local applications' state replication needs. The second is replicating the collected information in the cluster. The third is to keep a registry of all the application's state storage needs in the cluster. See Fig. 5a.

The SR's responsibility is also threefold, like AIR. SR is the AIR counterpart for storage. It gathers the storage capability provided by the local DCN, replicates this information in the cluster, and holds a registry with all the available storage in the cluster. See Fig. 5b.

The ASSP does the actual paring; it uses the application's state replication needs to find a state storage for each application. ASSP uses the information in AIR and SR to do the pairing. The ASSP instance running on the DCN with the leader CCM does the pairing; see Sec. V.

The SSPR keeps the registry of application storage allocation, i.e., the application-storage pair. The ASSP updates the SSPR if any change in the requested or available storage impacts the pairing made. Such as the adding or removing of DCNs or applications. SSPR uses the CCM to replicate the paring information in the cluster. See Fig. 5a.

B. Use cases

This section shows the interaction between the different components for four key use cases.

1) Application start (registering/deregistering): A redundant application registers itself with the ASR at startup (1);

(a) Normal operation.

(b) Failing DCN scenario.

Fig. 6: Component interaction during (a) normal operation and (b) failure of DCN running a primary instance of an application.

see Fig. 5a. ASR registers the application information in PSRM through AIR (2). AIR uses the CCM to replicate the application information in the cluster (3). When the CCM has successfully replicated application information in the cluster, all AIR instances are informed and update their registry (4). AIR notifies the ASSP when there is a change in the registered applications (5). A change in the CCM leader state also triggers ASSP to evaluate the current pairings. The ASSP located with the CCM leader pairs the application with storage located on a DCN other than the primary application. Once ASSP has made a paring (or removed one), it asks SSPR to update (6). SSPR requests the CCM to replicate the updated pairing in the cluster (7). When the pairing information has replicated in the cluster, the CCM in all DCNs informs the SSPR of the changed pairing information (8).

2) DCN startup (storage registering/deregistering): A DCN providing storage registers its information upon startup, as shown in Fig. 5b. It informs the SR in PSRM (1), and the SR replicates this storage information using CCM (2). CCM ensures all SR instances are updated (3). When SR detects a change in information, it notifies ASSP (4). If the storage update requires a change in the application state storage pairing, ASSP in the DCN with the leader CCM updates and passes this information to SSPR (5). SSPR then replicates this pairing across the cluster using CCM (6), which distributes the updated pairing to SSPR on all DCNs (7).

An alternative scenario occurs when storage becomes unavailable, triggering ASR to mark the storage as unavailable, leading ASSP to reassign alternative storage to the impacted application.

3) Normal operation: Fig. 6a illustrates the standard operation process. The application communicates its status to the AFD (1). The AFD in DCN 1 then sends a heartbeat to its backup counterpart in DCN 3 (2). As long as the AFD in DCN 3 receives confirmation that DCN 1's primary instance is operational (3), the backup application in DCN 3 remains on warm standby. The primary application instance replicates its state through ASR (4), which retrieves storage information from SSPR (5-6) and then stores the application state data, for

Fig. 7: The VSR-QC state machine.

example, in DCN 2 (7). The ASR in the backup continuously verifies access to the storage and the validity of the state data, retrieving storage location from SSPR (8-9) and checking its accessibility and data validity (10). Any change in storage accessibility is reported to PSRM and SR (11), initiating the process shown in Fig. 5b. Both primary and backup ASRs perform this accessibility check.

4) DCN failure: Fig. 6b depicts a DCN failure and the subsequent actions by the backup application instance in DCN 3 to take the primary role. If the primary application instance fails to trigger the AFD, the warm standby application in DCN 3 becomes the primary (1-4). Upon assuming the primary role (4), it requests the latest state from the ASR in ARF (5). The ASR then inquires about SSPR for the storage location (6-7), retrieves the latest state from the storage (e.g., DCN 2) (8-9), and supplies the retrieved state to the application (10).

V. CCM - Cluster Consensus Manager

PSR utilizes CCM for cluster consensus, employing VSR-QC as its consensus protocol. This section details VSR-QC and outlines the CCM sub-components for implementing CCM with VSR-QC.

A. VSR-QC protocol

VSR-QC, influenced by VSR [18], Raft [15], and Omni-Paxos [19], operates under a non-Byzantine failure-recovery model. It assumes protocol instances don't continuously crash and recover. Messages may be lost, but the system is generally synchronous, following a partially synchronous model. When describing the protocol, we use the term VSR-QC instance rather than server, node, or DCN since VSR-QC can run multiple instances per node / DCN.

Like Raft, a VSR-QC instance can be in either of the three states $Follower$, $Electing$, or $Leader$, shown in Fig. 7.

The leader is the VSR-QC instance in state $Leader$ and it is the driver of the replication. The leader is elected from the set of followers. A follower can only become a leader if it is QC and elected by a majority. Each instance has a unique and persistently stored identification, id.

Followers are VSR-QC instances in state $Follower$. Each VSR-QC instance maintains its own instance of the replicated log. The replicated log is an ordered list of replicated requests. The log is orderly replicated to all functioning VSR-QC instances in the replica group. The replica group is the set of VSR-QC instances forming the distributed replication. VSR-QC as VSR, Raft, and Paxos tolerate f faulty VSR-QC

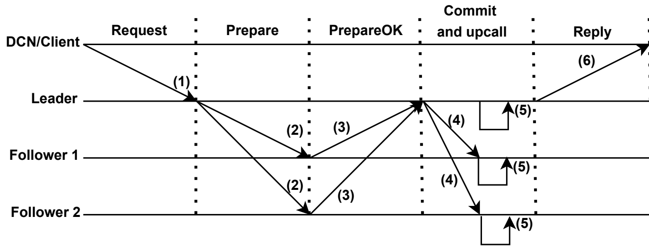


Fig. 8: Normal replication flow of VSR-QC.

instances in the replica group. Hence, $2f+1$ is the minimum replica group size to be f fault tolerant.

VSR-QC utilizes the view concept [18], [15]. A view is an integer, $ViewNumber$, incremented each time a leader election process is started.

The protocol's functionality is explained through four scenarios: (i) normal operation, (ii) leader election, (iii) synchronization, and (iv) failure detection, concluded with a brief discussion on configuration.

1) Normal operation: The normal operation of VSR-QC is similar to VSR [18], illustrated in Fig. 8 and summarized below.

A DCN (or other client) issues a request by sending a $\langle Request, rid, msg\rangle$ message to the leader (1). Step 3 in Fig. 5a is a PSR request example. The rid is a tuple comprising the client ID and a request number, forming a unique request ID, rid. The rid prevents double processing of requests in case of a leader failure while a request is uncompleted. The payload of the request is msg. Unprocessed requests, identified by rid, prompt the leader to dispatch a $\langle Prepare, v, n, r, m\rangle$ message to all followers (2), where v is the current $ViewNumber$, n the $OpNumber$, and r and m are the rid and msg. The $OpNumber$ is an integer incremented by the leader for each finalized request.

Followers process $Prepare$ messages sequentially in $OpNumber$ order. Upon having all prior log entries, a follower adds the new entry, stores the rid, and sends a $\langle PrepareOK, v, n\rangle$ back to the leader. If preceding entries are missing, the follower attempts synchronization (see Sec. V-A3), withholding $PrepareOK$ until all previous and current entries are stored in the log.

The leader waits for replies from f followers with $PrepareOK$ (f followers plus the leader constitute a majority). After receiving at least f $PrepareOK$, the request is stored in the replicated log of a majority, and the leader issues a commit with the $\langle Commit, v, k\rangle$ message (4), where k is the $CommitNumber$. The $CommitNumber$ is the highest $OpNumber$ that has been committed. Committed entries can not be changed or removed.

After sending the $Commit$, the leader performs the upcall to the distributed application (5). The upcall is the term for passing the request to the distributed application layer, PSR, in our case, exemplified in step 4 in Fig. 5a. The followers issue the upcall when they receive the $Commit$ if all previous entries are committed.

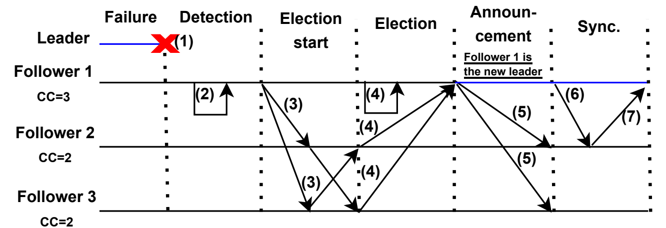


Fig. 9: Leader election.

2) Leader election: As mentioned in Sec. II, VSR does not handle partial connectivity because it requires QC voters [18], [19]. VSR-QC is partial connection tolerant because VSR-QC uses QC as leader criteria, but it does not require that the voters are QC, similar to Omni-Paxos [19].

The failure detection catches leader failure and determines which instances are QC or not; see Sec. V-A4. Fig. 9 shows a leader election sequence that starts with the failure of the current leader (1). Eventually, the failed leader is detected by another VSR-QC instance, in this case, Follower 1 (2).

Follower 1 detects the failed leader and initiates an election by entering the $Electing$ state, increments its $ViewNumber$, sends an $\langle ElectionStart, v\rangle$ message to all other VSR-QC instances (3), and starts the election timeout timer. The $ElectionStart$ message contains v set to the current (just incremented) $ViewNumber$ of the VSR-QC instance.

Follower 2 receives the $ElectionStart$ message from Follower 1 and enters the $Electing$ state if the received v is higher than $ViewNumber$ and assigns v to its $ViewNumber$. Follower 2 also sends an $ElectionStart$ message to all other instances when entering the $Electing$ state and starts the election period timer.

Suppose no leader has presented itself directly via the $ElectionComplete$ message or indirectly via the heartbeat. In that case, when the election period timer expires, the election process restarts by incrementing the $ViewNumber$ again and re-entering the $Electing$ state.

If a new $ElectionStart$ message with v higher than $ViewNumber$ is received before the election period has ended, the election period restarts and the above-described actions repeat.

In $Electing$, VSR-QC instances cast one vote per election period ($ViewNumber$) using $\langle ElectionVote, lid, v, n, k\rangle$, addressed to the prospective new leader. This message includes lid (the prospective leader's id), $ViewNumber$, $OpNumber$, and $CommitNumber$, as depicted in step (4) in Fig. 9.

Voting is based on the Connectivity Count (cc), updated by the failure detection; see Sec. V-A4. The cc reflects the number of connected VSR-QC instances. Votes are given to the instance with the highest cc over the QC limit. In case of equal cc values, the tie is broken by id, favoring the lowest id.

A VSR-QC instance receiving an $ElectionVote$ message enters the $Electing$ state if the received v is higher than $ViewNumber$ and performs the above-described action when entering state $Electing$. In state $Electing$ it counts all

$ElectionVote$ messages with $v = ViewNumber$ received within the election period as valid votes. If it gets an $ElectionVote$ with a v higher than its $ViewNumber$, it re-enters $Electing$, resets the vote count, restarts the election timer, and updates its $ViewNumber$ to v.

A VSR-QC instance in $Electing$ that receives $f + 1$ valid votes accepts that it is the new leader and enters the $Leader$ state and announces itself as the new leader by sending out a $\langle ElectionComplete, lid, v, n, k\rangle$ message where lid is id, v is $ViewNumber$, n the $OpNumber$, and k the $CommitNumber$. Fig. 9 (4) shows Follower 1 obtaining the majority of votes, becoming the new leader, and making the announcement with the $ElectionComplete$ message (5). The other VSR-QC instance in state $Electing$ enters the $Follower$ state when receiving $ElectionComplete$, or a $Heartbeat$ indicating a leader, with a v equal or higher than $ViewNumber$.

A leader that loses QC leaves the leader role and initiates a new election by sending the $ElectionStart$ message; see Fig. 7.

The new leader must ensure it has the latest log entries, which it does by requesting the entries it is missing, if any, from the most up-to-date follower. The n k in the $ElectionVote$ message has informed the leader about the most up-to-date follower, and it is to that follower the leader requests a synchronization, step (6) and (7) in Fig. 9. Synchronization is further described in Sec. V-A3.

3) Synchronization: VSR-QC requires no persistent storage to store the log; it assumes that a majority never fails at the same time. However, if desired, VSR-QC, as the VSR inspiration, can be modified to use persistent storage and reduce the synchronization needed upon recovery [18].

This section describes synchronization steps to bring a VSR-QC instance that, for whatever reason, has become outdated in synchronization again. We divided the synchronization description into three steps: (i) detection, (ii) follower synchronization, and (iii) (newly elected) leader synchronization.

A follower detects that it is not synchronized when receiving $Prepare$ or a $Heartbeat$ message with a v and n higher than the follower's $ViewNumber$ and $OpNumber$. A leader detects lagging when it has received a majority of votes by comparing the v and n in the received $ElectionVote$ message with its $ViewNumber$ and $OpNumber$.

A follower that is out of synchronization uses the $\langle SyncMeReq, i\rangle$ message where i is the id of the follower. The receiving VSR-QC instance, regardless of its current role, will reply with the $\langle SyncMeReply, v, n, k, l\rangle$ message, where v,n,k, and l is the $ViewNumber$, $OpNumber$, $CommitNumber$, and log entries of VSR-QC instance i.

The leader is the most updated VSR-QC instance since it is the designated receiver of client requests and is the driver of advancement. However, a new leader might not possess the latest entries immediately after the election. Therefore, as described in Sec. V-A2, a newly elected leader's first step is synchronizing itself with the latest entries. The information in the $ElectionVote$ concludes which is the most updated VSR-QC instance in the majority. The leader sends a $SyncMeReq$ message to one of the most updated VSR-QC instances to retrieve the log and perform any missing commits and upcalls. After synchronization completion, the leader starts accepting and processing client requests.

Do note there are several ways to make the synchronization handling more efficient; some are discussed in the VSR description [18].

4) Failure detection: All the VSR-QC instances send $Heartbeat$ to one another. A concrete realization example of such an exchange is a multicast group dedicated to failure detection within the replica group.

The $\langle Heartbeat, v, n, k, i, p, c\rangle$ message conveys each instance's replication and connectivity status. Replication status includes $ViewNumber(v)$, $OpNumber(n)$, $CommitNumber(k)$, instance id (i), and the leader's id (p), with p set to zero if no leader is identified. Hence, this message gossips the leader's identity. Connectivity status is represented by c, indicating which instances are connected in a simple bit-field format, where each bit corresponds to a VSR-QC instance.

Based on these heartbeats, VSR-QC instances update their Connectivity Count (cc). Only QC followers can start elections, and votes are given to the highest cc, averting continuous re-elections. The leader will relinquish its role if it loses QC status.

5) Configuration: Adding and removing VSR-QC instances to the replica group is not covered for page conservation reasons. We envision that the mechanisms VSR uses for adding/removing members are suitable for VSR-QC as well [18].

B. Components

Fig. 10 displays the components of CCM, with the CCM Service Abstraction (CSA) acting as an interface. CSA's role is to provide an easy-to-use interface to the replicated services like PSR and abstract the underlying consensus protocol. CSA functions include issuing requests, upcall registration, and leader checks. CCM's architecture, as shown in Fig. 10, comprises four sub-components: (i) Leader Elector (LE), (ii) Failure Detector (FD), (iii) Group Member Manager (GMM), and (iv) Log Replicator (LR). LE and FD handle leader election and failure detection (Sec. V-A2 and V-A4), GMM manages and updates replica group membership, and LR, detailed in Sec. V-A1, manages replication, performs upcalls for new log entries, and handles synchronization.

VI. Implementation, Execution and result

A. Implementation

We developed a PSR prototype for VxWorks based on the architecture described in the previous sections. The prototype is available on GitHub [23].

The prototype version of the ASSP application-storage pairing algorithm pairs applications with available storage based on the sequence of their registration. The primary goal of the

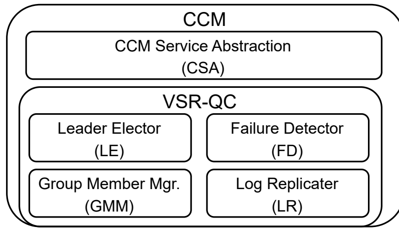


Fig. 10: CCM Components.

algorithm is to find storage on another DCN than the DCN hosting the primary application instance; the secondary goal is to find the least utilized storage. The prototype version pairs an application requiring storage with the first found unused storage. If no unused storage exists, it searches for the first storage used by only one other application. If that fails, no storage exists for the application. The exploration of more advanced pairing algorithms is future work.

The ARF implementation of the AFD uses a UDP-based heartbeat message protocol. The heartbeat messages are sent on requests from the application. The ASR state replication uses a UDP-based message encapsulating the most recent state as reported by the application. This message is directed to the state-storing DCN, exemplified by DCN 2 in Fig. 6a. The ASR is responsible for operating a storage server on the storage DCN. This server receives the incoming state messages and preserves the most recent within RAM.

The prototype includes a Test Application (TAPP), a redundant application that can function as either primary or backup. We utilize the TAPP to assess PSR's performance and to draw comparisons with the naive state replication approach depicted in Fig. 2. In its backup role, the TAPP remains in warm standby, meaning it's loaded into RAM and primed to switch to the primary role when the AFD detects a failure of the original primary instance of TAPP. The primary TAPP instance requests that the AFD send heartbeats and the ASR transmit its latest state to the storage. The state includes a sequence number that is incremented in each iteration. The state size and period time are adjustable.

When a TAPP instance takes the primary role, it requests the latest state using its local ASR. The local ASR, in turn, sends a state request message to the ASR on the storage DCN to obtain the most recent state, as illustrated in Fig. 6b.

To conduct failover testing, the primary TAPP instance is instructed to cease operation, which prompts the backup instance to assume the primary role upon the AFD heartbeat timing out. In transitioning to the primary role, the TAPP anticipates a specific state from the ARF. Knowing the sequence number of the latest state before a commanded shutdown, the TAPP can confirm whether it has successfully retrieved the most recent state.

B. Setup and Execution

We utilize virtual machines running on VMware 17 as DCNs. Each virtual machine has one CPU, one core, and two GB of RAM. These machines are hosted on a Lenovo ThinkPad P15, featuring a 2.7 GHz Intel I7 processor and 48 GB of RAM. The bandwidth of the virtual network interface connecting the virtual machines is limited to 1024 Kbps. This limitation is imposed to make evaluations feasible in a virtualized environment without overloading the host computer. On these virtual DCNs, we run the PSR prototype, including the TAPP, on VxWorks 21.07.

Our experiments involve five different redundancy patterns: (i) one primary and one backup (1p), (ii) two primaries and one backup (2p), (iii) three primaries and one backup (3p), (iv) four primaries and one backup (4p), and (v) five primaries and one backup (5p). The primaries run the TAPP instance in the primary mode, while the backups host the backup TAPP, as depicted in Fig. 3. In the 1p configuration, there are four TAPP instances: two running as primaries on the primary DCN and two as warm standby backups. This pattern continues, resulting in 8 instances for 2p, 12 for 3p, 16 for 4p, and so on.

Each TAPP instance operates on a 40-millisecond cycle, sending a heartbeat (via AFD) and replicating its state (using ASR). The volume of state data replicated in each cycle is adjustable. We begin our tests with 128 bytes of state data, increasing it in 128-byte increments until network resource overutilization on one or more DCNs causes the test to fail. At each increment, we simulate a controlled failure of the TAPP instance on DCN 1 by commanding it to stop.

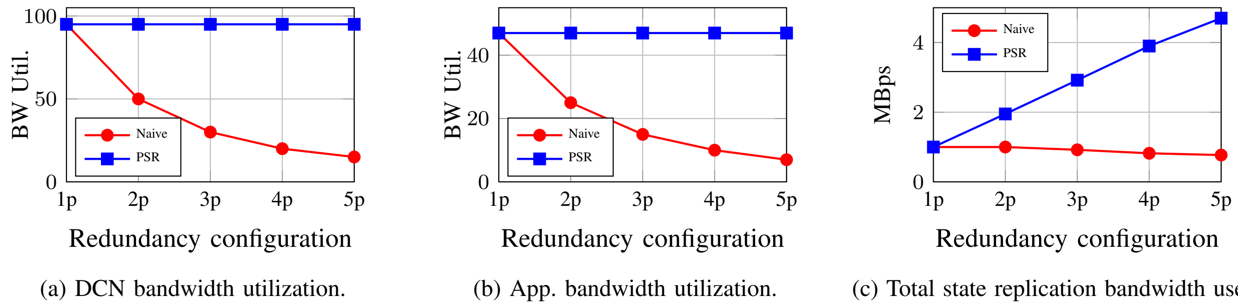
A test is considered to have failed when a backup TAPP instance erroneously transitions to the primary role due to AFD not receiving heartbeats – a consequence of network congestion from state replication traffic. Similarly, a test fails if a triggered failure doesn't result in the backup retrieving the latest state. We define the point at which tests begin to fail as the 'failure threshold.'

In our tests, we run the system in PSR mode, employing PSR for state replication. Each participating virtual DCN contributes storage for two TAPP instances in this mode. Therefore, in the 1p configuration, two DCNs provide storage. However, for fault tolerance, the primary can only use storage on the backup DCN, not on itself. In the 2p setup, three DCNs offer storage, assigned to TAPP by PSR, demonstrating the concept illustrated in Fig. 3. In contrast, the naive mode only uses the backup for storage, as shown in Fig. 2.

C. Result

The graphs presented in Fig 11 illustrate the bandwidth usage for state replication under functioning redundancy, the TAPP state replication increment **before** the failure threshold. In other words, the graphs highlight the point at which an additional 128-byte increment in the TAPP state data usage leads to system failure. Thus, these graphs offer insights into the differing aspects of the state replication bandwidth threshold for both naive state replication and PSR.

For the 1p configuration, the state replication threshold is identical between PSR and the naive approach. This is because, in a 1p setup, the naive and PSR methods replicate the state to the only backup available. However, as illustrated in Fig. 11a, the bandwidth available to each DCN decreases as we



(a) DCN bandwidth utilization. (b) App. bandwidth utilization. (c) Total state replication bandwidth used.

Fig. 11: Graphs showing the threshold bandwidth utilization per DCN, application, and the total state replication bandwidth.

expand the configuration using the naive approach. In contrast, with PSR, the bandwidth remains consistent regardless of the number of DCNs since each added DCN also contributes storage, as shown in Fig. 11a.

Fig. 11b displays the bandwidth utilization per application. In our experiment, we consistently deployed two TAPP instances per DCN, meaning the threshold bandwidth utilization for each application is effectively half that of the DCN.

Finally, Fig. 11c depicts the total bandwidth used for state replication across all applications in the various configurations. Notably, the total bandwidth usage for PSR increases as more configurations are added. This increase is attributed to each newly added DCN hosting both the TAPP and provides storage. Conversely, in the naive approach, where only the backup provides storage, the total bandwidth usage slightly decreases. This decrease is likely due to the increased number of heartbeat messages and overhead.

VII. Conclusion and Future Work

This paper introduced an architecture that separates state replication storage from backup in a decentralized system, employing the VSR-QC consensus protocol to maintain consistency. We evaluated the state replication capacity, comparing PSR with naive state replication methods. Our results show that PSR significantly increases the feasible state replication data volume, enabling a single DCN to back up multiple primaries.

Future research goals include bounded, low-latency, state data retrieval mechanisms, and reliability modeling to find cost-efficient deployments for real applications that satisfy given reliability targets. Another future research possibility is optimizing application and state storage pairing, given available resources and response time requirements. A last example of future research is investigating the integration of PSR in a context where a system like Kubernetes orchestrates the DCNs and applications.

References

[1] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[2] S. Singh, V. M. Chary, and P. A. Rahman, "Dual redundant profibus network architecture in hot standby fault tolerant control systems," in *Int. Conf. on Advances in Eng. & Tech. Research (ICAETR)*, 2014.

[3] A. Simion and C. Bira, "A review of redundancy in plc-based systems," *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, vol. 12493, pp. 269–276, 2023.

[4] T. Hegazy and M. Hefeeda, "Industrial automation as a cloud service," *IEEE Trans. Par. and Distr. Syst.*, vol. 26, no. 10, pp. 2750–2763, 2015.

[5] B. Johansson, M. Rågberger, T. Nolte, and A. V. Papadopoulos, "Kubernetes orchestration of high availability distributed control systems," in *IEEE Int. Conf. on Ind. Tech. (ICIT)*, 2022.

[6] J. Stój, "Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols," *IEEE Trans. on Autom. Science and Eng.*, vol. 18, no. 4, pp. 2035–2047, 2020.

[7] A. Shakarami, M. Ghobaei-Arani, A. Shahidinejad, M. Masdari, and H. Shakarami, "Data replication schemes in cloud computing: a survey," *Cluster Computing*, vol. 24, pp. 2545–2579, 2021.

[8] I. A. Ibrahim, W. Dai, and M. Bassiouni, "Intelligent data placement mechanism for replicas distribution in cloud storage systems," in *IEEE Int. Conf. on Smart Cloud (SmartCloud)*, pp. 134–139, 2016.

[9] H. Zhang, B. Lin, Z. Liu, and W. Guo, "Data replication placement strategy based on bidding mode for cloud storage cluster," in *Web Information System and Application Conf.*, pp. 207–212, 2014.

[10] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson, "Analyzing the performance of persistent storage for fault-tolerant stateful fog applications," *Journal of systems architecture*, vol. 144, p. 103004, 2023.

[11] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[12] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[13] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, p. 133–169, may 1998.

[14] L. Lamport, "Paxos made simple," *ACM SIGACT News*, pp. 51–58, 2001.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, pp. 305–319, 2014.

[16] R. Van Renesse, N. Schiper, and F. B. Schneider, "Vive la différence: Paxos vs. viewstamped replication vs. zab," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, pp. 472–484, 2014.

[17] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pp. 8–17, 1988.

[18] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.

[19] H. Ng, S. Haridi, and P. Carbone, "Omni-paxos: Breaking the barriers of partial connectivity," in *Proceedings of the Eighteenth European Conference on Computer Systems*, pp. 314–330, 2023.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[21] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 245–256, 2011.

[22] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, pp. 173–186, 1999.

[23] "Psr prototype implementation on github." https://github.com/Burne77a/psr. Accessed: 2023-11-22.