HELMUT SCHMIDT UNIVERSITY

DOCTORAL DISSERTATION

---

# Algorithms for High-Performance State-Machine Replication

---

*Author:*
Marius Poke

*Supervisor:*
Dr. Colin W. Glass

*A dissertation submitted*
*in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy*

*in the*

Institute for Mechatronics
Faculty of Mechanical Engineering

August 1, 2019

The oral examination took place on June 7th, 2019 and was attended by the following expert witnesses:

- Univ.-Prof. Dr.-Ing. Delf Sachau, Helmut Schmidt Universität, Fakultät für Maschinenbau (Referent)
- Dr. Colin W. Glass, Helmut Schmidt Universität, Fakultät für Maschinenbau (Betreuer und Korreferent)
- Prof. Dr. rer. nat. Dirk Pflüger, Universität Stuttgart (Korreferent)

This dissertation expands on the following papers:

- Marius Poke, Torsten Hoefler. *DARE: High-Performance State Machine Replication on RDMA Networks (Extended Version)* [139]
- Marius Poke, Torsten Hoefler, Colin W. Glass. *AllConcur: Leaderless Concurrent Atomic Broadcast (Extended Version)* [140]
- Marius Poke, Colin W. Glass. *Formal Specification and Safety Proof of a Leaderless Concurrent Atomic Broadcast Algorithm* [137]
- Marius Poke, Colin W. Glass. *A Dual Digraph Approach for Leaderless Atomic Broadcast (Extended Version)* [136]

Most of the content of these papers is included in some form in this dissertation and it is reproduced with permission from the authors.

# Abstract

Many distributed systems require coordination between the servers involved. At the same time, with increasing number of servers, these systems become more prone to single-server failures. Therefore, a high-quality service deployed on these systems must enable coordination, while tolerating failures. This can be achieved through state-machine replication. State-machine replication is fault tolerant through redundancy and coordination is achieved through strong consistency. This in turn requires ordering user requests and propagating them to all replicas, which execute them deterministically and sequentially, i.e., in total order. Guaranteeing this total order requires the execution of a distributed agreement algorithm, such as consensus or atomic broadcast. Consequently, strong consistency adds a considerable performance overhead, with typical state-machine replication request rates being orders of magnitude lower than the request rates of non-replicated services. The aim of this dissertation is to devise new efficient solutions that reduce this performance overhead.

This dissertation presents three novel algorithms—DARE, AllConcur, and AllConcur+—that stretch the performance boundaries of state-machine replication. The three algorithms target two contrasting use cases of replication—replicating a service as a mechanism to achieve high availability and replicating a service as a requirement of the application. Replication is a well-known approach to high availability. Providing strong consistency among replicas, allows a distributed service to hide failures and appear to its users as a coherent and centralized service. In such cases, scaling out state-machine replication to more than a handful of servers is not necessary. However, most existing algorithms rely on message-passing for communication. DARE is a novel high-performance state-machine replication algorithm that replaces the message-passing mechanism with remote direct memory access (RDMA) one-sided primitives. Therefore, DARE enables operators to fully utilize the new capabilities of the growing number of RDMA-capable networks.

Besides providing high availability, having multiple consistent replicas may be a requirement of the application, as is the case for distributed ledgers. In such cases, scaling out to hundreds of servers is not uncommon. Most practical state-machine replication approaches were designed mainly to enable highly available distributed services and they are not well suited for large-scale deployments. AllConcur is a novel leaderless concurrent atomic broadcast algorithm that enables state-machine replication to scale out to hundreds of servers, while achieving high performance. Besides adopting a decentralized approach, AllConcur reduces the work by replacing the traditional all-to-all communication pattern adopted by many existing algorithms with a digraph-based communication model that relies on a sparse digraph. This results in sublinear work per broadcast message and is the main reason for the high performance at scale. Moreover, AllConcur employs a novel early termination mechanism that reduces latency. As a result, AllConcur is highly competitive with regard to existing solutions at scale and outperforms standard leader-based approaches, such as Libpaxos.

The sparse digraph used by AllConcur for communication reduces the work per broadcast message. However, to reliably disseminate messages, the digraph must also be resilient; this resiliency entails redundancy, which limits the reduction of work. AllConcur+ is a novel leaderless concurrent atomic broadcast algorithm that lifts this limitation by adopting a dual-digraph approach: During intervals with no failures, it achieves minimal work by using a redundancy-free digraph. When failures do occur, it automatically recovers by switching to

the resilient digraph. As a result, by leveraging redundancy-free execution during intervals with no failures, AllConcur+ achieves significantly higher throughput and lower latency than both AllConcur and other state of the art atomic broadcast algorithms.

Overall, this dissertation addresses the challenges of designing novel algorithms with the purpose of enhancing the performance of state-machine replication for both small- and large-scale deployments. We believe that the research contributions made by the three algorithms presented will serve as a good foundation for many use cases and furthermore facilitate the future improvements of state-machine replication.

# Acknowledgements

I wish to express my genuine gratitude to Dr. Colin W. Glass. His invaluable guidance through each stage of the process made this dissertation possible. His inexhaustible patience and continuous stream of fresh ideas were instrumental in the creation and completion of this work. He never stopped challenging me and steered me in the right direction whenever he thought I needed it. But most importantly, the myriad of stimulating discussions we had together helped me fathom the intricate world of distributed agreement. Thank you for your continuous support throughout this journey.

I would also like to thank both Prof. Delf Sachau and Prof. Dirk Pflüger for supporting me in my research and the successful completion of this dissertation.

My sincere thanks go to Prof. Torsten Hoefler for introducing me to the field of distributed systems and teaching me, among many other things, that a good system design must strike a balance between theory and practice. I am very grateful to him for his guidance during the time I spent at the Scalable Parallel Computing Laboratory at ETH Zürich, which resulted in the design of DARE, and for his invaluable contributions and insights during the ensuing collaboration, which resulted in the design of AllConcur. The work that made DARE possible was supported by Microsoft Research through its Swiss Joint Research Centre.

I would like to thank Prof. Michael Resch for his support during the time I spent at the High Performance Computing Center in Stuttgart. By accepting me into the Cluster of Excellence in Simulation Technology at the University of Stuttgart, he gave me a unique opportunity to be part an interdisciplinary research association that enabled me to meet new people with whom to exchange ideas. To a certain extent, the research that this dissertation relies on was supported by the German Research Foundation (DFG) as part of the Cluster of Excellence in Simulation Technology (EXC 310/2).

I thank my former colleagues from both the Scalable Parallel Computing Laboratory at ETH Zürich and the High Performance Computing Center in Stuttgart, especially Timo Schneider, Maciej Besta, José Gracia, and Daniel Rubio Bonilla, for providing a stimulating environment, helpful discussions and being always willing to help me.

Last but not the least, I would like to express my profound gratitude to my family—my wife, my mother, and my brother—for providing me with unfailing support and continuous encouragement throughout my years of study and especially, through the ups and downs of grad school. This accomplishment would not have been possible without them. Thank you.

# Contents

# List of Figures

# List of Tables

*Dedicated to my father who would have been proud.*

# Chapter 1

# Introduction

In 1948, the world's first modern electronic computer ran its first program [28]. Today, after more than 70 years of staggering advancements, computer systems are ubiquitous, being used in business, academia, government, and at home; e.g., in 2017, more than two-thirds of the global population were connected by mobile devices [78]. During this time, the development of powerful microprocessors combined with the invention of high-speed computer networks enabled *distributed systems*. A distributed system is a group of independent computers that can communicate with each other [12] and, as a whole, it "appears to its users as a single coherent system" [151]. Therefore, in order to provide a consistent view to the users, the independent components of a distributed system require some form of *collaboration*.

A well-know example of a distributed system is the Internet, a global system of interconnected computer networks, with the Web being one of its main application. The Web is an extremely large distributed information system (i.e., as of February 2019, the indexable Web contains at least 5.16 billion pages[1]), that services hundreds of millions of users simultaneously. This level of performance can mainly be achieved due to the successful *partitioning* of web pages, i.e., each web site is managing its own part of the entire data set [155]. In addition, many web sites employ *replication* techniques to improve accessibility [149], i.e., multiple copies of a web site's data are placed at well-chosen locations. Replicating a web site has multiple benefits: it usually reduces the latency of accessing web pages, as the user requests are distributed across multiple replicas and redirected to replicas located close to the users; moreover, it improves the availability of the web site, as the failure of one replica does not result in the site's pages being inaccessible. A main difficulty with replicating a web site though, is managing *consistency* among replicas [155]. Intuitively, multiple copies of a web site's data are consistent when they are always identical. Thus, having multiple copies means that each page update must be propagated to all replicas. This propagation may take a while, especially in a large network such as the Internet. A way to avoid inconsistencies, even when multiple updates are propagated concurrently, is for this distributed system to execute a *distributed agreement* algorithm.

In this dissertation, we focus on distributed systems that have the following three main properties:

- We consider distributed systems that adopt replication techniques to improve the quality of provided services. Although the nature of these systems may vary—with examples including but not being limited to global-scale web-services [40, 76, 152], distributed ledgers [10], multiplayer video games [17, 18], and distributed operating systems [146, 114]—they all can be usually modeled as a distributed network of servers, each maintaining a copy of a *shared state*. Note that we distinguish between servers and clients: A *server* can both query and update its state, while a *client* can only submit query and update requests to one or more servers. Moreover, we consider applications for which the state updates cannot be reduced.

[1] https://www.worldwidewebsize.com/

- We consider distributed systems that provide *strong consistency* guarantees, i.e., all updates are seen in the same order at all servers[2]. Some distributed systems adopt weaker consistency models in order to improve the overall performance; for example, Amazon's Dynamo [46] implements eventual consistency [157], which entails that, in the absence of further updates, servers will eventually converge to the same state. Therefore, the burden of consistency management is shifted to the application layer. We do not consider such systems here.

- We consider distributed systems that are fault-tolerant. The ability to tolerate failures is essential. This is especially true for systems that handle critical data or are scaled out across a large number of servers. For the former, failures may lead to inconsistencies, which are unacceptable; for the latter, failures become so common that the lack of fault tolerance becomes a performance issue. For example, given a constant mean time between failures per server of around 2 years [67], a system with 1,000 servers would sustain a single-server failure more than once per day.

A well-known approach of implementing fault-tolerant distributed systems while guaranteeing strong consistency is *state-machine replication* [144, 90]. In state-machine replication, the shared state is modelled as a deterministic (potentially infinite) state machine, replicated across the participating servers and updated by executing well-defined operations (i.e., client requests). State-machine replication implements *active replication* [41]: It orders and propagates the operations to all servers, which will then execute them sequentially. This means that all copies of the state machine start from the same initial state, transition through the same succession of states, and therefore, produce the same sequence of outputs.

## 1.1 Use cases for state-machine replication

State-machine replication is a powerful building block for implementing general-purpose distributed services. An essential property of state-machine replication is that it provides strong consistency among replicas, which implicitly enables a distributed service to hide failures and appear to its users as a coherent and centralized service, while retaining advantages from being distributed. As a consequence, state-machine replication is most commonly employed as a means to achieve high availability; actually, it was invented mainly for replicating traditional applications, such as databases, in order to provide fault tolerance. When replicating for fault tolerance, it is not necessary to scale out across a large number of servers. Most state-machine replication deployments have only a handful of replicas [130]. For instance, common deployments with five servers tolerate up to two failures; even more, they allow for one server to be temporarily taken off-line for reconfiguration, while still tolerating one potential failure.

In practice, such state-machine replication deployments are typically used to provide coordination services to large distributed systems. Although the nature of these coordination services varies, ranging from configuration management to locking [76, 27], the pattern for using the replicated state machines is usually the same: The servers of a distributed system coordinate their activities by sending requests to a replicated state machine; once a request is executed, a reply is sent back to the server that sent that request. In general, the data stored by such a coordination system is sufficiently small that each replica will fit on a single server. However, this may not always be the case. Some large databases, such as Google's Spanner [40], store too much data to fit on a single server and therefore, must be partitioned. For availability, each partition is implemented as a replicated state machine; for instance,

[2] Strong consistency does not entail that all servers have identical states; due to the propagation time of updates, some servers may get updates faster then others.

Spanner uses the well-known Paxos [92] algorithm (see Section 3.1 for details) to implement state-machine replication. For consistency, requests that span multiple partitions use a two-phase commit protocol.

At a first glance, scaling out state-machine replication across more than a handful of servers seems unnecessary. However, besides providing high availability, having multiple consistent replicas may be a requirement of the application. For some applications, the number of replicas can easily be in the range of hundreds [10, 17]. For example, in a multiplayer video game, the players share a common state, which they periodically update. Similarly to the above pattern of using state-machine replication as a coordination service, the players could reliably order their updates through a state machine, which, for fault tolerance, is replicated across only a handful of servers. However, for the players to have a consistent view, each update must be propagated to all players, which will then apply them sequentially to their states.

Another well-known example of applications that require large-scale state-machine replication are distributed ledgers [10]. In these systems, a ledger for recording transactions is distributed across multiple servers, with each server maintaining a copy. The servers receive transactions as input, and their goal is to regularly agree on a common sequence of transactions to be added to the ledger. Scaling out a distributed ledger may have multiple benefits. For instance, a distributed ledger servicing clients world-wide may require to scale out across multiple geographically distributed servers in order to reduce the latency of local queries[3]; moreover, scaling out the ledger enables multiple companies to share it, while each company being able to independently validate every new transaction and to privately query their own copies.

## 1.2 Research contributions

To order and propagate the requests to all servers, state-machine replication requires the execution of a distributed agreement algorithm, such as consensus and atomic broadcast: The servers must agree on both the validity and the ordering of requests. However, the communication and synchronization cost of agreement entails that state-machine replication adds a considerable overhead to the system's performance, i.e., typical state-machine replication request rates are lower than the request rates of non-replicated systems.

For distributed services that require both high availability and strong consistency at very high request rates, e.g., airline reservation systems [152], high-performance state-machine replication is essential. This leads us to the first research question of this dissertation:

- **Research Question.** How can we devise and develop efficient solutions that push the limits of high-performance state-machine replication?

To address this question, we propose DARE, a novel high-performance *leader-based* state-machine replication algorithm. In leader-based algorithms, the order of requests is established by one of the servers after being elected leader; the leader holds this responsibility until it is suspected of having failed. In most existing algorithms that enable state-machine replication [92, 105, 84, 130], servers communicate with each other through message-passing, often implemented over UDP or TCP channels. DARE replaces the message-passing mechanism with RDMA, which enables servers to remotely access memory in the user-space of other servers. Every remote access is fully performed by the hardware without involving the operating system at either the origin or target of the access, resulting thus

[3] A local query entails reading the state of a single (preferably nearby) server and thus, it can return stale data. For strongly consistent reads, queries need to be serialized through state-machine replication. Therefore, local queries avoid the overhead of serialization at the cost of (possibly) outdated data.

in high-throughput, low-latency communication. Moreover, DARE is designed entirely for RDMA—it uses RDMA for ordering and executing both read and write requests, for detecting failures and for leader election. At the time of its publication, DARE was, to our knowledge, the only state-machine replication algorithm designed entirely with one-sided RDMA semantics and therefore, able to exploit the full potential of RDMA-capable networks, such as InfiniBand [77]. To evaluate DARE, we provide an open-source reference implementation over high-performance InfiniBand Verbs, which we used to build a strongly-consistent key-value store. Our evaluation shows that DARE improves the latency compared to existing state-machine replication algorithms by up to 35 times. Thus, by using remote direct memory access (RDMA) techniques in atypical ways, DARE pushes the limits of high-performance state-machine replication implementations by more than an order of magnitude.

Other applications, such as distributed ledgers [10], require large-scale state-machine replication deployments. However, the overhead of agreement also entails that the performance of state-machine replication is not scaling well with the number of servers[4] [155].

This leads us to the second research question of this dissertation:

- **Research Question.** How can we scale out state-machine replication across hundreds of servers while achieving high performance?

To address this question, we propose AllConcur, a novel *leaderless* concurrent atomic broadcast algorithm, that enables large-scale high-performance state-machine replication. Atomic broadcast is a communication primitive that ensures that all servers receive the same ordered sequence of messages. Most practical atomic broadcast algorithms were designed mainly to enable highly available distributed services [40, 27, 76] and therefore, we argue that they are not well suited for large-scale state-machine replication. For providing total order, these algorithm's work per broadcast message[5] is linear in the number of servers. Ordering is provided either through leader-based approaches (e.g., [92, 84, 27, 130]), which entail heavy workloads on the leader, since it must disseminate every message to all other servers, or by tagging messages with timestamps that reflect causal ordering [91] (e.g. [87, 68]) and therefore, must contain information on every server.

AllConcur adopts a leaderless approach, distributing the workload evenly among all servers and adds no overhead to messages (as no timestamps, reflecting causal ordering, are required). In AllConcur, servers exchange messages concurrently through an overlay network that is: (1) regular and thus, the workload is evenly balanced among servers; (2) sparse and thus, the work per broadcast message is sublinear; and (3) resilient and thus, the messages are reliably disseminated (similar to the diffusion of updates [108]). The overlay network's resiliency is given by its vertex-connectivity and can be adapted to system-specific requirements. As a result, AllConcur can strike a balance between reliability and performance—a higher vertex-connectivity entails higher fault tolerance, but also a higher network's degree, which consequently increases the work per broadcast message. Moreover, AllConcur adopts a novel early termination mechanism that reduces the expected number of communication steps significantly. To evaluate AllConcur, we provide open-source reference implementations over both standard sockets-based TCP and high-performance InfiniBand Verbs. Our evaluation shows that AllConcur can handle up to 135 million (8-byte) requests per second and achieves 17× higher throughput than Libpaxos [147], an implementation of Paxos [92].

The resiliency of AllConcur's overlay network comes at the cost of redundancy: Every server sends each message to all its successors in the overlay network and consequently, it receives each message from every predecessor. As a result, AllConcur introduces $d$ times more

[4] Note that we are focusing here on the scalability with the number of servers, not clients. Most existing state-machine replication algorithms actually scale well with clients, easily supporting on the order of thousands [160].

[5] In the absence of encryption, we assume that the main work performed by a server is sending and receiving messages.

messages into the network (with $d$ the network's degree), which imposes a lower bound on the work per broadcast message. To lift this bound, we propose AllConcur+, a novel leaderless concurrent atomic broadcast algorithm that extends AllConcur with the aim of improving performance. AllConcur+'s design is based on the following assertion: Although the frequency of failures in distributed systems makes many non-fault-tolerant services unfeasible [67, 70, 52], intervals with no failures are common enough that the constant overhead of redundancy leads to suboptimal solutions. Therefore, during intervals with no failures, AllConcur+ achieves minimal work by using a redundancy-free overlay network, such as one of the dissemination schemes typical for unrooted collectives [75]. When failures do occur, it automatically recovers by switching to the resilient overlay network of AllConcur. Our evaluation of AllConcur+'s performance is based on OMNeT++, a discrete-event simulator [156]. When no failures occur, AllConcur+ achieves comparable throughput to non-fault-tolerant algorithms and significantly outperforms AllConcur, LCR [68] and Libpaxos [147] both in terms of throughput and latency. Furthermore, our evaluation of failure scenarios shows that AllConcur+'s expected performance is robust with regard to occasional failures.

In summary, this dissertation's key contributions are the design, implementation, and evaluation of three novel algorithms that enable high-performance state-machine replication:

- **DARE**, a *direct access* state-machine replication algorithm that uses RDMA features in atypical ways to ensure high performance and reliability.
- **AllConcur**, a *leaderless* concurrent atomic broadcast algorithm that enables the implementation of large-scale high-performance state-machine replication by distributing the workload evenly among all servers.
- **AllConcur+**, a *leaderless* concurrent atomic broadcast algorithm that leverages redundancy-free state-machine replication during intervals with no failures to further improve performance.

In addition, for all three algorithms, informal proofs of correctness are provided, showing that both safety and liveness are guaranteed. Moreover, for AllConcur, both a formal specification and a mechanically verifiable proof of safety are provided.

**Structure of the dissertation.** The remainder of this dissertation is structured as follows. We state our assumptions about the system model and provide an overview of state-machine replication in Chapter 2. Then, we discuss the most related state of the art algorithms and implementations in Chapter 3. Afterwards, we present the three algorithms in Chapters 4–6: DARE in Chapter 4, AllConcur in Chapter 5, and AllConcur+ in Chapter 6. Finally, we conclude the dissertation in Chapter 7.

# Chapter 2

# System Model and Overview

In this chapter, we present the assumptions made about the system model and provide an overview of state-machine replication by describing several concepts that are repeatedly referred to throughout this dissertation.

## 2.1 System Model

We consider a distributed system composed of a group of $n$ interconnected servers, i.e., $p_0, \ldots, p_{n-1}$. When modeling a distributed system, it is common to address the following questions:

- What type of failures are tolerated by the system?
- What is the synchronism of the communication network and the servers?
- How are the participating servers communicating with each other?

Throughout this dissertation, unless stated otherwise, we make the following assumptions about the system model:

**Failure model.** The failures a distributed system can tolerate range from *Byzantine* to *fail-stop*. Byzantine failures [99], also known as arbitrary failures, entail that faulty servers may behave arbitrarily, even maliciously, while non-faulty servers follow the specification of the algorithm. Fail-stop failures entail that faulty servers operate correctly until they fail and once they fail, they can no longer influence the operation of other servers in the group. Although the fail-stop model cannot account for arbitrary failures, it can be reasonably assumed for many distributed systems. Moreover, algorithms tolerating fail-stop failures are more efficient than their Byzantine fault-tolerant counterparts. In the algorithms we present in this dissertation, we assume that the $n$ server are subject to a maximum of $f$ fail-stop failures, i.e., no Byzantine behavior is tolerated.

**Synchrony model.** In the context of synchrony models, there are two well-known types of distributed systems—*synchronous* and *asynchronous*. In synchronous systems, there are fixed upper bounds on both the communication network delays and the time intervals between consecutive steps of a server; moreover, these bounds are known a priori and algorithms can depend on them. This synchronous model involves a reduced amount of uncertainty, which makes it convenient for designing distributed algorithms. However, the assumption of a synchronous model is generally not practical in real-world distributed systems. For instance, it is difficult to bound network delays in the Internet; even when such a bound exists, it would be too large to justify designing an algorithm that depends on it. In contrast, asynchronous systems make no timing assumptions, which makes them more practical. Yet, according to the impossibility result of Fischer, Lynch, and Paterson [59], in asynchronous systems,

there are no consensus (or atomic broadcast) algorithms that can tolerate even one failure. To overcome this, it is possible to use one of the partial synchrony models [56] (e.g., both bounds exists, but they are not known a priori). As an alternative, for all our algorithms, we assume an asynchronous model augmented with a failure detector [34], i.e., a distributed oracle that provides (possibly incorrect) information about faulty servers. In Section 2.2.4, we provide a short overview of failure detectors.

**Communication model.** The servers communicate either by sending messages or by directly accessing remote memory. We discuss the details in the following two subsections, i.e., Sections 2.1.1 and 2.1.2.

### 2.1.1 Message-passing

One of the most common communication models in distributed systems is message-passing—servers communicate by exchanging messages. For instance, the well-known TCP and UDP protocols can be both used to implement a communication service through messages. We use the message-passing model in both AllConcur and AllConcur+. We assume the servers are physically connected through an interconnect that allows any two servers to send messages to each other, i.e., there is a sequence of links, switches and routers connecting any two servers. Yet, since both AllConcur and AllConcur+ are meant for large-scale, the traditional all-to-all communication pattern adopted by many existing algorithms [92, 105, 84, 130] is not suitable. Therefore, we model the actual communication by an overlay network described by a digraph $G$—a server $p_i$ can send messages directly to another server $p_j$ if and only if $G$ has an edge $(p_i, p_j)$. The edges of $G$ describe FIFO reliable channels, i.e., we assume the following properties [131]:

- (no creation) if $p_j$ receives a message $m$ from $p_i$, then $p_i$ sent $m$ to $p_j$;
- (no duplication) $p_j$ receives every message at most once;
- (no loss) if $p_i$ sends $m$ to $p_j$, and $p_j$ is non-faulty, then $p_j$ eventually receives $m$;
- (FIFO order) if $p_i$ sends $m$ to $p_j$ before sending $m'$ to $p_j$, then $p_j$ will not receive $m'$ before receiving $m$.

In practice, FIFO reliable channels are easy to implement, using sequence numbers and retransmissions (e.g., the TCP protocol).

Formally, a digraph $G$ can be defined as a set of $n$ vertices $\mathcal{V}(G) = \{v_i : 0 \leq i \leq n-1\}$ and a set of directed edges $\mathcal{E}(G) \subseteq \{(u,v) : u,v \in \mathcal{V}(G) \text{ and } u \neq v\}$. Throughout this dissertation, we use the terms vertex and server interchangeably. In the context of fault-tolerant distributed systems, we are interested in the following four parameters of a digraph $G$: (1) degree $d(G)$; (2) diameter $D(G)$; (3) vertex-connectivity $\kappa(G)$; and (4) fault diameter $D_f(G,f)$. Table 2.1 summarizes all the digraph notations used throughout this dissertation.

**Degree.** A digraph's degree relates to the concepts of both successor and predecessor of a vertex—given an edge $(u,v) \in \mathcal{E}(G)$, $v$ is a *successor* of $u$, while $u$ is a *predecessor* of $v$. For a vertex $v$, the set of all successors is denoted by $v^+(G)$, while the set of all predecessors by $v^-(G)$. The out-degree of a vertex is the number of its successors, i.e., $|v^+(G)|$, while the in-degree is the number of its predecessors, i.e., $|v^-(G)|$. $G$'s *degree*, denoted by $d(G)$, is the maximum in- or out-degree over all vertices; moreover, $G$ is *d-regular* (or just regular) if $d(G) = |v^+(G)| = |v^-(G)| = d, \forall v \in \mathcal{V}(G)$.

**Diameter.** A path from vertex $u$ to vertex $v$ is a sequence of vertices $\pi_{u,v} = (v_{x_1}, \ldots, v_{x_d})$ that satisfies four conditions: (1) $v_{x_1} = u$; (2) $v_{x_d} = v$; (3) $v_{x_i} \neq v_{x_{j \neq i}}, \forall i, j$; and (4) $(v_{x_i}, v_{x_{i+1}}) \in E, \forall 1 \leq i < d$. The length of a path, denoted by $|\pi_{u,v}|$ is defined by the number of contained

| Notation | Description | Notation | Description |
|---|---|---|---|
| $G$ | the digraph | $d(G)$ | degree |
| $\mathscr{V}(G)$ | vertices | $D(G)$ | diameter |
| $\mathscr{E}(G)$ | directed edges | $\pi_{u,v}$ | path from $u$ to $v$ |
| $v^+(G)$ | successors of $v$ | $\kappa(G)$ | vertex-connectivity |
| $v^-(G)$ | predecessors of $v$ | $D_f(G,f)$ | fault diameter |

TABLE 2.1: Digraph notations.

edges. $G$'s *diameter*, denoted by $D(G)$, is the length of the longest shortest path between any two vertices.

**Connectivity.** $G$ is connected if $\exists \pi_{u,v}, \forall u \neq v \in \mathscr{V}(G)$. *Vertex-connectivity*, denoted by $\kappa(G)$, is the minimum number of vertices whose removal results in a disconnected or a single-vertex digraph. An alternative formulation is based on the notion of disjoint paths: two paths are *vertex-disjoint* if they contain no common internal vertices. Thus, according to Menger's theorem, the vertex-connectivity equals the minimum number of vertex-disjoint paths between any two vertices. Vertex-connectivity is bounded by the degree, i.e., $\kappa(G) \leq d(G)$; digraphs with $\kappa(G) = d(G)$ are said to be *optimally connected* [115, 48].

**Fault diameter.** Let $F \subset \mathscr{V}(G)$ be a set of $f < \kappa(G)$ vertices that are removed from $G$ resulting in a digraph $G_F$ with $V(G_F) = \mathscr{V}(G) \setminus F$ and $E(G_F) = \{(u,v) \in \mathscr{E}(G) : u,v \in V(G_F)\}$. For any subset $F$, the resulting digraph $G_F$ is connected; yet, the diameter of $G_F$ may be larger than $D(G)$. $G$'s *fault diameter*, denoted by $D_f(G,f)$, is the maximum diameter of $G_F, \forall F \subset \mathscr{V}(G)$.

### 2.1.2 One-sided communication

**Overview of RDMA.** Remote direct memory access (RDMA) is an interface that enables zero-copy communication between servers—it transfers data directly between the user-space of two servers, without kernel interference or memory copying. To enable this mechanism, RDMA uses so called queue pairs (QPs), which are logical communication endpoints, with a context in the network interface controller (NIC). Every QP consists of a send queue and a receive queue. Servers communicate by posting operations to these queues using the *verbs* API.

RDMA supports two types of verbs—*messaging verbs* and *memory verbs*. Messaging verbs entail two-sided communication (i.e., message-passing), which is similar to standard sockets-based TCP; yet, unlike TCP, two-sided RDMA operations are performed without any interaction with the operating system at either end of the communication. Messaging verbs include send and receive verbs. Memory verbs entail one-sided communication: Servers access memory in the user-space of other servers by bypassing the remote CPUs[1]. Therefore, this mechanism is fundamentally different from existing message-passing mechanisms. For instance, RDMA one-sided operations change the system's failure-characteristics: a CPU can fail but its memory is still remotely accessible. Memory verbs include both read and write RDMA operations (and atomic operations).

RDMA supports both reliable and unreliable transports. Reliable transports guarantee that a server's messages are delivered in-ordered by the destination's NIC; unreliable transports provide no such guarantee. Moreover, the transports can be either connected or unconnected (i.e., datagram). Connected transports require a one-to-one mapping between QPs,

[1]Note that before memory can be accessed, a direct mapping from logical addresses in user-space to physical memory addresses is established in the NIC; this memory exposure is performed through the operating system to guarantee protection. However, the remote access is then fully performed by the hardware.

i.e., for each remote destination, a server needs to create a QP and connect it to one of the QPs of that destination. In contrast, a datagram QP can be used to communicate with multiple destinations. Current RDMA networks support the following transports: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). All three support both send and receive verbs. However, UD supports neither read nor write operations, while UC supports only read operations. Moreover, UD support multicast (i.e., one-to-many) transfers.

**Communication model in DARE.** We use RDMA in DARE: We assume the servers are connected through an interconnect with support for RDMA, such as InfiniBand or RoCE (RDMA over Converged Ethernet). Moreover, we assume an one-sided reliable communication model provided by the RC transport—to communicate with each other, servers reliably access remote memory by invoking read or write RDMA operations. Since DARE is not intended for large-scales, we consider the traditional all-to-all communication pattern, which entails that every server can communicate directly with every other server. In addition, we use UD to simplify non-performance-critical parts of DARE, such as setup and interaction with clients.

## 2.2 Overview of state-machine replication

State-machine replication enables fault-tolerant distributed systems while guaranteeing strong consistency [144, 90]. The idea behind state-machine replication is to model a distributed network of servers as a collection of (potentially infinite) state machines that transition deterministically between states by executing well-defined operations. Consequently, every server implements a state machine, such as a key-value store. For consistency, all the state machines start from the same initial state and, *by executing the same sequence of operations*, they transition through the same succession of states. Therefore, state-machine replication implements active replication [41]: first, a *distributed agreement* algorithm orders and propagates the operations to all servers; second, every server executes the operations sequentially.

It is common to implement state-machine replication using a *replicated log* [130]: Server store the operations into local buffers (i.e., the logs) and then, they execute each operation in order. As long as the logs contain the same sequence of operations, the state machines remain consistent. Thus, the servers must agree on each operation's position in the log; consequently, they must use a distributed agreement algorithm, such as *atomic broadcast* or *consensus*. Note that consensus and atomic broadcast are equivalent [34].

### 2.2.1 Atomic broadcast

Informally, we can define (non-uniform) atomic broadcast as a communication primitive that ensures all messages are delivered[2] in the same order by all non-faulty servers. To formally define it, we use the notations from Chandra and Toueg [34]: $m$ is a message (uniquely identified); $A\text{-}broadcast(m)$ and $A\text{-}deliver(m)$ are communication primitives for broadcasting and delivering messages atomically; and $sender(m)$ is the server that A-broadcasts $m$. Any *non-uniform* atomic broadcast algorithm must satisfy the following four properties [34, 69, 47]:

- (Validity) If a non-faulty server A-broadcasts $m$, then it eventually A-delivers $m$.
- (Agreement) If a non-faulty server A-delivers $m$, then all non-faulty servers eventually A-deliver $m$.

[2]We distinguish between receiving and delivering a message [19]. Servers can delay the delivery of received messages.

- (Integrity) For any message $m$, every non-faulty server A-delivers $m$ at most once, and only if $m$ was previously A-broadcast by $sender(m)$.
- (Total order) If two non-faulty servers $p_i$ and $p_j$ A-deliver messages $m_1$ and $m_2$, then $p_i$ A-delivers $m_1$ before $m_2$, if and only if $p_j$ A-delivers $m_1$ before $m_2$.

Integrity and total order are safety properties—they must hold at any point during execution. Validity and agreement are liveness property—they must eventually hold (to ensure progress).

**Uniformity.** The agreement and the total order properties above are non-uniform—they apply only to non-faulty servers. Uniform properties are stronger guarantees, i.e., they apply to all server, including faulty ones [47]:

- (Uniform agreement) If a server A-delivers $m$, then all non-faulty servers eventually A-deliver $m$.
- (Uniform total order) If two servers $p_i$ and $p_j$ A-deliver messages $m_1$ and $m_2$, then $p_i$ A-delivers $m_1$ before $m_2$, if and only if $p_j$ A-delivers $m_1$ before $m_2$.

### 2.2.2 Reliable broadcast

When only validity, agreement and integrity hold, the broadcast is *reliable*; we use *R-broadcast*$(m)$ and *R-deliver*$(m)$ to denote the communication primitives of reliable broadcast. Informally, (non-uniform) reliable broadcast ensures that all messages are reliably received by all (non-faulty) servers. A straightforward reliable broadcast algorithm uses a complete digraph for message dissemination [34]. When a server executes *R-broadcast*$(m)$, it sends $m$ to all other servers; when a server receives $m$ for the first time, it executes *R-deliver*$(m)$ only after sending $m$ to all other servers. Clearly, this algorithm solves the reliable broadcast problem. Yet, the all-to-all communication pattern is unnecessary: In order to tolerate up to $f$ faulty servers, it is sufficient for the reliable broadcast to use a digraph with vertex-connectivity larger than $f$ (i.e., despite any $f$ failures, the servers remain connected).

### 2.2.3 Consensus

In the consensus problem, each server has an input and an initially unset output. The servers *propose* their inputs and then, they irreversibly *decide* upon a single value for the outputs. Any *non-uniform* consensus algorithm must satisfy the following three properties [12]:

- (Termination) Any non-faulty server must decide.
- (Agreement) Any two non-faulty servers must decide on the same value.
- (Validity) If all the inputs are the same, then the value decided upon must be the value of the common input.

Agreement and validity are safety properties, while termination is a liveness property.

**Uniformity.** The agreement property above is non-uniform, i.e., it applies only to non-faulty servers. For uniform consensus, the stronger guarantee of *uniform* agreement needs to apply to all servers, including faulty ones:

- (Uniform agreement) If a server decides, then all non-faulty servers eventually decide the same.

### 2.2.4 Failure detectors

Failure detectors (FD) have two properties [34]:

- (Accuracy) No server is suspected to have failed before actually failing.
- (Completeness) Eventually, every faulty server is suspected to have failed by every non-faulty server.

Accuracy is a safety property, while completeness is a liveness property. If both properties hold, then the FD is *perfect* (denoted by $\mathscr{P}$) [34]. Since in asynchronous systems, failed and slow servers cannot be distinguished [59], guaranteeing accuracy requires some assumptions of synchrony [56]. Although such assumptions can be practical, especially for deployments within a single datacenter, to widen applicability (e.g., deployments over multiple datacenters), the FD can be *eventually perfect* (denoted by $\Diamond\mathscr{P}$) [34]. $\Diamond\mathscr{P}$ guarantees accuracy only eventually, i.e.,

- (Eventual accuracy) Eventually, no server is suspected to have failed without actually failing.

In theory, eventual accuracy entails that there is a time after which all non-faulty servers always trust each other. In practice though, it is sufficient for all non-faulty servers to trust each other "sufficiently long" [2] so that the system can make progress.

# Chapter 3

# Related Work

The importance of practical resilient distributed services sparked numerous proposals of algorithms and implementations of state-machine replication. In this chapter, we discuss the state of the art approaches that are most related to the algorithms presented in this dissertation. In particular, we focus on related approaches that assume a fail-stop failure model, i.e., we do not consider Byzantine fault-tolerant algorithms, such as PBFT [31] and HoneyBadgerBFT [116]. All three algorithms developed within this dissertation—DARE, AllConcur, and AllConcur+—enable high-performance state-machine replication, which is a well-known method of implementing fault-tolerant distributed systems that guarantee strong consistency [144, 90]. State-machine replication implements active replication [41] and therefore, it requires (for ordering and propagating client requests to all replicas) the execution of a distributed agreement algorithm, such as consensus or atomic broadcast. DARE is a high-performance state-machine replication algorithm designed entirely with RDMA semantics (see Chapter 4); both AllConcur and AllConcur+ are atomic broadcast algorithms that enable us to scale out state-machine replication across hundreds of servers, while achieving high performance (see Chapters 5 and 6).

The remainder of this chapter has the following structure: Section 3.1 introduces Paxos [92], Lamport's classic consensus algorithm; Section 3.2 describes Raft [130], a consensus algorithm that is the basis for DARE's design; Section 3.3 presents other approaches that use RDMA to improve the performance of distributed systems; and Section 3.4 gives a brief overview to existing atomic broadcast algorithms.

## 3.1 Paxos

Paxos [92, 93] is the most widely used consensus algorithm for implementing state-machine replication. In particular, Multi-Paxos, a practical deployment that combines multiple instances of Paxos, is often at the core of real-world distributed services, such as Google's Chubby lock service [27], Megastore [13] and Spanner [40] storage systems, and Microsoft's Windows Azure Storage [29] and Autopilot cluster management service [79].

Paxos is a term for describing a family of algorithms for solving consensus in an asynchronous system, where servers may fail by crashing. Paxos was first introduced by Lamport in its seminal paper [92], which was first published in 1989. Note that around the same time, Oki and Liskov independently proposed Viewstamped Replication [129, 105], a consensus algorithm with many similarities to Paxos [100, 154]. Moreover, the original specification of Paxos [92, 93] was followed by several papers that further describe the algorithm or discuss how to implement it [100, 101, 23, 106, 113, 24, 88, 32, 153]. In general, Paxos splits the execution of consensus into three roles: proposers, acceptors, and learners. Proposers propose values, acceptors decide upon those values, and learners learn the decided values.

**Safety.** Paxos always guarantees safety, i.e., no two learners can learn different values, even when failures occur. To this end, Paxos relies on quorums, which are subsets of acceptors

with the property that any two subsets share at least one acceptor. The common way to implement quorums is to select any subset containing at least a majority of acceptors, i.e., if there are a total of $n_a$ acceptors, a quorum consist of at least $\lceil (n_a + 1)/2 \rceil$ acceptors. In a nutshell, the reason for always guaranteeing safety is that every decided value requires a quorum to agree upon it; therefore, for any two successive decisions, there is at least one acceptor that agreed with both decided values.

**Two-phase execution.** In more details, the Paxos algorithm operates in two phases [93]. In Phase 1, a proposer sends to a quorum (of acceptors) a *prepare* message that contains a proposal number $s$ that is greater than any other proposal number it used before. Acceptors wait for prepare messages from any proposer. When an acceptor receives a prepare message, it checks its proposal number $s$. If $s$ is not greater than any proposal numbers it already received, it ignores the prepare message. Otherwise, it sends back to the proposer a *promise* message, promising not to accept any future messages with proposal numbers lower than $s$. In addition, the promise message contains both the proposal number and the corresponding value of the latest proposal accepted (if any) by the acceptor (see accepted messages below).

In Phase 2, if the proposer receives promise messages (for proposal number $s$) from a quorum, then it sends to each of the acceptors (in the quorum) an *accept* message that contains $s$ and a value $v$. Intuitively, the proposer makes a request, to the acceptors, to accept value $v$ associated with proposal number $s$. The choice of $v$ depends on the received promise messages: If any of the responding acceptors have already accepted a value, then $v$ is set to the value with the highest proposal number among all received promise messages; otherwise, the proposer sets $v$ to a value of its choice. An acceptor that receives an accept message with a proposal number $s$ accepts it if and only if it has not already promised (in Phase 1) to accept only messages with proposal numbers greater than $s$. If it accepts the proposal, the acceptor sends an *accepted* message to every learner. Otherwise, it ignores the accept message.

As an optimization, instead of just ignoring prepare or accept messages, acceptors can reply with negative acknowledgements [93]. As a result, a proposer would get informed that it should abandon its proposal and try again with a higher proposal number. Moreover, in order to be able to recover after failures (in the case of a crash-recovery failure model), acceptors must save their responses to stable storage before sending them [93].

**Liveness.** Having multiple proposers may hinder the progress of the algorithm by sending prepare messages with increasing proposal numbers [93]. For instance, a proposer $P_1$ successfully completes Phase 1 for proposal number $s_1$. Then, another proposer $P_2$ also completes Phase 1 for $s_2 > s_1$, which entails $P_1$ cannot complete Phase 2. Consequently, $P_1$ completes Phase 1 for another proposal number $s_3 > s_2$, which entails $P_2$ cannot complete Phase 2. Potentially, this scenario can go on forever and thus, not allowing the algorithm to make progress. Therefore, in order to guarantee progress (i.e., for liveness to hold), Paxos requires a distinguished proposer (also known as a leader or a coordinator). The leader is elected by the participating servers. As long as the elected leader is non-faulty and can communicate with a quorum of non-faulty acceptors, it will eventually manage to propose a value that is accepted.

Besides guaranteeing progress, electing a leader enables a trade-off between the delay of learning a decided value (i.e., how soon do learners receive accepted messages) and the amount of accepted messages introduced into the network [93]. On the one hand, acceptors can send every accepted message to every learner. Therefore, the delay is minimized. Yet, given $n_a$ acceptors and $n_l$ learners, the number of accepted messages introduced into the network (per decided value) is $n_a \times n_l$. On the other hand, acceptors can send every accepted message to the leader, which sends it further to every learner. Therefore, the number of

accepted messages (per decided value) is reduced to $n_a + n_l$, but at the cost of one extra communication step.

**State-machine replication.** Paxos enables the implementation of state-machine replication—a sequence of Paxos instances are used to reach consensus on the entries in a replicated log [93]. To increase performance in practical deployments (i.e. reduce the message complexity), every participating server typically plays all the roles at the same time—proposer, acceptor and learner. Moreover, although the original description of Paxos [92, 93] does not specify the details of how a leader is elected, most implementations rely on some timing assumptions[1] (e.g., [106, 88, 40]), such as the existence of $\diamond\mathscr{P}$ [34]. It is essential to notice that safety is always guaranteed, even if multiple proposers consider themselves leaders simultaneously. Safety ensures that the replicas remain consistent, a property necessary for the correct implementation of state-machine replication. In Chapter 4, we show that similar to Paxos, DARE also guarantees safety invariably. Moreover, in Chapters 5 and 6, we show that, under the assumption of $\diamond\mathscr{P}$, both AllConcur and AllConcur+ always guarantee safety, due to the so called forward-backward mechanism (see Section 5.2.1 for details).

A common optimization when using Paxos to implement state-machine replication is for the leader to execute a sequence of Phase 1 instances in advance [93]. As a result, the cost of reaching agreement on a request is given by the cost of only executing Phase 2.

**Paxos variants.** Several variants were proposed with the aim of making the basic Paxos algorithm more efficient:

- Disk Paxos [64] is a disk-based version of the Paxos algorithm that replaces acceptors with disks and as a result, it enables a fine-grained failure model, where progress is guaranteed even if all processors but one have failed (i.e., progress requires at least a majority of disks to be available).

- Cheap Paxos [98] is a variation of the Paxos algorithm that uses a reduced set of acceptors. In order to tolerate $f$ failures, the leader sends prepare and accept messages only to a fixed quorum of $f+1$ acceptors, while a set of $f$ cheap auxiliary acceptors are kept idle and used only when failures occur.

- Fast Paxos [96] is a variant of Paxos that reduces the number of communication steps needed for a learner to learn a decided value. Instead of the three communication steps required by Paxos (i.e., the client sends the request to the leader, the leader sends accept messages to the acceptor, and the acceptors send accepted messages to the learners), Fast Paxos requires only two, by allowing the clients to send accept messages to the acceptors. Yet, this comes at the cost of requiring $3f+1$ acceptors instead of $2f+1$ for safety.

- Another variant of Paxos is to allow the out-of-order processing of non-interfering requests [95, 119]. The intuition behind this optimization is that if two requests are non-interfering, then the order in which they are executed does not affect the final outcome. Relaxing the ordering requirement leads to an increase in performance. However, for this, the consensus algorithm must be able to identify conflicting requests (i.e., servers need to attach ordering constraints to each request), which adds significant complexity to Paxos.

[1] According to the famous result of Fischer, Lynch, and Patterson [59], a reliable algorithm for leader election in an asynchronous system must rely either on randomness or on timing assumptions.

**Why not Paxos?** We design DARE as a leader-based algorithm that provides high availability (see Chapter 4). Therefore, it relies on some of the same principles as Paxos, as do other existing leader-based consensus algorithms [84, 105, 130]. For instance, it replicates requests on a quorum of servers in order to ensure safety, it uses a leader election algorithm to choose a distinguished proposer, and it guarantees liveness as long as a majority of servers are non-faulty and can communicate with each other. However, Paxos relies on message-passing for communication and therefore, cannot benefit from the high-throughput and the low-latency of RDMA-capable networks, such as InfiniBand. In Chapter 4, we describe how DARE replaces message-passing with one-sided RDMA primitives.

We design both AllConcur and AllConcur+ as leaderless atomic broadcast algorithms that enable state-machine replication to scale out to hundreds of servers while achieving high performance (see Chapters 5 and 6). Paxos also enables the implementation of state-machine replication. However, in order to make progress, Paxos requires a leader to be elected. Once elected, for agreeing on each request, the leader must communicate directly with every other server, i.e., in Paxos, the servers communicate through an overlay network described by a complete digraph. Therefore, in Paxos (and in other leader-based algorithms [84, 105, 130]), the work per agreed upon request is linear in the number of servers. This means that leader-based approaches are not suitable for large-scale deployments of state-machine replication. In Chapters 5 and 6, we show that both AllConcur and AllConcur+ require sublinear work per agreed upon request. The reason is twofold. First, both algorithms are leaderless and therefore, they distribute the workload evenly among all servers. Second, the server exchange messages through an overlay network described by a sparse digraph.

## 3.2 Raft

Although being widely used in practice [27, 13, 40, 29, 79], Paxos is difficult both to understand and to implement, as several papers have argued [88, 32, 130, 153]. This difficulty is emphasized even more by Lamport himself in his subsequent attempt to explain Paxos, where he claims that the original presentation [92] was probably "Greek to many readers" [93]. The main issue is that the specification of Paxos is rather theoretical and it omits some of the details necessary for a correct implementation [88, 32].

Raft is a leader-based consensus algorithm that shares many similarities to Paxos, Viewstamped Replication [129, 105], and Zab [2] [84], but it is designed with understandability as the primary goal [130]. To this end, Raft is described directly as a solution for implementing a replicated log. This is in contrast to Paxos, which first describes the consensus algorithm for deciding on a single value and then, describes how to combine multiple instances of this algorithm in order to implement the replicated log [93]. Moreover, Raft is described in sufficient detail to facilitate its correct implementation in practical systems. For instance, Raft's description provides mechanisms for both electing a leader and changing the set of participating servers.

**Leader-based approach.** In Raft, the set of participating servers communicate through message-passing[3]. Moreover, Raft adopts a typical leader-based approach—the servers elect a leader, which is responsible for managing the replicated log, until it is suspected to have failed. As a result, the execution of Raft is modeled as a sequence of asynchronous terms.

[2] Zab is a consensus algorithm that resembles Viewstamped Replication and is used in the Apache Zookeeper coordination service [76].

[3] Actually, servers in Raft communicate with remote procedure calls, which are typically implemented via message-passing: A server sends a message to a remote server, requesting it to execute a specified procedure; the remote server replies by sending back a response message.

Each term starts with one or more servers (i.e., candidates) trying to become leaders. For safety, Raft guarantees no more than one leader per term; some elections may result in no leader, which triggers the start of another election and consequently, of the subsequent term.

**Leader election.** Raft uses a heartbeat-based failure detector to identify faulty servers: Once a leader is elected, it starts sending heartbeat messages to the other servers; when a server receives no heartbeat messages for a given period of time, it starts an election by proposing itself as a candidate for the leadership of the subsequent term. When starting an election, a candidate sends vote requests to all remote servers. In order to become leader, the candidate must receive votes from at least a majority of servers (itself included). The vote requests serve two purposes: to neutralize the suspected leader and to elect a new leader. Neutralizing the suspected leader is similar to other leader-based consensus algorithms, such as Paxos, Viewstamped Replication, and Zab. When a remote server receives a vote request, it moves to the subsequent term and consequently, no longer accepts requests from the previous leader. However, unlike these other algorithms, Raft ensures that an elected leader has an up-to-date log, i.e., it contains all already agreed-upon entries. As a result, once elected, a leader does not need any additional mechanisms to catch up with the other servers and consequently, can start replicating the log.

**Log replication.** Raft's log replication mechanism results in a simple flow of data—clients send state machine operations to the leader, which first appends them to its log as new entries and then replicates these entries on the remote servers by sending append requests. As a consequence, each log entry contains a state machine operation; moreover, Raft ensures that each entry is uniquely defined by both the term, in which the leader received it, and the index, that identifies its position in the log. Once the leader decides that an entry is safely replicated (i.e., committed), it applies the enclosed operation to its state machine and sends the result back to the client. For a log entry to be committed, it needs to reside in the logs of a majority of servers. However, this condition is not always sufficient. For instance, a leader may fail after committing an entry, but before notifying the remote servers that the entry is committed; as a result, although the new leader has the entry (due to the requirement of its log to be up-to-date), it does not commit it (yet). In general, failures may lead to several inconsistencies in the logs. To fix these inconsistencies, a new leader adjusts the remote logs by replicating its entire log on the remote servers. This requires the leader to transfer to each remote server the entries necessary for the remote log to match its own. To facilitate this log adjustment, while guaranteeing safety, Raft maintains the following invariant, referred to as the log matching property: If two logs have an identical entry (i.e., with the same term and index), then all the preceding entries are identical as well. Therefore, a new leader must identify for each remote server the latest entry common for both logs.

**Dynamic membership.** Keeping the set of participating servers fixed is unrealistic in practice. For example, in order to maintain reliability, failed servers need to be replaced by allowing other servers to join; also, it may be necessary to (intentionally) remove slow servers. Therefore, practical distributed services must support dynamic membership. In order to guarantee safety, changing the set of participants (i.e., the configuration) entails a two-phase approach (e.g., [105]). In Raft, configurations are stored in special entries in the replicated log. When a server encounters such an entry, it updates its own configuration accordingly, regardless of whether the entry is committed. To change the configuration, Rafts uses a mechanism similar to the one used by Zab [84]: It first switches to a transitional configuration, in which both the old and the new set of servers are participating in the consensus algorithm; then, once this transitional configuration is committed, it switches to the new configuration. A key

property of this mechanism is that, during the transitional configuration, electing leaders and replicating log entries requires a majority of servers from both configurations (old and new). As a result, Raft can continue servicing clients even when the configuration changes.

**Why not Raft?** Raft is designed to increase the understandability and facilitate the implementation of state-machine replication. As a result, in the design of DARE, we adopt some of Raft's design choices (see Chapter 4). For instance, DARE also splits the execution of consensus into a sequence of terms; in every term, at most one leader is responsible for replicating the log; every elected leader is guaranteed to have an up-to-date log; and once elected, the leader adjusts the logs of the remote servers in order to match its own. Moreover, in order to change the set of participating servers without interrupting normal operation, DARE uses Raft's transitional configuration. However, as is the case with Paxos, Raft also relies on message-passing for communication. In Chapter 4, we describe how designing DARE entirely for RDMA improves the performance of state-machine replication by more than an order of magnitude. For example, DARE implements the replicated log as a circular buffer, which enables the leader to directly access the remote logs without involving the CPUs of the remote servers. This results in both faster log replication and higher reliability; moreover, the remote servers are available for other tasks, such as recovery. Also, DARE increases the efficiency of Raft's log adjustment mechanism: Adjusting a remote log through RDMA always requires only two remote accesses, while the number of messages needed by Raft is dependent on the number of non-matching entries (see Section 4.3.3.2 for details).

## 3.3 Accelerating distributed systems with RDMA

Since the concept of zero-copy transfers was first introduced [158], several projects have used RDMA in order to enhance the performance of various systems. The nature of these systems varies, ranging from high-performance computing systems, where the MPI standard [123] was successfully integrated with RDMA [74], to distributed storage systems [83, 82, 117, 86, 53, 54], where an in-memory key-value store is distributed across multiple servers. For instance, several attempts were made to improve the performance of Memcached [83, 82, 117], a well-known key-value store that was designed as a memory caching layer [60] and is widely used in industry (e.g., by Facebook for its back-end services [128]). In Memcached, as in other in-memory key-value stores, the data is distributed across the memory of multiple servers, which use memory-efficient data structures, such as hash tables, to provide fast access to their underlying data; typically, clients access this data by issuing get and put operations. Memcached provides an open-source implementation over the traditional BSD Sockets interface (i.e., message-passing).

In [83], the authors extend the Memcached implementation and make it RDMA capable. To this end, they use the Unified Communication Runtime [81], a communication library that exposes an active message [159] based API. Clients need to issue active messages for both get and put operations; these messages carry the control information necessary for activating a data transfer through RDMA. Therefore, RDMA is used as an optimization for message-passing. To further improve performance, the same authors propose a hybrid model that uses both the Reliable Connection (RC) and Unreliable Datagram (UD) transports [82]. Nonetheless, both approaches provide no support for one-sided RMDA reads or writes. In contrast, we design DARE entirely with one-sided RDMA semantics (see Chapter 4) and therefore, it enables the implementation of a key-value store with support for both one-sided RMDA reads or writes.

Pilaf [117] is a key-value store that supports one-sided RDMA reads—clients access a server's hash table directly through RDMA. To allow RDMA reads, every server exposes two

memory regions, one consisting of an array of fixed size hash table entries and one containing the actual keys and values. Therefore, for each get operation, a client probes a server's hash table using two RDMA reads—the first one to get a hash table entry and the second one to get the corresponding key-value string. Moreover, by using 64-bit CRCs to detect inconsistent reads (caused by read-write races), Pilaf provides linearizability. However, to simplify the design, it avoids write-write races between servers and clients: In Pilaf, writes are handled locally by the servers after receiving client requests via traditional messaging.

In contrast, the authors of HERD [86] argue that using one-sided RDMA reads in key-value stores is inefficient (since they entail multiple round trips). Instead, HERD clients use one-sided RDMA writes over an Unreliable Connection (UC) to write their requests directly to the request memory regions, which are exposed by every server. The servers are polling these memory regions for incoming requests. Once a server executes a request, it sends back the response through a message over UD. Neither Pilaf nor HERD employ any mechanism for replication—while HERD provides no fault tolerance, Pilaf provides reliability (but no availability) by logging to a disk. As we show in Chapter 4, DARE uses one-sided RMDA operations to improve the performance of data replication and therefore, enables high-performance key-value stores that provide both reliability and availability. Both Pilaf and HERD use one-sided RDMA to speed up client access to the key-value store; thus, they are complementary to DARE.

Similarly to Pilaf, FaRM [53, 54] supports one-sided RDMA reads, while providing strong consistency guarantees. Unlike Pilaf though, FaRM is a more general-purpose main memory distributed computing platform: It exposes the memory of all servers as a shared address space and provides support for distributed transactions with strict serializability. Moreover, to exploit locality and avoid unnecessary data transfers, FaRM adopts a symmetric model, in which each machine acts as both server and client. FaRM uses one-sided RDMA reads to access remote data. Also, it uses RDMA writes to implement a message-passing primitive, which requires for each sender-receiver pair a circular log to be stored on the receiver side; the sender appends messages to the log using one-sided RDMA writes, while the receiver periodically polls the log to process new messages. To provide both reliability and availability, FaRM uses primary-backup replication [25]. In contrast, our design of DARE's replication mechanism is based on quorums, as other Paxos-like algorithms (see Chapter 4). Moreover, DARE is designed as a wait-free replication algorithm—unlike in FaRM, the leader must not wait for replies from the remote servers. Nonetheless, FaRM is a more complex system that provides strictly serializable transactions that can span multiple servers.

Derecho [80] is a recently published open-source software library for building fault-tolerant distributed systems within a datacenter. The goal of Derecho is to enable arising time-critical applications, such as services for self-driving cars, which require a mixture of consistency and real-time responsiveness. For fast communication, Derecho relies on RDMC [15], a zero-copy reliable multicast abstractions implemented over RDMA unicast (i.e., two-sided RDMA operations). Note that, as an optimization, Derecho uses a small-message multicast protocol that uses one-sided RDMA writes. RDMC provides no fault tolerance though—when it senses a failure, RDMC stops, while the state of the disrupted multicasts remains unclear. Moreover, with multiple senders to the same group, RDMC provides no ordering on concurrent messages. Therefore, in order to offer Paxos-based guarantees (e.g., multicast atomicity), Derecho combines the virtual synchrony model[4] [21] with a variation of Paxos [20]. In particular, to take advantage of the full potential of RDMA networking, Derecho adapts Paxos to be highly asynchronous. To this end, is uses a shared state

[4]In virtual synchrony, the distributed system is modeled as a group of servers that evolves through a series of epochs, each entailing a fixed group membership.

table (SST), which is a tabular distributed shared memory abstraction. The SST is replicated across the entire group and contains one row for every server in the group; a server has both read and write access to its own row, but it can only read the rows of other servers. As a result, the SST eliminates write-write races: To share data, a server first updates its own row in its local SST and then, propagates the update remotely through one-sided RDMA writes. More significant is that the SST enables the so called monotonic predicates, which have the property that once they hold, they will continue to hold. Using such predicates, Derecho creates a series of highly-efficient protocols, such as atomic multicast or view change.

## 3.4 Atomic broadcast

Atomic broadcast algorithms (similarly to consensus algorithms) enable the implementation of state-machine replication. Actually, consensus and atomic broadcast are equivalent [34], i.e., one can be used to implement the other. Défago, Schiper, and Urbán provide a general overview of atomic broadcast algorithms [47]. Based on how total order is established, they consider five classes of atomic broadcast algorithms: fixed sequencer; moving sequencer; privilege-based; communication history; and destinations agreement. In general, they identify three different roles that can be played by a participating server—sender, receiver, and sequencer. Senders are the sources from where messages originate, receivers are the destinations of these messages, and sequencers are not necessarily a sender or a receiver[5], but they are involved in message ordering. As the names suggest, both fixed sequencer and moving sequencer algorithms establish total order through a sequencer. In both privilege-based and communication history algorithms, total order is established by the senders, while in destinations agreement algorithms, by the receivers.

In fixed sequencer algorithms [30, 125, 19, 85], the order is established by an elected sequencer that solely holds this responsibility until it is suspected of having failed. In a nutshell, the sequencer must assign a sequence number to each message sent (by any sender); then, the message (together with its sequence number) is sent further to the destinations. Depending on the communication pattern, the fixed sequencer algorithms can be split further into three variants. In the first variant, senders use unicast to send the messages to the sequencer, which then broadcast them (together with the sequence numbers) to the destinations [125, 85]. In the second variant, senders broadcast messages to both the sequencer and the destinations; then, for each message, the sequencer broadcasts a sequence number to the destinations [30, 85]. In the third variant, senders obtain sequence numbers from the sequencer (after one-to-one exchanges) and then broadcast the sequenced message to the destinations [19].

Moving sequencer algorithms [35, 109], are similar to fixed sequencer algorithms, but the role of sequencer is transfered between servers. As a result, such algorithms distribute the load among multiple servers. In a nutshell, the senders A-broadcast messages by sending them to all sequencer. Sequencers constantly circulate a token that carries both a sequence number and a list of already sequenced messages. When a sequencer receives the token, it assign a sequence number to every received message that was not yet sequenced and then, it sends all these messages to the destinations. Finally, the sequencer updates the token and sends it further to the next sequencer.

In privilege-based algorithms [8, 62, 57], senders A-broadcast when they are given the privilege to do so. The privilege is given to one sender at a time, in the form of a token. In a nutshell, the senders circulate a token that carries a sequence number. Before A-broadcasting messages, each sender must wait to receive the token. Once a sender has the token, it assigns sequence numbers to its messages and then, send the sequenced message to the destination. Finally, the sender updates the token and sends it further to the next sender. Since both

[5]Note that a server can take multiple roles at the same time.

moving sequencer and privilege-based algorithms rely on a token, they are also referred to as token-based algorithms. However, while in moving sequencer algorithms the token is circulated among the sequencers as a method to improve performance (i.e., balance the load), in privilege-based algorithms, passing the token between the senders is necessary to ensure liveness.

These first three classes of algorithms, i.e., fixed sequencer, moving sequencer, and privilege-based, rely (at any given time) on a distinguished server (i.e., a so called leader) to provide total order[6]. Therefore, the work is unbalanced—the distinguished server is on the critical path for all communication, leading to linear work per A-broadcast message (see Section 5.1 for more details). We design both AllConcur and AllConcur+ as leaderless atomic broadcast algorithms: They balance the work evenly among all servers; in addition, all servers are allowed to A-broadcast at the same time. In Chapters 5 and 6, we show that both AllConcur and AllConcur+ achieve sublinear work per A-broadcast message.

The last two classes of algorithms are leaderless—total order is determined without a leader, either by the senders (in communication history algorithms) or by the destinations (in destinations agreement algorithms). In communication history algorithms, as in privilege-based algorithms, the senders are responsible for establishing total order. However, instead of waiting for a token, the senders can A-broadcast messages at any time. Usually, A-broadcast messages carry logical timestamps (we do not survey algorithms that rely on physical timestamps). These timestamps are used by the destinations in order to decide when to safely deliver messages. Défago, Schiper, and Urbán distinguish between two variants of communication history algorithms—causal history and deterministic merge algorithms [47]. Causal history algorithms [133, 58, 126, 49, 120, 87, 68] transform the partial order provided by the timestamps into total order (i.e., the causal order [91] is extended by ordering concurrent messages [19]). However, in such algorithms, the timestamps must provide information on every participating server; thus, the size of every message is linear in the number of servers (see Section 5.1 for more details). We design both AllConcur and AllConcur+ to keep the size of messages constant—they add no overhead to messages, as no timestamps reflecting causal ordering are required.

In deterministic merge algorithms [14, 37], the messages are timestamped independently (i.e., no causal order) and delivered according to a deterministic policy of merging the streams of messages coming from each sender. Both AllConcur and AllConcur+ can be classified as deterministic merge algorithms—every message is timestamped with the round number and the merging policy is round-robin (except for messages that are lost due to failures). The Atom algorithm [14] uses the same merging policy. However, it uses an early-deciding mechanism that entails waiting for the worst-case given the actual number of failures [51]. To avoid always waiting for the worst case, both AllConcur and AllConcur+ adopt a novel early termination mechanism that we describe in Section 5.2.1. Also, servers in Atom exchange messages according to an all-to-all communication pattern, which is not suitable for large scales. In contrast, both AllConcur and AllConcur+ adopt a digraph-based communication model that relies on a sparse digraph (see Section 2.1.1). Finally, methods to decrease latency by adaptively changing the merging policy (e.g., based on sending rates [37]) could also be applied to both AllConcur and AllConcur+; for instance, at the end of a round, the system could agree on a set of servers with slow sending rates to skip one or more rounds.

In destinations agreement algorithms (as the name suggests) the destinations reach an agreement on the delivery order. Conceptually, these algorithms require no leader for establishing total order. However, in many existing destinations agreement algorithms, the destinations reach agreement either through centralized mechanisms [21, 107] or by solving

[6]Although both moving sequencer and privilege-based algorithms allow for the responsibility of providing total order to change from server to server, at any given time, only one server holds this responsibility.

consensus [34, 121, 142, 9]. Common consensus algorithms, such as Paxos, Viewstamped Replication, Zab, and Raft, rely on leader-based approaches, resulting in centralized destinations agreement algorithms and consequently, in linear work per A-broadcast message. Both AllConcur and AllConcur+ can be also classified as a destinations agreement algorithms—in every round, the servers agree on a set of messages to A-deliver. However, as we show in Chapters 5 and 6, the servers reach agreement through a completely decentralized mechanism.

In addition, some destinations agreement algorithms rely on the spontaneous total-order property (i.e., with high probability, messages broadcast in local-area networks are received in total order) as a condition for message delivery [143], as an optimization [131], or as an alternative to overcome the FLP [59] impossibility result [132]. However, breaking this property leads to potential livelocks [143], the need of solving consensus [131], or unbounded runs [132]. Neither AllConcur nor AllConcur+ makes any assumptions on the order in which messages are received.

Finally, some algorithms can fit multiple classes (so called hybrid algorithms). For example, Ring-Paxos [110], a high throughput atomic broadcast algorithm, relies on a coordinator, but the communication is done using a logical ring, similarly to the majority of privilege-based algorithms. The unicast-based version of the algorithm [111] places all servers (i.e., acceptors, proposers and learners) in a logical uni-directional ring. Therefore, each server communicates by sending messages only to its successor in the ring, which results in high-throughput [68]. However, the ring topology entails that the latency of message dissemination is linear in the number of servers, which make Ring-Paxos not suitable for large scales. The design of AllConcur entails that servers communicate via an overlay network described by any resilient digraph[7] (see Chapter 5). Moreover, during intervals with no failures, the design of AllConcur+ requires only a connected digraph (see Chapter 6). Therefore, both AllConcur and AllConcur+ enable the trade-off between high-throughput and low-latency topologies. Clearly, an alternative is to use Ring-Paxos as a reliable sequencer that enable a large group of $n$ servers to reach agreement on the order of messages (see Figure 5.1a for more details). Yet, in such a deployment, the high-throughput of Ring-Paxos results only in a constant increase in performance (given by the number of servers running Ring-Paxos). The work per A-broadcast message is still linear in $n$.

[7] The requirement for the digraph to be sparse is only needed for performance, i.e., it ensures that the work per broadcast message is sublinear. This requirement is not necessary for guaranteeing safety.

# Chapter 4

# DARE

State-machine replication is a powerful building block that enables the implementation of highly available distributed services. However, traditional state-machine replication algorithms rely on message-passing for communication and therefore, cannot benefit from emerging fast networks, such as InfiniBand. In this chapter, we present DARE[1], a novel direct access leader-based state-machine replication algorithm that uses RDMA features, such as QP disconnect and QP timeouts, in atypical ways to ensure highest performance and reliability [138]. To our knowledge, DARE is the first state-machine replication algorithm to be designed entirely with RDMA semantics—it uses RDMA for ordering and executing both read and write requests, for detecting failures and for leader election. Our evaluation at the end of this chapter shows that our DARE implementation over high-performance InfiniBand Verbs improves state-machine replication performance by more than an order of magnitude.

The remainder of this chapter states the problem in Section 4.1; gives a short overview of leader-based consensus in Section 4.2; provides a thorough description of DARE's design in Section 4.3; shows that DARE guarantees both safety and liveness in Section 4.4; models DARE's RDMA performance in a failure-free scenario in Section 4.5; analyses both the availability and reliability of DARE through a failure model for RDMA in Section 4.6; and evaluates DARE's performance in Section 4.7.

Most of the content of this chapter is reproduced from the following paper:

- Marius Poke, Torsten Hoefler. *DARE: High-Performance State Machine Replication on RDMA Networks (Extended Version)* [139]

## 4.1 Problem statement

The rapid growth of global data-analytics and web-services requires scaling single logical services to thousands of physical machines. With a constant mean time between failures per server ($\approx$2 years in modern datacenters [67]), the probability of a single-server failure grows dramatically; for instance, in system with 1,000 servers, a failure would occur more than once a day. This is especially problematic if a single server failure causes a global system outage.

State-machine replication prevents such global outages and can hide server failures while ensuring strong consistency of the overall system. State-machine replication is often at the core of global-scale services (e.g., in Google's Spanner [40] or Yahoo!'s Zookeeper [76]). However, typical state-machine replication request rates are orders of magnitude lower than the request rates of the overall systems. Thus, highly scalable systems typically utilize state-machine replication only for management tasks, while improving overall performance by relaxing request ordering [46]. This implicitly shifts the burden of consistency management to the application layer. At the same time, many services, such as airline reservation systems, require a consistent view of the complete distributed database at very high request rates [152].

[1] DARE: Direct Access REplication

In this chapter, we utilize remote direct memory access (RDMA) networking to push the limits of high-performance state-machine replication by more than an order of magnitude. High-performance RDMA network architectures such as InfiniBand or RDMA over Converged Ethernet (RoCE) are quickly adopted in datacenter networking due to their relatively low cost and high performance. However, simply emulating messages over RDMA (e.g., using the IPoIB protocol) leaves most of the performance potential untapped [65]. To exploit the whole potential of RDMA-capable networks, the careful design of new remote memory access algorithms is necessary.

## 4.2 Leader-based consensus

It is common for consensus algorithms to adopt a *leader-based* approach [92, 105, 130, 76, 122]. Leader-based consensus algorithms delegate the proposal and decision (see Section 2.2.3) to a distinguished server, i.e., the leader. The leader can both propose and decide upon values until the other servers decide to elect a new leader. Consequently, both conditions required by safety—agreement and validity—can be satisfied if there is no more than one leader at any given time. Furthermore, liveness requires that eventually a leader is elected and eventually that leader decides.

The impossibility result of Fischer, Lynch, and Paterson, states that liveness cannot be ensured in an asynchronous model where servers can fail [59]: In an asynchronous model, it is not possible to tell whether a server has failed or is just running slowly. To overcome this, DARE uses failure detectors. In particular, it assumes $\diamond\mathscr{P}$, which satisfies both completeness and eventual accuracy (see Section 2.2.4). Section 4.4.2.2 outlines how to implement $\diamond\mathscr{P}$ with RDMA semantics. To guarantee termination of leader-based consensus under the assumption of $\diamond\mathscr{P}$ a majority of the servers must be non-faulty [34]. Therefore, DARE solves consensus in a group of $n$ servers under the assumption that a maximum of $f = \lfloor \frac{n-1}{2} \rfloor$ can fail.

## 4.3 The design of the DARE algorithm

DARE is an state-machine replication algorithm that solves consensus through a leader-based approach: A distinguished leader acts as the interface between clients and the state machines. When the leader is suspected of having failed, the servers elect another leader. Each election causes the beginning of a new *term*—a period of time in which at most one leader exits. A server that wins an election during a term becomes the leader for that term. Furthermore, to make progress, DARE requires the existence of a *quorum*; that is, at least $q = \lceil \frac{n+1}{2} \rceil$ servers must agree on the next step. This ensures that after any $f = \lfloor \frac{n-1}{2} \rfloor$ failures there is still at least one non-faulty server that is aware of the previous step (since $q > f$). That server guarantees the safe continuation of the algorithm.

Existing leader-based state-machine replication algorithms and implementations, such as VR [105], Raft [130], and ZooKeeper [76], rely on message passing, often implemented over UDP or TCP channels. DARE replaces the message-passing mechanism with RDMA; it assumes that servers are connected through an interconnect with support for RDMA, such as InfiniBand [77]. To our knowledge, DARE is the first state-machine replication algorithm that can exploit the whole potential of RDMA-capable networks. All of the main sub-algorithms (see below) entail the design of special methods in order to support remotely accessible data structures; we detail the design of these methods in the following subsections.

**DARE outline.** We decompose the DARE algorithm into three main sub-algorithms that contribute to the implementation of leader-based state-machine replication:

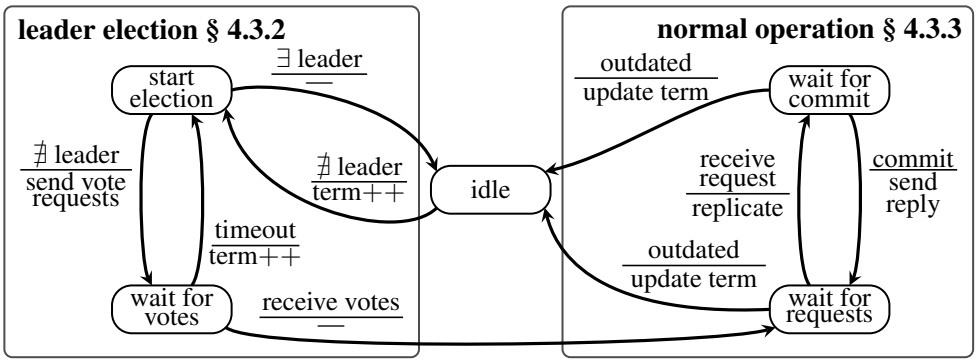



FIGURE 4.1: Outline of both leader election and normal operation algorithms of DARE. Solid boxes indicate states; arrows indicate transitions. Each transition is described by its precondition (top) and postcondition (bottom); for example, a transition labeled as $\frac{x}{y}$ has precondition $x$ and postcondition $y$; also, the lack of a postcondition is marked by a dash.

- **Leader election:** the servers elect a distinguished server as their leader (§ 4.3.2).
- **Normal operation:** the leader orders and propagates operations in a consistent manner (§ 4.3.3).
- **Group reconfiguration:** either the group's membership or size changes (§ 4.3.4).

The first two sub-algorithms—leader election and normal operation—are the essence of any leader-based state-machine replication algorithm; group reconfiguration is an extension that enables DARE to change the set of participating servers. Figure 4.1 shows an outline of both leader election and normal operation.

All servers start in an idle state, in which they remain as long as a leader exists. When a server suspects the leader to have failed (see the heartbeat mechanism in Section 4.4.2.2) it starts a new election (left side of Figure 4.1). First, it proposes itself as the leader for the subsequent term by sending vote requests to the other servers. Then, the server either becomes the leader after receiving votes from a quorum (itself included) or it starts a new election after timing out (§ 4.3.2). If another server becomes leader, the server returns to the idle state.

Once a server becomes the leader, it starts the normal operation algorithm (right of Figure 4.1). In particular, it must ensure the consistency of the state-machine replicas. Therefore, when it receives a client request, the leader replicates it on the other servers with the intention to *commit* it; for safety, a request is committed when it resides on at least a majority of servers. After a request is committed, the leader sends a reply to the client that sent the request. Finally, a leader returns to the idle state if it is *outdated*. A leader is outdated if another leader of a more recent term exists; for example, a temporary overload on the current leader can cause a majority of the servers to elect a new leader.

In the remainder of this section, we first present the basics of the DARE algorithm (§ 4.3.1): we specify the internal state (i.e., the main data structures) of a server; and we outline how clients and servers interact with each other. Then, we describe in detail the three main sub-algorithms of DARE: leader election (§ 4.3.2); normal operation (§ 4.3.3); and group reconfiguration (§ 4.3.4).

### 4.3.1 DARE basics

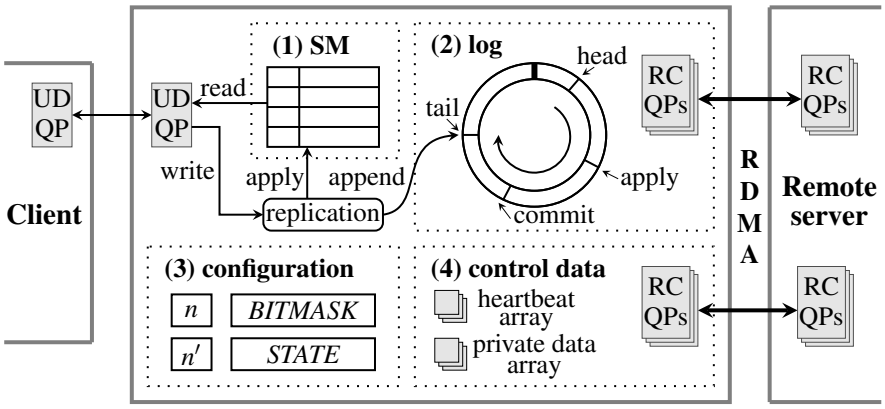



FIGURE 4.2: The internal state and interface of a DARE server. In the center is described the internal state: (1) the client state machine (SM); (2) the log; (3) the configuration; and (4) the control data. To the left is outlined how the server interacts with a client, while to the right is outlined how it interacts with other remote servers.

#### 4.3.1.1 Server internal state

The internal state of each server consists of four main data structures depicted in Figure 4.2: (1) the client state machine; (2) the log; (3) the configuration; and (4) the control data. The *state machine* is an opaque object that can be updated by the server by applying well-defined operations received from clients. For consistency, servers apply the operations in the same order; this is achieved by first appending the operations to the log.

The *log* is a circular buffer composed of entries that have sequential indexes; each entry contains the term in which it was created. Usually, log entries store operations that need to be applied to the state machine; however, some log entries are used by DARE for internal operations, such as log pruning (§ 4.3.3.3) and group reconfiguration (§ 4.3.4). For consistency, we consider a log entry to be *committed* if it resides on a majority of servers. The log is described by four dynamic pointers, which follow each other clockwise in a circle:

- **head** points to the first entry in the log; it is updated locally during log pruning (§ 4.3.3.3);
- **apply** points to the first entry that is not applied to the state machine; it is updated locally;
- **commit** points to the first not-committed log entry; it is updated by the leader during log replication (§ 4.3.3.2);
- **tail** points to the end of the log; it is updated by the leader during log replication (§ 4.3.3.2).

The *configuration* data structure is a high level description of the group of servers. It contains four fields: (1) the current group size $n$; (2) a bitmask indicating the active servers; (3) the new group size $n'$; and (4) an identifier of the current state. The last two fields are needed by DARE for resizing the group without interrupting normal operation [130]. Section 4.3.4 describes in details the role of these fields in DARE's group reconfiguration algorithm.

Finally, the *control data* consists of a set of arrays that have an entry per server. One such array is the *private data array* that is used by servers as reliable storage (§ 4.3.2.3). Another example is the *heartbeat array* used by the leader to maintain its leadership (§ 4.4.2.2). We specify the rest of the arrays as we proceed with the description of the algorithm.

**In-memory data structures: benefits and challenges.** The internal state of a DARE server consists of in-memory data structures. The benefit of an in-memory state is twofold. First, accessing in-memory data has lower latency than accessing on-disk data. Second, in-memory data structures can be remotely accessed through RDMA. In particular, in DARE, the leader uses the commit and tail pointers to manage the remote logs directly through RDMA. Thus, since the target servers are not active, they can perform other operations, such as saving the state machine on stable storage for higher reliability. Also, RDMA accesses are performed by the hardware without any interaction with the OS; this often leads to higher performance as compared to message passing [65].

The in-memory approach entails though that the entire state is volatile. Therefore, when high reliability is required, DARE uses *raw replication*. Raw replication makes an item of data reliable by scattering copies of it among different nodes. As a result, this approach can tolerate any number of simultaneous node failures as long as at least one copy of every data item remains accessible. In Section 4.6, we discuss reliability in more details.

#### 4.3.1.2 Communication interface

DARE relies on both unreliable and reliable communication. Unreliable communication is implemented over unreliable datagram (UD) Queue Pairs (QPs), which support both unicast (i.e., one-to-one) and multicast (i.e., one-to-many) transfers. The multicast support makes UD QPs practical in the context of a dynamic group membership, where the identity of the servers may be unknown. Therefore, we implement the interaction between group members and clients over UD QPs. Note that new servers joining the group act initially as clients and, thus, they also use the UD QPs to access the group (§ 4.3.4).

The InfiniBand architecture specification's Reliable Connection (RC) transport mechanism does not lose packets [77]; therefore, DARE implements reliable communication over RC QPs. Since the servers need remote access to both the log and the control data, any pair of servers is connected by two RC QPs: (1) a control QP that grants remote access to the control data; and (2) a log QP that grants remote access to the local log (see Figure 4.2).

### 4.3.2 Leader election

We adapt a traditional leader election algorithm [130, 105] to RDMA semantics: A server sends vote requests to the other servers and then it waits for votes from at least $\lfloor n/2 \rfloor$ servers, before it becomes the leader. In addition, a server cannot vote twice in the same term. Therefore, DARE guarantees at most one leader per term, which satisfies the safety requirement of leader-based consensus (see Section 4.2).

In the remainder of this section, we describe our RDMA design of the voting mechanism. Figure 4.3 outlines this mechanism during a successful leader election in a group of three servers. Although our approach is similar to a message-passing one, it requires special care when managing the log accesses. In particular, by using RDMA semantics, the leader bypasses the CPUs of the remote servers when accessing their logs; as a result, the servers are unaware of any updates to their logs. This hinders the ability of a server to participate in elections. Thus, we first outline how DARE uses *QP state transitions* to allow servers to manage the remote access to their own memory; then, we describe the voting mechanism.

#### 4.3.2.1 Managing log access

Once a QP is created, it needs to be transitioned through a sequence of states to become fully operational; moreover, at any time the QP can be locally reset to the original state, which is non-operational. Thus, DARE servers can decide on either exclusive local access

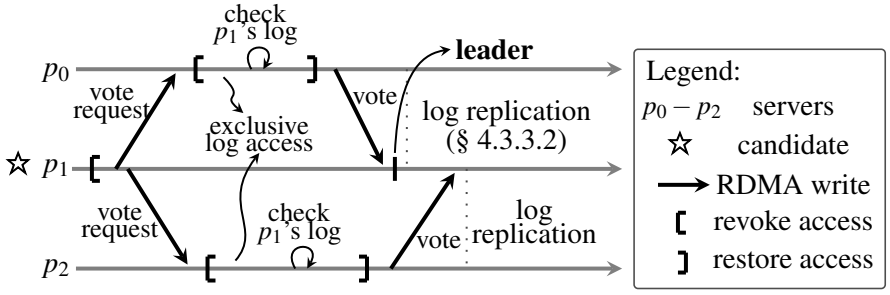



FIGURE 4.3: The voting mechanism during a successful leader election in a scenario with a group of three servers. Server $p_1$ proposes itself as candidate for the leadership of the subsequent term and it becomes leader once it receives a vote from $p_0$.

or shared remote access. For exclusive local access, the QP is reset to the original non-operational state; while for remote log access, the servers move the QP in the ready-to-send state [77], which is fully-operational. Besides managing access to their logs, DARE servers use QP state transitions for both connecting and disconnecting servers during group reconfiguration (§ 4.3.4).

#### 4.3.2.2 Becoming a candidate

The leader election algorithm starts when a server suspects the leader to have failed. In Figure 4.3, server $p_1$ starts an election by revoking remote access to its log; this ensures that an outdated leader cannot update the log. Then, it proposes itself as a *candidate* for the leadership of the subsequent term. That is, it sends vote requests to the other servers: It updates its corresponding entry in the *vote request array* (one of the control data arrays) at all other servers by issuing RDMA write operations (see Figure 4.3). An entry in the vote request array consists of all the information a server requires to decide if it should vote for the candidate: the candidate's current term and both the index and the term of the candidate's last log entry (see Section 4.3.2.3).

Depending on the internal state of the candidate, we distinguish between three possible outcomes (depicted in the left side of Figure 4.1): (1) the candidate becomes the leader if it receives the vote from at least $\lfloor n/2 \rfloor$ servers; (2) it decides to support the leadership of another candidate more suited to become leader (§ 4.3.2.3); or (3) otherwise, it starts another election after a timeout period. The candidate restores remote log access for every server from which it received a vote; this ensures that a new leader can proceed with the log replication algorithm (§ 4.3.3.2). In Figure 4.3, candidate $p_1$ becomes leader after it receives a vote from server $p_0$; this is sufficient, since $n = 3$. Once it becomes leader, $p_1$ starts replicating the log of the servers from which it received votes.

In general, two or more candidates may receive together $\lfloor n/2 \rfloor$ or more votes; however, none receives enough to become leader. Consecutively, a new election starts. If the candidates are synchronized, this scenario can repeat several times, reducing the availability (or even the liveness) of the algorithm. To avoid this, DARE servers use randomized timeouts [130].

#### 4.3.2.3 Answering vote requests

Servers not aware of a leader periodically check their local vote request array for incoming requests. They only consider requests for the leadership of a higher (more recent) term than their own; on receiving a valid request, the servers increase their own term. In Figure 4.3,

servers $p_0$ and $p_2$ receive vote requests from candidate $p_1$. Both servers grant their vote after first checking that the candidate's latests log entry is at least as recent as their own; an entry is more recent than another if it has either a higher term or the same term but a higher index [130]. Note that while performing the check, both servers need exclusive access to their own logs (see Figure 4.3). By checking a candidate's latest log entry, the servers ensure that the log of a possible leader contains the most recent entry among a majority of servers. In Section 4.4.1, we show that this property is essential for DARE's safety.

A server's volatile internal state introduces an additional challenge. The server may fail after voting for a candidate and then recover during the same term. If after recovery, it receives a vote request from another candidate, but for the same term, the server could grant its vote. Thus, two servers may become leaders during the same term, which breaks the safety of our algorithm. To avoid such scenarios, prior to answering vote requests, each server makes its decision reliable by replicating it via the private data array (§ 4.3.1.1).

**RDMA vs. message-passing: leader election.** Our RDMA design of leader election increases DARE's availability. When the leader fails, the distributed service becomes unavailable until a new leader is elected. For the leader election to start, the servers need to first detect the failure (§ 4.4.2.2). Then, the election time depends on the period a candidate waits for votes before restarting the election. This period needs to be large enough for the vote requests to reach the servers and at least $\lfloor n/2 \rfloor$ votes to arrive back at the candidate. The RDMA-capable interconnect allows us to reduce this period; thus, our design increases availability. In addition, due to the in-memory approach, servers do not need to write their vote on stable storage before answering a vote request.

### 4.3.3 Normal operation

The normal operation algorithm entails the existence of a sole leader that has the support of at least a majority of servers (including itself). The leader is responsible for three tasks: serving clients; managing the logs; and, if needed, reconfiguring the group. In the remainder of this section, we describe the first two tasks; we defer the discussion of group reconfiguration to Section 4.3.4. First we specify how clients interact with DARE. Then, we present a log replication algorithm designed entirely for RDMA (§ 4.3.3.2); also, we outline a log pruning mechanism that prevents the overflowing of the logs.

#### 4.3.3.1 Client interaction

Clients interact with the group of servers by sending requests through either multicast or unicast. To identify the leader of the group, clients send their first request via multicast. Multicast requests are considered only by the leader. Once the leader replies, clients send subsequent requests directly to the leader via unicast. However, if the request is not answered in a predefined period of time, clients re-send the request through multicast. Note that this is also the case when sending requests to an outdated leader, since it cannot answer client requests (see below). Also, the current implementation assumes that a client waits for a reply before sending the subsequent request. Yet, DARE handles different clients asynchronously: The leader can execute, at the same time, requests from multiple clients. This increases the algorithm's throughput (see Figure 4.9a in Section 4.7).

Regardless of the nature of the state machine, clients can send either write or read requests:

**Write requests.** Write requests contain operations that alter the state machine; for example, a write request may consist of updating the value of a key in a key-value store. Such

operations need to be applied to all state-machine replicas in the same order. Therefore, when receiving a write request, the leader stores the enclosed operation into an entry that is appended to the log. Then, it replicates the log entry on other servers with the purpose to commit it (see Section 4.3.3.2). As a safety requirement, each DARE server applies only operations stored in committed log entries.

Write requests may contain operations that are not idempotent, i.e., they change the state-machine replicas every time they are applied. DARE enforces *linearizable semantics* [72]: It guarantees that each operation is applied only once. First, every client tags its requests with strictly increasing sequence numbers. Then, for each client, the leader keeps track of the sequence number of the last executed request; moreover, to ensure that these sequence numbers propagate to subsequent terms, the leader stores them in the log entries alongside the operations. Therefore, subsequent leaders can identify an already executed request and avoid re-applying the enclosed operation.

Furthermore, to increase the throughput of strongly consistent writes, DARE executes write requests in batches: The leader first appends the operations of all consecutively received write requests to the log; then, it replicates all the entries at once.

**Read requests.** Read requests contain operations that do not alter the state machine; for example, a read request may consist of getting the value of a key from a key-value store. For such operations, replication is not required: For efficiency, the leader answers read requests directly from the local state machine. To ensure that reads do not return stale data, DARE imposes two constraints: (1) an outdated leader cannot answer read requests; and (2) a leader with an outdated state machine cannot answer read requests.

First, to verify whether a leader is outdated, DARE uses a property of leader election—any successful election requires at least a majority of servers to increase their terms (cf. § 4.3.2.3). Therefore, before answering a read request, the leader reads the term of at least $\lfloor n/2 \rfloor$ servers; if it finds no term higher than its own, then it can safely answer the read request. As an optimization, the leader verifies whether it is outdated only once for a batch of consecutively received read requests; thus, DARE's read throughput increases.

Second, before answering a read request, the leader must ensure that all operations stored in committed log entries are applied to the local state machine. In Section 4.4.1, we show that the leader's log contains all the committed entries that store operations not yet applied by all non-faulty servers. However, a new leader may not be aware of all the committed entries (§ 4.3.3.2); therefore, the local SM is outdated. As a solution, a new leader appends to its log an entry with no operation. This starts the log replication algorithm that commits also all the preceding log entries.

#### 4.3.3.2 Log replication

The core of the normal operation algorithm is *log replication*—the leader asynchronously replicates log entries on the other servers with the intention to commit them (see right side of Figure 4.1). In DARE, log replication is performed entirely through RDMA: The leader writes its own log entries into the logs of the remote servers. Yet, after a new leader is elected, the logs can contain not-committed entries. These entries may differ from the ones stored at the same position in the new leader's log; for example, Figure 4.4a shows the logs after server $p_1$ becomes the leader of a group of three servers. Before a newly elected leader can replicate log entries, it must first remove all the remote not-committed entries. Therefore, we split the log replication algorithm into two phases: (1) log adjustment; and (2) direct log update.

**Log adjustment.** During the direct log update (see below), the leader "lazily" updates the commit pointers of the remote servers; by lazy update we mean that there is no need to wait

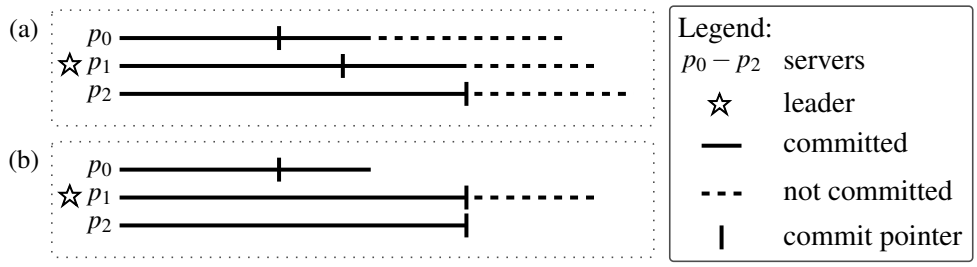



FIGURE 4.4: The logs in a group of three servers: (a) after server $p_1$ is elected leader; and (b) after log adjustment. For clarity, the apply pointers are omitted.

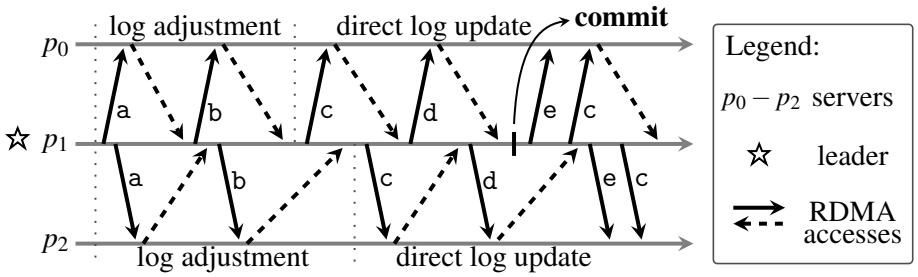



FIGURE 4.5: The RDMA accesses during log replication: (a) read the remote not-committed entries; (b) write the remote tail pointer; (c) write the remote log; (d) write the remote tail pointer; and (e) write the remote commit pointer. Solid arrows denote RDMA accesses, while dashed arrows denote acknowledgements indicating that the matching accesses were completed.

for completion. As a result, servers may not be aware of all their committed log entries. Therefore, adjusting a remote log by setting its tail pointer to the corresponding commit pointer may remove committed entries (see server $p_0$'s log in Figure 4.4a). A correct approach sets the remote tail pointer to the first not-committed entry. Consequently, in DARE, the leader adjusts a remote log by performing two subsequent RDMA accesses (labeled by `a` and `b` in Figure 4.5): first, it reads the remote not-committed entries; and second, it updates the remote tail pointer to indicate the first non-matching entry when compared to its own log. In addition, the leader updates its own commit pointer. Figure 4.4b shows the logs of the three servers after the log adjustment phase is completed: $p_1$ discards the not-committed entries from both $p_0$ and $p_2$, and then updates its own commit pointer.

Reading the remote not-committed entries could be rather costly in case of large operations. Yet, the operations are not needed for the leader to identify the first non-matching entry. Thus, as an optimization, before granting log access to a new leader, each server creates a buffer with only the index and the term of all of its not-committed entries. As a result, the leader avoids large data transfers over the network.

**Direct log update.** The second phase of log replication consists of three RDMA accesses (labeled by `c`, `d` and `e` in Figure 4.5). First, for each adjusted remote log, the leader writes all entries between the remote and the local tail pointers. Second, the leader updates the tail pointers of all the servers for which the first access completed successfully. To commit log entries, the leader sets the local commit pointer to the minimum tail pointer among at least a majority of servers (itself included). Third, for the remote servers to apply the just committed entries, the leader "lazily" updates the remote commit pointers.

Figure 4.5 shows an example of RDMA accesses during log replication for a scenario with a group of three servers. Solid arrows denote RDMA accesses, while dashed arrows

denote acknowledgements indicating that the matching accesses were completed. Once $p_1$, the leader, receives the confirmation that server $p_0$'s log is adjusted, it starts updating it, although it is not yet aware that server $p_2$'s log is adjusted (the access is delayed). When the delayed access completes, $p_1$ can also start the direct log update phase for server $p_2$. Furthermore, $p_1$ commits its log entries once it updates the tail pointer of server $p_0$ (since there are three servers in total).

**RDMA vs. message-passing: log replication.** Replicating the logs through RDMA has several benefits over the more traditional message-passing approach. RDMA accesses remove the overhead on the target, which has two consequences: first, the leader commits log entries faster; and second, the servers are available for other tasks, such as recovery (§ 4.3.4). Moreover, RDMA allows for servers with a faulty CPU, but with both NIC and memory working, to be remotely accessible during log replication, hence, increasing both availability and reliability (§ 4.6). Finally, RDMA allows for efficient log adjustment: In DARE, log adjustment entails two RDMA accesses regardless of the number of non-matching log entries; yet, in Raft [130] for example, the leader must send a message for each non-matching log entry.

#### 4.3.3.3 Log pruning: removing applied entries

Every server applies the operations stored in the log entries between its apply and commit pointers; once an operation is applied, the server advances its apply pointer. When an operation is applied by all the non-faulty servers in the group, the entry containing it can be removed from the log. Thus, the leader advances its own head pointer to the smallest apply pointer in the group; then, it appends to the log a `HEAD` entry that contains the new head pointer. Servers update their head pointer only when they encounter a committed `HEAD` entry; thus, all subsequent leaders will be aware of the updated head pointer. Furthermore, when the log is full, the leader blocks until the remote servers advance their apply pointers. To avoid waiting, the leader can remove the server with the lowest apply pointer on the grounds that it hinders the performance of the entire group (cf. [45]).

### 4.3.4 Group reconfiguration

DARE is intended for dynamical environments where servers can fail at any time. Thus, the group of servers can modify both its membership and its size; we refer to this as *group reconfiguration*. DARE handles group reconfigurations through the configuration data structure (§ 4.3.1.1). A configuration can be in three states: (1) a *stable* state that entails a group of $n$ servers with the non-faulty servers indicated by a bitmask; (2) an *extended* state used for adding servers to a full group (see below); and (3) a *transitional* state that allows for the group to be resized without interrupting normal operation [130]. The last two states require the new group size $n'$ to be set.

We define three operations that are sufficient to describe all group reconfiguration scenarios: (1) remove a server; (2) add a server; and (3) decrease the group size. For example, since a server's internal state is volatile, a transient failure entails removing a server followed by adding it back; also, increasing the size entails adding a server when the group is already full. The three operations can be initiated only by a leader in a stable configuration. Each operation may entail multiple phases. Yet, each phase contains the following steps: the leader modifies its configuration; then, it appends to the log an entry that contains the updated configuration (a `CONFIG` entry); and once the `CONFIG` entry is committed, the phase completes and a possible subsequent phase can start. When a server encounters a `CONFIG` log entry, it updates its own configuration accordingly regardless of whether the entry is committed. In

the remainder of this section, we first describe how DARE implements the three operations; then, we outline how a server recovers its internal state. If not stated otherwise, we assume all configurations to be stable.

**Removing a server.** A server may be removed in one of the following cases: the log is full and cannot be pruned (§ 4.3.3.3); the group size is decreased; or the leader suspects the server to have failed. The leader detects failed (or unavailable) servers by using the QP timeouts provided by the RC transport mechanism [77]. In all cases, removing a server is a single-phase operation. The leader disconnects its QPs (§ 4.3.2.1) with the server and then it updates the bitmask of its configuration accordingly. Also, it adds a `CONFIG` log entry with the updated configuration; once the log entry is committed, the server is removed.

**Adding a server.** Adding a server to a group is similar to removing a server; the only difference is that the QPs with the server must be connected instead of disconnected. Yet, if the group is full, adding a server requires first to increase the group size, which, without previously adding a server, decreases the fault tolerance of the group. Intuitively, this is because the new group starts already with a failure since the new server is not yet added. Thus, adding a server to a full group is a three-phase operation: (1) adding the server; (2) increasing the group size; (3) stabilizing the configuration.

First, the leader establishes a reliable connection with the server; also, it creates an extended configuration with $n' = n + 1$. This configuration allows the added server to recover; yet, the server cannot participate in DARE's sub-algorithms. Second, the leader increases the group size without interrupting normal operation [130]. In particular, it moves the configuration to a transitional state, in which all servers are participating in DARE's sub-algorithms. The servers form two groups—the original group of $n$ server and the new group of $n'$ servers; majorities from both groups are required for both electing a leader and committing a log entry, such as the `CONFIG` entry containing the transitional configuration. Finally, the leader stabilizes the configuration: It sets $n$ to the new size $n'$ and it moves back into the stable state.

**Decreasing the group size.** On the one hand, adding more servers leads usually to higher reliability (see Figure 4.7 in Section 4.6). On the other hand, it decreases the performance, since more servers are required to form a majority. Therefore, DARE allows the group size to be decreased. Decreasing the group size is a two-phase operation: first, the leader creates a transitional configuration that contains both the old and the new sizes; and second, it stabilizes it by removing the extra servers from the end of the old configurations. Note that the leader can be one of the servers that need to be removed; in this case, the leader removes itself once the decrease operation completes.

**Recovery.** When added to the group, a server needs to recover its internal state before participating in DARE's sub-algorithms; in particular, it needs to retrieve both the state machine and the log. To retrieve the state machine, the new server asks any server, except for the leader, to create a snapshot of its state machine; then, it reads the remote snapshot. Once the state machine is recovered, the server reads the committed log entries of the same server. After it recovers, the server sends a vote to the leader as a notification that it can participate in log replication. Note that the recovery is performed entirely through RDMA.

**RDMA vs. message-passing: recovery.** Our RDMA approach reduces the impact that recovery has on normal operation. The reason for this is twofold. First, contrary to message-passing state-machine replication, in DARE, the leader manages the logs directly without involving the CPUs of the remote servers; thus, servers can create a snapshot of their state

machines without interrupting normal operation. Second, the new server can retrieve both the state machine and the log of a remote server directly through RDMA.

## 4.4 Safety and liveness

In this section, we show that DARE guarantees both safety and liveness.

### 4.4.1 Safety argument

In addition to the safety requirement of consensus (see Section 2.2.3), DARE guarantees the following safety property necessary for state-machine replication: each replica of the state machine executes the same sequence of operations. DARE exhibits similar properties as existing state-machine replication algorithms (e.g., Raft [130]). Therefore, we use an analogous approach to show that safety holds at all times. In particular, we show that DARE satisfies two properties [130]:

- (Log matching) If two logs have an identical entry, then they have all the preceding entries identical as well.
- (Leader completeness) The log of every leader contains all already-committed entries.

To account for log pruning, the leader completeness property considers only log entries that store operations not yet applied by all non-faulty servers.

DARE servers apply only operations enclosed in committed entries. In addition, a server applies the operation enclosed in a log entry only after it applies all the operations stored in log entries with lower indexes. Finally, the leader completeness property ensures that every committed entry eventually resides in the log of every non-faulty server. Therefore, DARE guarantees the safety of state-machine replication.

#### 4.4.1.1 Log matching

We show that the log matching property holds through complete induction on the term number. Initially, all logs are empty; therefore, the base case holds. Then, in the inductive step, we prove that if the property holds until the beginning of a given term, then it holds throughout that term. DARE guarantees that there is at most one leader during the term. If no leader exists, then the logs remain unmodified throughout the term and hence, the property holds. If there is a leader though, then it must adjust each remote log before updating it (cf. the log replication algorithm described in Section 4.3.3.2). Once a log is adjusted, its most recent entry is identical to an entry in the leader's log; thus, according to the inductive step, all the preceding entries are identical as well. Moreover, to update the remote logs, the leader copies sequences of entries from its own log. As a result, the log matching property holds at all times.

#### 4.4.1.2 Leader completeness

To guarantee the leader completeness property, DARE relies on the following propositions:

**Proposition 4.4.1.** *In order to become a leader, a server requires votes from at least a majority of servers (see Section 4.3.2.2).*

**Proposition 4.4.2.** *A leader's log contains the most recent entry among at least a majority of servers (see Section 4.3.2.3).*

**Proposition 4.4.3.** *Every committed entry is replicated on at least a majority of servers (see Section 4.3.3.2).*

**Proposition 4.4.4.** *The log matching property holds.*

Our argument that, given these four propositions, the leader completeness property holds is based on a proof provided by Raft [130]. The proof is by contradiction. We assume that a log entry $x$, that is committed by the leader $p_i$ of a term $T_i$, is not stored in the log of the leader of some subsequent term. Let $T_j > T_i$ be the smallest such term and $p_j$ its leader. Then, when $p_j$ was elected, $x$ was not in its log (since leaders never discard their own log entries). First, $p_i$ replicated $x$ on at least a majority of servers (cf. Proposition 4.4.3); second, $p_j$ received votes from at least a majority of servers (cf. Proposition 4.4.1). Therefore, there is at least one server that both has $x$ in its log and voted for $p_j$. Let $p_k$ be such a server.

Clearly, $p_i$ added $x$ to $p_k$'s log before $p_k$ voted for $p_j$ (otherwise, $p_i$ would not have had access to $p_k$'s log). Moreover, $p_k$ must have had $x$ in its log when it voted for $p_j$: Since all the leaders in between $p_i$ and $p_j$ had $x$ in their log, $x$ could not have been removed during log adjustment. Therefore, $p_k$ considered that $p_j$'s latests log entry is at least as recent as its own (cf. Proposition 4.4.2). We distinguish two cases. First, $p_j$'s and $p_k$'s latest log entries are from the same term, with $p_j$'s entry having a higher or equal index. Consequently, all of $p_k$'s log entries are included into $p_j$'s log, which contradicts the assumption that $x$ is not in $p_j$'s log. Second, $p_j$'s latests log entry is from a higher term than $p_k$'s latests log entry. Clearly, this term is $\geq T_i$, since $x$ is in $p_k$'s log. This means that $p_j$'s latests log entry was added by a leader that was already having $x$ in its log and thus, $p_j$'s log must also contain $x$ (cf. Proposition 4.4.4), which again leads to a contradiction. As a result, the leader completeness property holds.

### 4.4.2 Liveness argument

In theory, DARE's liveness is guaranteed by the following condition: there is a time after which a server $p_*$ is successfully elected to be leader and then answers all client requests. This entails that $p_*$ must have the support of at least a majority of the servers (itself included). The eventual accuracy property of $\diamond\mathscr{P}$ [34] ensures that eventually all non-faulty servers always trust each other (see Section 2.2.4). Moreover, according to the assumption made in Section 4.2, there are at least a majority of non-faulty servers. Without loss of generality, we assume one of these servers to be $p_*$. Therefore, from a theoretical point of view, DARE guarantees liveness.

In practice, no leader can be trusted forever by other servers. However, it is sufficient for a non-faulty leader to have the support of a majority of the servers for "sufficiently long" [2] so that it can answer at least one client request. Then, DARE must ensure that every time a leader fails (or loses the support of a majority) another leader is elected. We split this requirement into two conditions: (1) all the non-faulty servers eventually detect a faulty leader; and (2) once a faulty leader is detected, a new leader is eventually elected. The former condition results from the completeness property of $\diamond\mathscr{P}$, i.e., eventually, every faulty server is suspected to have failed by every non-faulty server [34]. Moreover, the former condition ensures that sufficient servers (i.e., at least a majority) participate in the leader election sub-algorithm; this, together with the randomized timeouts [130] used to restart a leader election, ensure the latter condition.

In the remainder of this section, we first describe how we utilize InfiniBand's timeout mechanisms [77] to obtain a model of partial synchrony required by $\diamond\mathscr{P}$ [34]. Then, we outline how to implement $\diamond\mathscr{P}$ with RDMA semantics.

#### 4.4.2.1 Synchronicity in RDMA networks

Synchronicity in the context of processors implies that there is a fixed bound on the time needed by a processor to execute any operation; intuitively, this guarantees the responsiveness of non-faulty processors. Since DARE uses RDMA for log replication, the processors are the NICs of the servers. These NICs are special-purpose processors that are in charge solely of the delivery of network packets at line-rate; that is, NICs avoid nondeterministic behavior, such as that introduced by preemption in general-purpose CPUs. Therefore, we can assume a bound on the execution time of NIC operations.

Synchronous communication requires a bound on the time within which any packet is delivered to a non-faulty server. Time can generally not be bounded in complex networks, however, datacenter networks usually deliver packets within a tight time bound. InfiniBand's reliable transport mechanism does not lose packets and notifies the user if the transmission experiences unrecoverable errors [77]: It uses Queue Pair timeouts that raise unrecoverable errors in the case of excessive contention in the network or congestion at the target. Thus, the RC service of InfiniBand offers a communication model where servers can ascertain deterministically if a packet sent to a remote server was acknowledged within a bounded period of time.

#### 4.4.2.2 Leader failure detection

The $\diamond\mathscr{P}$ FD used by DARE to detect failed leaders is based on an heartbeat mechanism implemented with RDMA semantics. Initially, every server suspects every other server to have failed. Then, the leader starts sending periodic heartbeats by writing its own term in the remote heartbeat arrays (see Section 4.3.1.1). Every other server checks its heartbeat array regularly, with a period $\Delta_{to}$: First, it selects the heartbeat with the most recent term; then, it compares this term with its own. If the terms are equal, then the leader is non-faulty; thus, the server extends its support. If its own term is smaller, then a change in leadership occurred; thus, the server updates its own term to indicate its support. Otherwise, the server assumes the leader has failed; thus, the completeness property holds [34].

In addition, when a server finds a heartbeat with a term smaller than its own, it first increments $\Delta_{to}$ to ensure that eventually a non-faulty leader will not be suspected; thus, the eventual accuracy property holds [34]. Then, it informs the owner of the heartbeat that it is an outdated leader, so it can return to the idle state (see Figure 4.1 in Section 4.3).

Finally, once a server votes for a candidate, it resets the time elapsed since the latest check of the heartbeat array; thus, the candidate has a chance to become leader before the server starts a new leader election.

## 4.5 Performance analysis

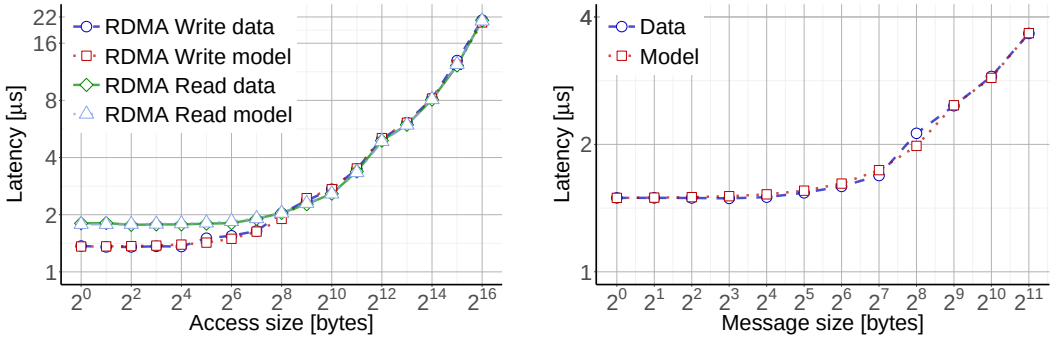
In this section, we analyze the performance of DARE during normal operation. We first provide a model of RDMA performance, which we then use to estimate both DARE's read and write latency.

### 4.5.1 Modeling RDMA performance

We estimate the performance of RDMA operations through a modified LogGP model [7]. The LogGP model consist of the following parameters: the latency $L$; the overhead $o$; the gap between messages $g$; the gap per byte for long messages $G$; and the number of processes or servers, which we denote by $n$ (instead of $P$). We make the common assumption that $o > g$ [7]; also, we assume that control packets, such as write acknowledgments and

| $o_{poll} = 0.07\mu s$ | RDMA/rd | RDMA/wr | | UD | |
|---|---|---|---|---|---|
| | | | inline | | inline |
| $o$ $[\mu s]$ | 0.29 | 0.26 | 0.36 | 0.62 | 0.47 |
| $L$ $[\mu s]$ | 1.38 | 1.61 | 0.93 | 0.85 | 0.54 |
| $G$ $[\mu s/KB]$ | 0.75 | 0.76 | 2.21 | 0.77 | 1.92 |
| $G_m$ $[\mu s/KB]$ | 0.26 | 0.25 | - | - | - |

TABLE 4.1: LogGP parameters measured on a 12-node InfiniBand cluster.




(A) RDMA Operations

(B) Messages over UD

FIGURE 4.6: Evaluation of the RDMA performance models. Both the data and the LogGP parameters are gathered from a 12-node InfiniBand cluster with an MTU of 4096 bytes.

read requests, are of size one byte. Moreover, we readjust the model to fit the properties of RDMA communication. In particular, we make the following assumptions: (1) the overhead of the target of the access is negligible; (2) the latency of control packets is integrated into the latency of RDMA accesses; (3) for large RDMA accesses, the bandwidth increases after transferring the first MTU bytes; (4) for RDMA write operations, $L$, $G$, and $o$ depend on whether the data is sent inline; and (5) $o_{poll}$ is the overhead of polling for completion. Table 4.1 specifies the LogGP parameters for the system used for evaluation, i.e., a 12-node InfiniBand cluster (see Section 4.7).

According to the assumptions above, the time of either writing or reading $s$ bytes through RDMA is estimated by

$$\begin{cases} o_{in} + L_{in} + (s-1)G_{in} + o_{poll} & \text{if inline} \\ o + L + (s-1)G + o_{poll} & \text{if } s \leq m \\ o + L + (m-1)G + (s-m)G_m + o_{poll} & \text{if } s > m, \end{cases} \tag{4.1}$$

where $m$ is the MTU of the system, $G$ is the gap per byte for the first $m$ bytes, and $G_m$ is the gap per byte after the first $m$ bytes. Figure 4.6a plots the latency of both read and write RDMA operations.

Besides RDMA operations, DARE also uses unreliable datagrams (UDs). To estimate the time of UD transfers, we use the original LogGP model; thus, the time of sending $s$ bytes over UD is

$$\begin{cases} 2o_{in} + L_{in} + (s-1)G_{in} & \text{if inline} \\ 2o + L + (s-1)G & \text{otherwise.} \end{cases} \tag{4.2}$$

Figure 4.6b plots the latency of messages over UD. Both RDMA and UD models fit the data on our system with coefficients of determination larger than 0.99. The good fit of the model

is also visible in Figures 4.6a and 4.6b.

### 4.5.2 An RDMA performance model of DARE

DARE is designed for high-performance state-machine replication. Its performance is given by the request latency—the amount of time clients need to wait for requests to be answered. During normal operation, client requests have two parts: (1) the UD transfer, which entails both sending the request and receiving a reply; and (2) the RDMA transfer, which consist of the leader's remote memory accesses. We use Equations (4.1) and (4.2) from Section 4.5.1 to estimate the latency of both UD and RDMA transfers. For readability, we consider the gap per byte $G$ only for the $s$ bytes of either the read or written data.

The UD transfer entails two messages: one short that is sent inline (request for reads and replies for writes); and one long that transfers the data. Thus, the latency of the UD transfer is bounded (due to the above simplification) by

$$t_{UD} \geq 2o_{in} + L_{in} + \begin{cases} 2o_{in} + L_{in} + (s-1)G_{in} & \text{if inline} \\ 2o + L + (s-1)G & \text{otherwise} \end{cases}.$$

The latency introduced by RDMA accesses depends on the request type. For read requests, the leader needs to wait for at least $q-1$ RDMA reads to complete (with $q = \lceil \frac{n+1}{2} \rceil$). Thus, the latency of the RDMA transfer in case of read requests is bounded by

$$t_{RDMA/rd} \geq (q-1)o + \max\{fo, L\} + (q-1)o_{poll},$$

with $f = \lfloor \frac{n-1}{2} \rfloor$ the maximum number of faulty servers. The max function indicates the overlap between the overhead of issuing the last $f$ reads and the latency of the $(q-1)$st one.

For write requests, the leader needs to go through the steps of log replication. Yet, the logs are adjusted only once per term; thus, assuming a fairly stable leader, the latency of log adjustment is negligible. During the direct log update phase, the leader accesses the logs of at least $q-1$ servers; for each, it issues three subsequent RDMA write operations (see Figure 4.5 in Section 4.3.3.2). Thus, the latency of the RDMA transfer in case of a write request is bounded by

$$\begin{aligned} t_{RDMA/wr} \geq & 2(q-1)o_{in} + L_{in} + 2(q-1)o_{poll} + \\ & \begin{cases} (q-1)o_{in} + \max\{fo_{in}, L_{in} + (s-1)G_{in}\} & \text{if inline} \\ (q-1)o + \max\{fo, L + (s-1)G\} & \text{otherwise} \end{cases}. \end{aligned}$$

Similar to read requests, the max function indicates the overlap between the last $f$ log update operations and the latency of the $(q-1)$st one. In Section 4.7, Figure 4.8 compares these bounds with measurements gathered on our system.

## 4.6 Fine-grained failure model

RDMA requires a different view of a failing system than message passing. In message passing, a failure of either the CPU or OS (e.g., a software failure) disables the whole node because each message needs both CPU and memory to progress. In RDMA systems, memory may still be accessed even if the CPU is blocked (e.g., the OS crashed) due to its OS bypass nature.

To account for the effects of RDMA, we propose a failure model that considers each individual component—CPU, main memory (DRAM), and NIC—separately. We make the common assumption that each component can fail independently of the other components in

| Component | AFR | MTTF | Reliability |
|---|---|---|---|
| Network [52, 67] | 1.00% | 876,000 | 4-nines |
| NIC [52, 67] | 1.00% | 876,000 | 4-nines |
| DRAM [70] | 39.5% | 22,177 | 2-nines |
| CPU [70] | 41.9% | 20,906 | 2-nines |
| Server [67] | 47.9% | 18,304 | 2-nines |

TABLE 4.2: Worst case scenario reliability data. The reliability is estimated over a period of 24 hours and expressed in the number of nines notation; the mean time to failure (MTTF) is expressed in hours.

the system [1, 31, 118]. For example, the CPU may execute a failed instruction in the OS and halt, the NIC may encounter too many bit errors to continue, or the memory may fail ECC checks. We also make the experimentally verified assumption that a CPU/OS failure does not influence the remote readability of the memory. Finally, we assume that the network (consisting of links and switches) can also fail.

Various sources provide failure data of systems and system components [61, 145, 127, 66, 52, 134, 70, 112]. Yet, systems range from very reliable ones with annualized failure rates (AFRs) per component below 0.2% [112] to relatively unreliable ones with component failure log events at an annual rate of more than 40% [70] (here we assume that a logged error impacted the function of the device). Thus, it is important to observe the reliability of the system that DARE is running on and adjust the parameters of our model accordingly. For the sake of presentation, we pick the *worst case* for DARE, i.e., the highest component errors that we found in the literature. Table 4.2 specifies this for the main components over a period of 24 hours.

### 4.6.1 Availability: zombie servers

We refer to servers with a blocked CPU, but with both a working NIC and memory as *zombie servers*. Zombie servers account for roughly half of the failure scenarios (cf. Table 4.2). Due to their non-functional CPU, zombie servers cannot participate in some algorithms, such as leader election. Yet, DARE accesses remote memory through RDMA operations that consume no *receive request* on the remote QP, and hence, no *work completions* [77]. Consequently, a zombie server's log can be used by the leader during log replication, increasing DARE's availability. Note that the log can be used only temporarily, since it cannot be pruned and eventually the leader will remove the zombie server. Moreover, even in case of permanent CPU failures, zombie servers may provide sufficient time for recovery without losing availability.

### 4.6.2 Reliability

As briefly mentioned in Section 4.3.1.1, our design exploits the concept of memory reliability through raw replication. In particular, DARE uses raw replication in two situations: (1) explicitly by a server before answering a vote request during leader election; and (2) implicitly by the leader during log replication. In both situations, at least $q = \left\lceil \frac{n+1}{2} \right\rceil$ replicas are created. Thus, DARE's reliability is given by the probability that no more than $q-1$ servers experience a memory failure (cf. Table 4.2, the failures probabilities of both NIC and network are negligible).

To compute the reliability of DARE's in-memory approach, we assume that the components are part of non-repairable populations: Having experienced a failure, the same component can rejoin the system; however, it is treated as a new individual of the population. The reliability of non-repairable populations is estimated by *lifetime distribution models* (LDMs). An LDM is a probability density function $f(t)$ defined over the time range $t = (0,\infty)$. The corresponding cumulative distribution function $F(t)$, gives the probability that a randomly selected individual fails by time $t$. We denote the probability of a single component to fail during a time interval $(t_{j-1},t_j]$ by $h_j$. We compute $h_j$ from the probability to fail by the end of the interval, conditioned by the probability to be alive at the beginning of the interval. We refer to this time interval as the *epoch* $j$; note that an epoch is bounded by the expected lifetime of the components. Thus, (cf. Bayes' theorem)

$$h_j = \frac{F(t_j) - F(t_{j-1})}{1 - F(t_{j-1})}.$$

Further, we introduce the discrete random variable $X_j$ to denote the number of components of a certain type that fail during epoch $j$. Also, we consider $k_j$ components to be available at the beginning of the epoch. Since all $k_j$ components have the same failure probability $h_j$, the random variable $X_j$ follows a binomial distribution, i.e., $X_j \sim B(k_j,h_j)$.

Components that failed during an epoch may not recover until the beginning of the subsequent epoch; therefore, $k_j$ varies. We express this variation through a discrete random variable $K_j$ that denotes the number of available components at the beginning of epoch $j$. We simplify the analysis of $K_j$, by assuming that the recovery time of a component is less than the length of an epoch; such an assumption is valid since the only side effect of increasing the epoch length is higher failure probabilities. Under this assumption, $K_j = k_1 - X_j$, where $k_1$ is the total number of available components.

We can express the probability of $x$ components failing during epoch $j$ by the following probability mass function (PMF):

$$Pr(X_j = x) = \sum_{k=x}^{k_1} Pr(K_j = k) \binom{k}{x} h_j^x (1-h_j)^{k-x},$$

where $Pr(K_j = k) = Pr(X_{j-1} = k_1 - k)$ gives the probability that $k \leq k_1$ components are available at the beginning of epoch $j$. The formula for $Pr(K_j = k)$ entails a recurrence with the initial value given by $Pr(X_1 = x) = \binom{k_1}{x} h_1^x (1-h_1)^{k_1-x}$.

Of particular interest to our analysis of reliability is the total number of unavailable components during an epoch $j$. We denote this number by the discrete random variable $U_j$, which consists of the combined failures in both $j$ and $j-1$ epochs (i.e., $U_j = X_{j-1} + X_j$). Thus, we express the probability that $u \leq k_1$ components are unavailable during epoch $j$ by the following PMF (cf. Bayes' theorem):

$$Pr(U_j = u) = \sum_{x=0}^{u} Pr(X_{j-1} = x) \binom{k_1 - x}{u - x} h_j^{u-x} (1-h_j)^{k_1-u}.$$

Finally, we estimate the probability that during an epoch $j$ no more than $\bar{k}$ (out of $k_1$) components are unavailable by the following cumulative distributed function:

$$\rho_j(\bar{k},k_1) = Pr(U_j \leq \bar{k}) = \sum_{u=0}^{\bar{k}} Pr(U_j = u).$$

As a result, $\rho_j(q-1,n)$ expresses DARE's reliability. To compute $h_j$, we use the data from Table 4.2 under the assumption that all components are modeled by exponential LDMs.

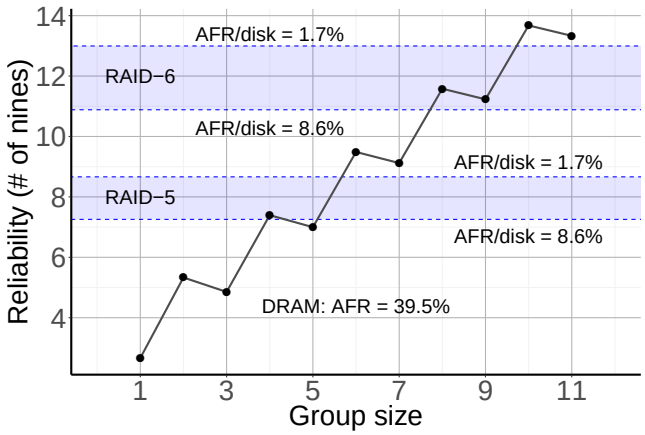



FIGURE 4.7: The reliability of DARE's in-memory approach in comparison to the reliability achieved by disks with RAID technologies. The reliability is estimated over a period of 24 hours.

Figure 4.7 plots the reliability as a function of the group size. Of particular interest is the decrease in reliability when the group size increases from an even to an odd value. This is expected since the group has one more server, but the size of a quorum remains unchanged. Also, Figure 4.7 compares the reliability of our in-memory approach with the one achieved by stable storage; the disk AFRs are according to [134]. We observe that for a group size of 6, DARE can achieve higher reliability than disks with RAID-5 [36], while 10 servers are sufficient to overpass the reliability of disks with RAID-6 [135].

## 4.7 Evaluation

We evaluate the performance of DARE in a practical setting. We use a 12-node InfiniBand cluster; each node has an Intel E5-2609 CPU clocked at 2.40GHz. The cluster is connected with a single switch using a single Mellanox QDR NIC (MT27500) at each node. Moreover, the nodes are running Linux, kernel version 3.12.18. DARE is implemented[2] in C and relies on two libraries: *libibverbs*, an implementation of the RDMA verbs for InfiniBand; and *libev*, a high-performance event loop. Each server runs an instance of DARE; yet, each server is single-threaded. Finally, to compile the code, we used GCC version 4.8.2.

We consider a key-value store as the client state machine: Clients access data through 64-byte keys. Moreover, since clients send requests through UD, the size of a request is limited by the network's MTU (i.e., 4096 bytes). Henceforth, we only state the size of the data associated with a key. The structure of this section is as follows: first, we evaluate both the latency and the throughput of DARE; then, we analyze the throughput for different workloads; finally, we compare the performance of DARE with other algorithms and implementations, such as ZooKeeper [76].

### 4.7.1 Latency

DARE is designed as a high-performance state-machine replication algorithm. Figure 4.8 shows the latency of both write and read requests (gets and puts in the context of a key-value store). In the benchmark, a single client reads and writes objects of varying size to/from a group of five servers; note that the size of a client request is given by the size of the value plus the header that contains the 64-byte key. Each measurement is repeated 1,000 times;

[2]DARE: `http://htor.inf.ethz.ch/sec/dare.tgz`

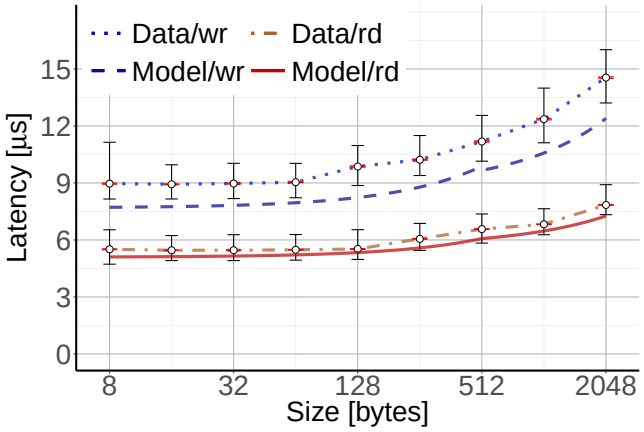
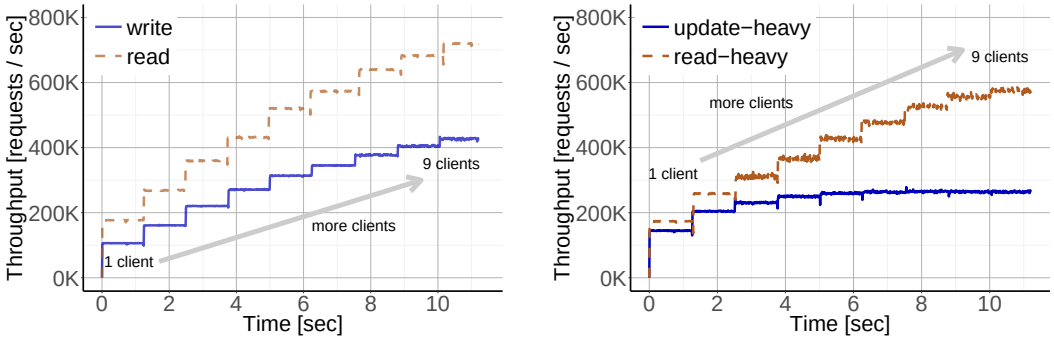



FIGURE 4.8: DARE's latency of both write (/wr) and read (/rd) requests in a group of five servers.




(A) Write and read throughput

(B) Real-world inspired workloads

FIGURE 4.9: Evaluation of DARE's throughput in a group of three servers. (a) The throughput for both write and read requests. (b) The throughput for both read-heavy and update-heavy workloads The results are for 64-byte requests.

the figure reports the 95% confidence interval around the median (small whiskers) and both the 2nd and the 98th percentiles (large whiskers). DARE has a read latency of less than $8\mu s$; the write latency is with $15\mu s$ slightly higher because of the higher complexity of the log replication algorithm. Also, Figure 4.8 evaluates the model described in Section 4.5.2. Of particular interest is the difference between our model and the measured write latency. In practice, the small RDMA overhead of $\approx 0.3\mu s$ (see Table 4.1) implies that a slight computational overhead may cause more than $\lfloor n/2 \rfloor$ servers to go through log replication; as a result, the write latency increases.

### 4.7.2 Throughput

We analyze DARE's throughput in a group of three servers that receives requests from up to nine clients. We calculate the throughput by sampling the number of answered requests in intervals of $10ms$. For 2048-byte requests, DARE achieves a maximum throughput of 760 MiB/s for reads and 470 MiB/s for writes. Furthermore, Figure 4.9a shows how the throughput (for 64-byte requests) increases with the number of clients. The reason for the increase is twofold: (1) DARE handles requests from different clients asynchronously; and (2) DARE batches requests together to reduce the latency. Therefore, with 9 clients, DARE answers over 720,000 read requests per seconds and over 460,000 write requests per second.

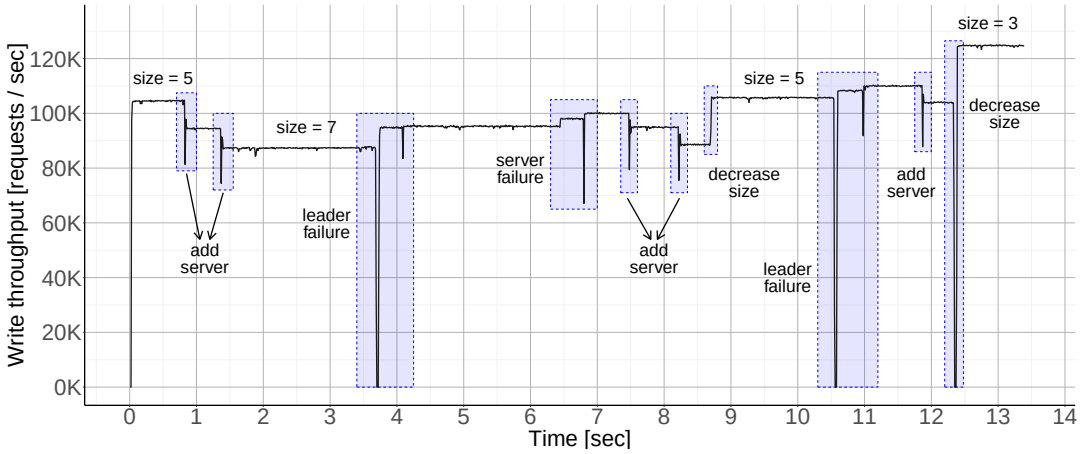



FIGURE 4.10: DARE's write throughput during a series of group reconfiguration scenarios: (1) two servers are joining the group causing an increase in size; (2) the leader fails; (3) a server that is not leader fails; (4) two servers are joining the group; (5) the group size is decreased back to five; (6) the leader fails; (7) a server joins the group; and (8) the group size is decreased, which entails the removal of two servers, one of them being the leader. The results are for 64-byte requests.

**Real-world inspired workloads.** The results of Figure 4.9a are valid for either read-only or write-only client state machines; however, this is usually not the case. Figure 4.9b shows DARE's throughput (for 64-byte requests) when applying real-world inspired workloads to a group of three servers. In particular, we use two workloads: read-heavy and update-heavy [39]. The read-heavy workload consist of 95% read requests; it is representative for applications such as photo tagging. The update-heavy workload consist of 50% writes; it is representative for applications such as an advertisement log that records recent user activities. For read-heavy workload, the throughput slightly fluctuates when more than one client sends requests. This is because DARE ensures linearizable semantics [72]; in particular, the leader cannot answer read requests until it answers all the preceding write requests. Moreover, when read and write requests are interleaved, DARE cannot take advantage of batching (since batches consist of either only read or only write requests). Therefore, for update-heavy workloads, the throughput saturates faster.

**Dynamic membership.** Further, we study the effect of a dynamic group membership on DARE's performance. In particular, Figure 4.10 shows the write throughput (for 64-byte requests) during a series of group reconfiguration scenarios. First, two servers are subsequently joining an already full group causing the size to increase. This implies that more servers are needed for a majority; therefore, the throughput decreases. Also, note that the two joins cause a brief drop in throughput, but no unavailability. Then, the leader fails causing a short period of unavailability (i.e., around 30*ms*) until a new leader is elected; this is shortly followed by a brief drop in performance when the new leader detects and removes the previously failed leader.

Next, a server fails. The leader handles the failure in two steps, both bringing an increase in throughput. First, it stops replicating log entries on the server since its QPs are inaccessible. Second, after a number of failed attempts to send a heartbeat (we use two in our evaluation), the leader removes the server. The removal of the failed server is followed by two other servers joining the group. Note that these joins have similar effects as the previous

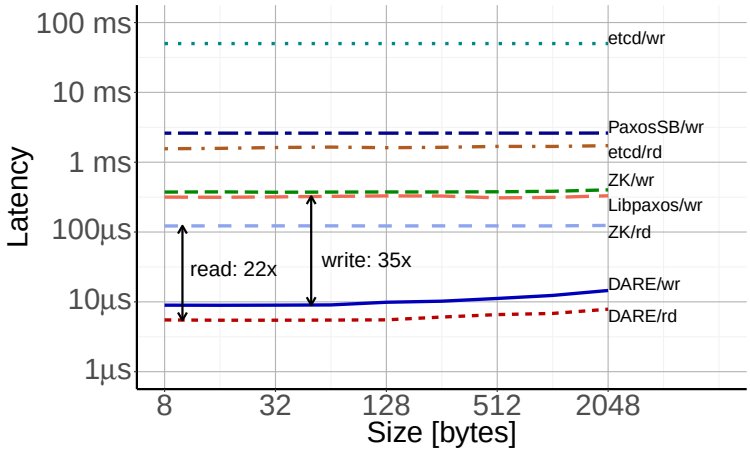



FIGURE 4.11: Evaluation of DARE's latency against other state of the art state-machine replication algorithms and implementations: write (/wr) and read (/rd) latency. A single client sends requests of varying size to a group of five servers.

ones. Once the group is back to a full size, the leader receives a request to decrease the size, which implies an increase in throughput.

In the new configuration, the leader fails again having a similar effect as the previous leader failure. After a new leader is elected, another server joins the group. Finally, the new leader decreases the group size to three. However, this operations entails the removal of two servers, one of them being the leader. Thus, the group is shortly unavailable until a new leader is elected.

### 4.7.3 Comparison to other algorithms

We conclude DARE's evaluation by comparing it with other state of the art state-machine replication algorithms and implementations. In particular, we measure the latency of four applications: ZooKeeper (ZK), a service for coordinating processes of distributed applications [76]; `etcd`[3], a key-value store that uses Raft [130] for reliability; and PaxosSB [88] and Libpaxos [147], both implementations of the Paxos algorithm [92], providing support only for writes.

For each application, we implemented a benchmark that measures the request latency in a similar manner as for DARE—a single client sends requests of varying size to a group of five servers. All applications use TCP/IP for communication; to allow a fair comparison, we utilize TCP/IP over InfiniBand ("IP over IB"). Also, for the applications that rely on stable storage, we utilize a RamDisk (an in-memory filesystem) as storage location. Figure 4.11 shows the request latency of the four applications on our system. For ZooKeeper, we observe a minimal read latency of $\approx 120\mu s$; the put performance depends on the disk performance, and with a RamDisk, it oscillates around $380\mu s$. In the case of `etcd`, a read requests takes around $1.6ms$, while a write request takes almost $50ms$. For both PaxosSB and Libpaxos, we measured only the write latency. While PaxosSB answers a write requests in around $2.6ms$, Libpaxos, with around $320\mu s$, attains a write latency lower than ZooKeeper.

In addition, Figure 4.11 compares our algorithm against all four applications. The latency of DARE is at least 22 times lower for read accesses and 35 times lower for write accesses. Also, we evaluate DARE's write throughput against ZooKeeper; in particular, we set up an

[3] `etcd` version 0.4.6: `https://github.com/coreos/etcd`

experiment were 9 clients send requests to a group of three servers. With a write throughput of $\approx$ 270 MiB/s, ZooKeeper is around 1.7*x* below the performance achieved by DARE. Finally, we compare DARE with the Chubby lock service [27]. Yet, since we cannot evaluate it on our system, we use the latency measurement from the original paper [27]. Chubby achieves read latencies of under 1 ms and write latencies of around 5-10 ms. Thus, DARE's performance is more than two orders of magnitude higher.

**Comparing DARE to APUS.** APUS [161] is a recent Paxos-like algorithm[4] that (similarly to DARE) uses one-sided RDMA operations to enable high-performance state-machine replication and therefore, relies on many of the same principles as DARE. APUS intercepts inbound socket calls, such as `recv()`, invoked by the threads of a program running on the leader. These calls are locally executed in order to retrieve the corresponding data. For each call, the invoking thread appends the retrieved data to the local log by adding a new entry and then it replicates the entry on the remote servers through one-sided RDMA writes. The remote servers poll for new entries; for each received entry, a server checks first whether the leader's view ID matches its own [113] and then it replies to the leader. The replies are performed through one-sided RDMA writes—each log entry has a reply array, with an element for each server (similar to the control data arrays in DARE). In order to complete an inbound socket call, the invoking thread polls on the reply array (associated with its log entry) until it receives positive replies from at least a majority of servers (itself included).

APUS allows multiple (leader) threads to concurrently replicate data; consequently, it can increase performance by simultaneously servicing multiple clients. In contrast, as it is currently implemented, DARE serializes all batches of requests, i.e., the leader starts handling a batch only once the previous one is completed. Moreover, excluding log adjustment (which is executed only once per term), the leader performs three one-sided RDMA writes for each replicated log entry. As a result, having multiple clients sending a mixture of both put and get requests (therefore, hindering batching), leads to requests waiting for consensus to start (see the update-heavy workload in Figure 4.9b). However, in general, the three one-sided RDMA writes necessary for log replication can be pipelined; for instance, the leader can start replicating the subsequent log entry before it completes updating the remote tail pointer. Therefore, DARE can also execute multiple requests in parallel. Finally, the leader election mechanism of APUS is similar to the one used in DARE and Raft.

[4] APUS is based on the algorithm described by Mazieres [113]; yet, despite its name, the *Paxos made practical* paper of Mazieres focuses more on Viewstamped Replication [105].

# Chapter 5

# AllConcur

Many applications require scaling out state-machine replication across a large number of servers, e.g., for some distributed ledgers, the number of replicas can easily be in the range of hundreds. However, most practical state-machine replication approaches were designed mainly for fault tolerance, where scaling out across more than a handful of servers is not necessary. Therefore, these approaches are not well suited for large-scale state-machine replication. In this chapter, we present AllConcur[1], a novel leaderless concurrent atomic broadcast algorithm that distributes the workload evenly among all servers and thus, it enables large-scale high-performance state-machine replication [141]. In AllConcur, all server exchange messages concurrently through a regular, sparse, and resilient overlay network. The overlay network's resiliency provides AllConcur's fault tolerance and can be adapted to system-specific requirements. Therefore, AllConcur trades off fault tolerance against performance. Moreover, AllConcur employs a novel early termination mechanism that reduces the expected number of communication steps significantly. Our evaluation at the end of this chapter shows that our AllConcur implementation over standard sockets-based TCP can handle more than one hundred million requests per second and outperforms standard leader-based approaches, such as Libpaxos [147].

The remainder of this chapter states the problem in Section 5.1; gives an overview of large-scale atomic broadcast and describes the early termination mechanism in Section 5.2; presents the design of AllConcur in Section 5.3; shows AllConcur's correctness through an informal proof in Section 5.4; provides a formal specification of AllConcur and, on the basis of this specification, a formal proof of AllConcur's safety property in Section 5.5; analyzes the AllConcur's performance, with a focus on work performed per server, communication time and storage requirements, in Section 5.6; and evaluates AllConcur's performance in Section 5.7.

Most of the content of this chapter is reproduced from the following papers:

- Marius Poke, Torsten Hoefler, Colin W. Glass. *AllConcur: Leaderless Concurrent Atomic Broadcast (Extended Version)* [140]
- Marius Poke, Colin W. Glass. *Formal Specification and Safety Proof of a Leaderless Concurrent Atomic Broadcast Algorithm* [137]
- Marius Poke, Colin W. Glass. *A Dual Digraph Approach for Leaderless Atomic Broadcast (Extended Version)* [136]

## 5.1 Problem statement

Atomic broadcast is a communication primitive that ensures that all servers receive the same ordered sequence of messages—it provides *total order* (see Section 2.2.1). Consequently,

[1] AllConcur: Algorithm for LeaderLess CONCURrent atomic broadcast

it is a key part of many state-machine replication implementations: State-machine replication uses atomic broadcast as a distributed agreement algorithm for ordering and propagating the requests to all servers. To establish total order, most practical atomic broadcast algorithms rely either on message timestamps that reflect causal ordering (e.g. [87, 68]) or on leader-based approaches (e.g., [92, 84, 105, 130]). These algorithms were designed mainly to provide fault tolerance, where scaling out aims at increasing both reliability and availability and a handful of servers usually suffice [40, 76, 27]. However, we argue that they are not well suited for large-scale deployments of state-machine replication.

Causal ordering [91] allows distributed systems that cannot agree on the exact time (i.e., most real-world systems) to agree on the order in which events happen. The idea behind it is to order events based on what may have caused them. For example, let a server $p_i$ send a message $m_{i,j}$ to a server $p_j$ and let $p_j$, after receiving $m_{i,j}$, send a message $m_{j,k}$ to a server $p_k$. Then, we can say that $p_i$ sending $m_{i,j}$ "caused" $p_k$ to receive $m_{j,k}$. This causal relationship results in a partial order, which can be extended to total order by ordering concurrent events [19]. In general, to provide this partial ordering, algorithms tag messages with timestamps that contain information on every server (e.g., [68]). As a result, an $\mathcal{O}(n)$ overhead is added to every message, which entails the work per A-broadcast message is linear in $n$ (with $n$ denoting the number of servers).

Alternative algorithms, that establish total order through leader-based approaches, usually use a leader-based group (e.g., Paxos [92]) as a reliable sequencer. When atomically broadcasting messages, the $n$ servers send them to the group's leader, which orders them and, for reliability, replicates them within the group. Subsequently, the replicated messages are disseminated to all servers. Figure 5.1a shows an example of a leader-based approach for $n = 9$ servers. Note that it would be inefficient to have all $n$ servers participate in the leader-based group; the group size does not depend on $n$, but only on the reliability of the group members (e.g., in Figure 5.1a, a group size of five is sufficient for a reliability target of 6-nines, estimated over a period of 24 hours and a server $MTTF \approx 2$ years). In straightforward implementations, all the servers interact directly with the leader, for both sending and receiving requests[2]. As a result, the work per A-broadcast message is linear in $n$.

Atomic broadcast algorithms that require linear work per A-broadcast message are not a good fit for implementing state-machine replication at large-scale. This is especially the case when considering that for some applications, the number of servers can easily be in the range of hundreds [10, 17].

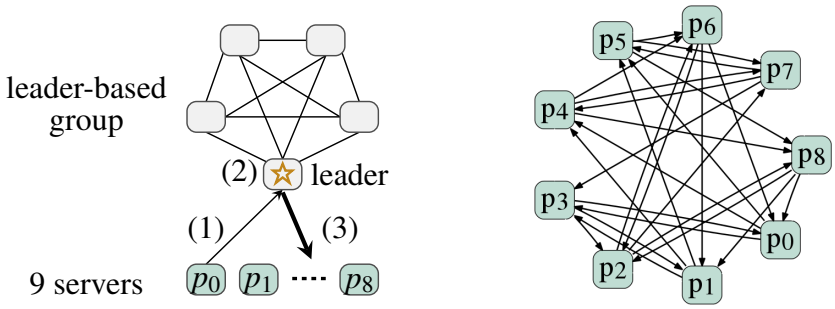
## 5.2 Large-scale atomic broadcast

In this chapter, we focus on atomic broadcast algorithms that require *sublinear* work per broadcast message: We develop a novel algorithm that requires no leader and no timestamps reflecting causal ordering, while at the same time being efficient.

First, we model the communication between servers by an overlay network described by a digraph $G$ (see Section 2.1.1). We assume $G$ has the following properties:

- *sparse*—to keep the work per broadcast message sublinear, $G$'s degree must be considerable smaller than $n$ (i.e., $d(G) \ll n$);
- *resilient*—to disseminate messages while tolerating up to $f$ failures, $G$'s vertex-connectivity must exceed $f$ (i.e., $\kappa(G) > f$);

[2] In principle, the leader can disseminate requests via a tree [75]. However, for fault tolerance, a reliable broadcast algorithm [26] is needed. To the best of our knowledge, there is no leader-based implementation that uses reliable broadcast for dissemination.




(A) Leader-based approach (B) Leaderless approach

FIGURE 5.1: Establishing total order among nine servers: (a) Using a leader-based group; three operations needed per request—(1) send; (2) replicate; and (3) disseminate. (b) Using a sparse, resilient and regular digraph $G_S(9,3)$ with $d(G_S) = \kappa(G_S) = 3$ (see Section 5.6.4 for details).

- *regular*—to balance the workload evenly among the servers, every server must have the same number of successors and predecessors as any other server (i.e., $d(G) = |v^+(G)| = |v^-(G)|, \forall v \in \mathcal{V}(G)$).

As a result, $G$ enables the implementation of reliable broadcast algorithms that require sublinear work per broadcast message. Figure 5.1b shows an example of such a digraph that can tolerate up to two failures, i.e., $G_S(9,3)$ is a regular optimally-connected digraph with degree three [150]. For details on the construction of a $G_S(n,d)$ digraph, see Section 5.6.4.

Second, we assume total order is established through *destinations agreement* [47]. In particular, we require all non-faulty servers to agree on a common set of messages; the messages in this set are then A-delivered in a deterministic order. Consequently, atomic broadcast is transformed into a sequence of *consensus* problems. This results in a *round-based* model: We split the execution of the atomic broadcast algorithm into *asynchronous rounds*, with the constraint that in each round, servers reach consensus on a common set of messages. The round-based model entails a mapping from the set of all possible messages to the set of all rounds: Every message is tagged with a sequence number indicating the round to which it "belongs".

Consensus has a known lower bound: In the worst case, any consensus algorithm that tolerates up to $f$ failures requires $f+1$ communication steps [97, 3]. Intuitively, a server may fail after sending a message to only one other server; this scenario may repeat up to $f$ times, resulting in only one server having the message; this server needs at least one additional communication step to disseminate the message. Note that if $G$ is used for dissemination, consensus requires (in the worst case) $f + D_f(G,f)$ communication steps, where $D_f(G,f)$ denotes $G$'s fault diameter (see Section 2.1.1). Yet, the only necessary and sufficient requirement for safety is for *every non-faulty server to A-deliver messages only once it has all the messages any other non-faulty server has*. To avoid assuming always the worst case, we design an *early termination* scheme that requires each server to track all the messages of a round before A-delivering the agreed upon set of messages.

### 5.2.1 Early termination

Early termination has two parts: (1) deciding whether a message was A-broadcast; and (2) tracking the A-broadcast messages. In general, deciding whether a message was A-broadcast entails waiting for $f + D_f(G,f)$ communication steps (the worst case scenario must be assumed for safety). This essentially eliminates any form of early termination, if at least one

server does not send a message. To avoid scenarios where servers wait for non-existent messages, we assume that, in each round, every server A-broadcasts a message, i.e., the atomic broadcast algorithm is *concurrent*. It is important to mention that this message can also be empty—the server A-broadcasts the information that it has nothing to broadcast.

During each round, every server tracks the A-broadcast messages through the received failure notifications. As an example, we consider a group of nine servers connected through a $G_S(n,d)$ digraph [150] with $d(G_S) = \kappa(G_S) = 3$ (see Figure 5.1b). Also, we consider the following scenario: $p_0$ sends its message $m_0$ only to $p_5$ and then it fails; $p_5$ receives $m_0$, but it fails before sending it further. Therefore, $m_0$ is lost; however, $p_6$ (for example) is not aware of this and to avoid waiting unnecessarily for the worst case bound, it tracks $m_0$ (see Figure 5.2). Server $p_6$ is not directly connected to $p_0$, so it cannot directly detect its failure. Yet, $p_0$'s non-faulty successors eventually detect $p_0$'s failure. Once they suspect $p_0$ to have failed, they stop accepting messages from $p_0$ (see below); also, they R-broadcast notifications of $p_0$'s failure. For example, let $p_4$ be the first to detect $p_0$'s failure and consequently, R-broadcasting a notification. When $p_6$ receives this notification, it first marks $p_0$ as failed and then starts suspecting that, before failing, $p_0$ sent $m_0$ to its successors, i.e., $p_3$ and $p_5$. Note though that $p_6$ does not suspect $p_4$. Had $p_4$ received $m_0$ from $p_0$, then $p_4$ would have relayed $m_0$, which would therefore have arrived to $p_6$ before the subsequent failure notification (due to assumption of FIFO reliable channels in Section 2.1.1). This procedure repeats for the other failure notifications received, e.g., $p_6$ detects $p_5$'s failure directly, $p_8$ detects $p_5$'s failure, $p_3$ detects $p_0$'s failure, and $p_7$ detects $p_5$'s failure. In the end, $p_6$ suspects only $p_0$ and $p_5$ of having $m_0$ and, since both are faulty, $p_6$ stops tracking $m_0$.

Early termination relies on the following proposition:

**Proposition 5.2.1.** *Let $p_i$, $p_j$ and $p_k$ be three servers. Then, $p_i$ receiving a notification of $p_j$'s failure sent by $p_k$ indicates that $p_i$ has all messages $p_k$ received directly from $p_j$.*

This proposition is guaranteed under the assumption of $\mathscr{P}$, which requires the accuracy property to hold, i.e., no false suspicion of failure (see Section 2.2.4).

#### 5.2.1.1 Probabilistic analysis of accuracy

Accuracy is difficult to guarantee in practice: Due to network delays, a server may falsely suspect another server to have failed. However, when the network delays can be approximated as part of a known distribution, accuracy can be probabilistically guaranteed. Let $T$ be a random variable that describes the network delays. Then, we denote by $Pr[T > t]$ the probability that a message delay exceeds a constant $t$.

We propose a failure detector based on a heartbeat mechanism. Every non-faulty server sends heartbeats to its successors in $G$; the heartbeats are sent periodically, with a period $\Delta_{hb}$. Every non-faulty server $p_i$ waits for heartbeats from its predecessors in $G$; if, within a period $\Delta_{to}$, $p_i$ receives no heartbeats from a predecessor $p_j$, it suspects $p_j$ to have failed. Since we assume heartbeat messages are delayed according to a known distribution, we can estimate the probability of accuracy to hold. In particular, we bound the probability of the proposed failure detector to behave indistinguishably from a perfect one.

The interval in which $p_i$ receives two heartbeats from a predecessor $p_j$ is bounded by $\Delta_{hb} + T$. In the interval $\Delta_{to}$, $p_j$ sends $\lfloor \Delta_{to}/\Delta_{hb} \rfloor$ heartbeats to $p_i$. The probability that $p_i$ does not receive the $k$'th heartbeat within the period $\Delta_{to}$ is bounded by $Pr[T > \Delta_{to} - k\Delta_{hb}]$. For $p_i$ to incorrectly suspect $p_j$ to have failed, it has to receive none of the $k$ heartbeats. Moreover, $p_i$ can incorrectly suspect $d(G)$ predecessors; also, there are $n$ servers that can incorrectly suspect their predecessors. Thus, the probability of the accuracy property to hold

is at least

$$(1 - \prod_{k=1}^{\lfloor \Delta_{to}/\Delta_{hb} \rfloor} Pr[T > \Delta_{to} - k\Delta_{hb}])^{n \cdot d(G)}.$$

Increasing both the timeout period and the heartbeat frequency increases the likelihood of accurate failure detection.

#### 5.2.1.2 Impact of eventual accuracy

The requirement of $\mathscr{P}$ (to guarantee Proposition 5.2.1) narrows applicability: In a non-synchronous system, even if message delays can be approximated by a known distribution, a heavy tail may lead to false failure detections and hence, to $\mathscr{P}$ not holding, potentially resulting in inconsistencies—safety is not *deterministically guaranteed*. In order for Proposition 5.2.1 to hold for $\Diamond\mathscr{P}$, a server must ignore any subsequent messages (except failure notifications) it receives from a predecessor it has suspected of having failed. This is equivalent to removing an edge from $G$ and thus, may lead to a disconnected digraph even if $f < \kappa(G)$. If we assume, however, that no more than $\kappa(G) - 1$ servers will be suspected to have failed (falsely or not), then $G$ will always remain connected. This assumption can be practical, especially for deployments within a single datacenter, with a conservative vertex-connectivity chosen for $G$.

**Forward-backward mechanism.** Instead, in order to allow deployments across multiple datacenters, we consider a *primary partition* membership approach [47]: Only servers from the *surviving partition*—a strongly connected component that contains at least a majority of servers—are allowed to make progress and A-deliver messages. To decide whether they are part of the surviving partition, the servers use a mechanism based on Kosaraju's algorithm to find strongly connected components [4, Chapter 6]: Once a server $p_i$ completes tracking all messages A-broadcast in a round, it R-broadcasts two control messages: (1) a forward message $\langle FWD, p_i \rangle$; and (2) a backward message $\langle BWD, p_i \rangle$. The backward message is R-broadcast using $G$'s transpose. In order to A-deliver the agreed upon set of messages, each server must receive both forward and backward messages from at least a majority of servers (itself included). Intuitively, a $\langle FWD, p_j \rangle$ message received by $p_i$ indicates that when $p_j$ decided on its set of messages, there was at least one path from $p_j$ to $p_i$; thus, $p_i$ knows of all the messages known by $p_j$. Similarly, a $\langle BWD, p_j \rangle$ message indicates that $p_j$ knows of all the messages known by $p_i$. Therefore, when a server A-delivers the messages of a round, it knows that at least a majority of the servers (including itself) A-deliver the same messages. Henceforth, we refer to this as the *forward-backward mechanism*.

## 5.3 The design of the AllConcur algorithm

AllConcur is a completely decentralized, round-based atomic broadcast algorithm that connects servers through an overlay network described by a digraph $G$, i.e., servers receive messages from their predecessors and send messages to their successors. In a nutshell, in every round $r$, every non-faulty server performs three tasks: (1) it A-broadcasts a single (possibly empty) message; (2) it tracks the messages A-broadcast in round $r$ using the early termination mechanism described in Section 5.2.1; and (3) once done with tracking, it A-delivers—in a deterministic order—all the messages A-broadcast in $r$ that it received. Note that A-delivering messages in a deterministic order entails that A-broadcast messages do not have to be received in the same order.

AllConcur tolerates up to $f$ faulty servers; consequently, it assumes that $G$'s vertex-connectivity exceeds the maximum number of failures, i.e., $k(G) > f$. AllConcur relies

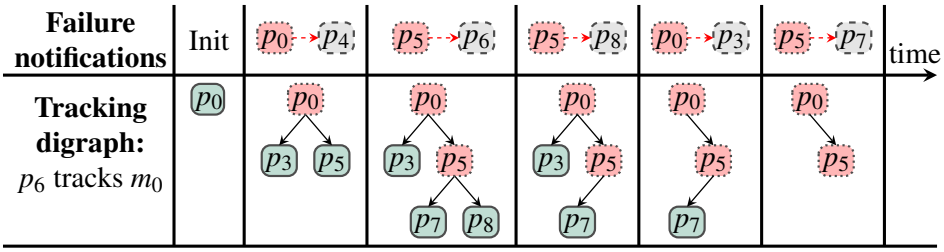



FIGURE 5.2: Example of message tracking—server $p_6$ tracking $p_0$'s message $m_0$ within a $G_S(9,3)$ digraph. Dotted red nodes indicate failed servers. Dashed gray nodes indicate servers from which $p_6$ received failures notifications (i.e, dashed red edges). Solid green nodes indicate servers suspected to have $m_0$.

on failure detectors (see Section 2.2.4): When a server fails, its successors detect the failure and R-broadcast failure notifications to the other servers; these failure notifications enable the early termination mechanism. Algorithm 5.1 shows the actions perform by a server $p_i$ during an AllConcur round; also, it outlines the steps necessary for transitioning to the subsequent round. For ease of presentation, Algorithm 5.1 omits the forward-backward mechanism required by $\Diamond\mathcal{P}$ (see Section 5.2); therefore, Algorithm 5.1 assumes $\mathcal{P}$.

In AllConcur, we distinguish between A-broadcast messages and control messages, such as failure notifications. We use $\langle BCAST, m_j\rangle$ to denote a message A-broadcast by $p_j$ and $\langle FAIL, p_j, p_k \in p_j^+(G)\rangle$ to denote a failure notification, R-broadcast by $p_k$, indicating $p_k$'s suspicion that its predecessor $p_j$ has failed. Note that if $p_i$ receives the notification and $p_k = p_i$, then it originated from $p_i$'s own failure detector. Algorithm 5.1 starts when at least one server A-broadcasts a message (line 1). Every server sends a message of its own, at the latest as a reaction upon receiving a message.

**Round completion.** To complete a round, AllConcur adopts the early termination mechanism described in Section 5.2.1. To track the messages that were A-broadcast during a round, each server $p_i$ stores an array $\mathbf{g_i}$ of $n$ digraphs, one for each server $p_* \in \mathcal{V}(G)$; we refer to these as *tracking digraphs*. The tracking digraph $\mathbf{g_i}[p_*]$ is a representation of $p_i$'s suspicion of $m_*$'s whereabouts (where $m_*$ is the message A-broadcast by $p_*$). The vertices of $\mathbf{g_i}[p_*]$ indicate the servers suspected (by $p_i$) of having $m_*$; in other words, these are the servers from which $p_i$ expects to get $m_*$. The edges of $\mathbf{g_i}[p_*]$ indicate the paths on which $m_*$ is suspected (by $p_i$) of having been transmitted; for instance, the edge $(p_j, p_k) \in \mathcal{E}(\mathbf{g_i}[p_*])$ indicates $p_i$'s suspicion that $p_k$ received $m_*$ directly from $p_j$.

Initially, $\mathcal{V}(\mathbf{g_i}[p_*]) = \{p_*\}, \forall p_* \neq p_i$ and $\mathcal{V}(\mathbf{g_i}[p_i]) = \emptyset$. Intuitively, this means that $p_i$ initially expects to get every message (expect its own) from the server that A-broadcast it. Also, notice that if $p_i$'s tracking digraph for $p_*$ has no vertices (i.e., $\mathcal{V}(\mathbf{g_i}[p_*]) = \emptyset$), then $p_i$ does not expect to receive $m_*$. This may be because either $p_i$ already receive $m_*$ or it is certain that nobody will receive it (i.e., $m_*$ is lost). Therefore, once all its tracking digraphs are empty, $p_i$ A-delivers (in a deterministic order) all received messages (line 7).

Figure 5.2 illustrates the changes to the tracking digraphs based on the $G_S(9,3)$ example in Section 5.2.1. For clarity, we show only how $p_6$ tracks $m_0$ by updating $\mathbf{g_6}[p_0]$. Initially, $\mathbf{g_6}[p_0]$ has only $p_0$ as vertex, i.e., only $p_0$ is suspected to have $m_0$. Then, $p_6$ receives a sequence of five failure notifications (indicated by dashed red edges in Figure 5.2):

- $\langle FAIL, p_0, p_4\rangle$ As a result, $p_6$ is certain $p_4$ has not received $m_0$ directly from $p_0$ (cf. Proposition 5.2.1). However, $p_0$ may have sent $m_0$ to its other successors, i.e., $p_3$ and $p_5$, and thus, $p_6$ adds them to $\mathbf{g_6}[p_0]$. Moreover, $p_6$ stops expecting $p_0$ to sent $m_0$, i.e., it marks it as failed (indicated by dotted red nodes in Figure 5.2).

**Algorithm 5.1:** The AllConcur algorithm under the assumption of both $\kappa(G) > f$ and $\mathscr{P}$; code executed by server $p_i$; see Table 2.1 for digraph notations.

**Input:** $n$; $f$; $G$; $m_i$; $M_i \leftarrow \emptyset$; $F_i \leftarrow \emptyset$; $\mathscr{V}(\mathbf{g_i}[p_i]) \leftarrow \emptyset$; $\mathscr{V}(\mathbf{g_i}[p_j]) \leftarrow \{p_j\}, \forall j \neq i$

1 **def** ***A-broadcast***$(m_i)$**:**
2 **send** $\langle BCAST, m_i \rangle$ **to** $p_i^+(G)$
3 $M_i \leftarrow M_i \cup \{m_i\}$
4 TryToCompleteRound()
5 **def** TryToCompleteRound()**:**
6 **if** $\mathscr{V}(\mathbf{g_i}[p]) = \emptyset, \forall p$ **then**
7 **foreach** $m \in$ `sort(`$M_i$`)` **do** *A-deliver*$(m)$ {A-deliver messages}
/* preparing for next round */
8 **foreach** *server* $p_*$ **do**
9 **if** $m_* \notin M_i$ **then** $\mathscr{V}(G) \leftarrow \mathscr{V}(G) \setminus \{p_*\}$ {remove servers}
10 **foreach** $(p, p_s) \in F_i$ *s.t.* $p \in \mathscr{V}(G)$ **do**
11 **send** $\langle FAIL, p, p_s \rangle$ **to** $p_i^+(G)$ {resend failures}
12 **receive** $\langle BCAST, m_j \rangle$**:**
13 **if** $m_i \notin M_i$ **then** *A-broadcast*$(m_i)$
14 $M_i \leftarrow M_i \cup \{m_j\}$
15 **for** $m \in M_i$ *not already sent* **do**
16 **send** $\langle BCAST, m \rangle$ **to** $p_i^+(G)$ {disseminate messages}
17 $\mathscr{V}(\mathbf{g_i}[p_j]) \leftarrow \emptyset$
18 TryToCompleteRound()
19 **receive** $\langle FAIL, p_j, p_k \in p_j^+(G) \rangle$**:**
/* if $k = i$ then notification from local $FD$ */
20 **send** $\langle FAIL, p_j, p_k \rangle$ **to** $p_i^+(G)$ {disseminate failures}
21 $F_i \leftarrow F_i \cup \{(p_j, p_k)\}$
22 **foreach** *server* $p_*$ **do**
23 **if** $p_j \notin \mathscr{V}(\mathbf{g_i}[p_*])$ **then** *continue*
24 **if** $p_j^+(\mathbf{g_i}[p_*]) = \emptyset$ **then**
/* maybe $p_j$ sent $m_*$ to someone in $p_j^+(G)$ */
25 $Q \leftarrow \{(p_j, p) : p \in p_j^+(G) \setminus \{p_k\}\}$ {FIFO queue}
26 **foreach** $(p_p, p) \in Q$ **do**
27 $Q \leftarrow Q \setminus \{(p_p, p)\}$
28 **if** $p \notin \mathscr{V}(\mathbf{g_i}[p_*])$ **then**
29 $\mathscr{V}(\mathbf{g_i}[p_*]) \leftarrow \mathscr{V}(\mathbf{g_i}[p_*]) \cup \{p\}$
30 **if** $\exists (p, *) \in F_i$ **then** $Q \leftarrow Q \cup \{(p, p_s) : p_s \in p^+(G)\} \setminus F_i$
31 $\mathscr{E}(\mathbf{g_i}[p_*]) \leftarrow \mathscr{E}(\mathbf{g_i}[p_*]) \cup \{(p_p, p)\}$
32 **else if** $p_k \in p_j^+(\mathbf{g_i}[p_*])$ **then**
/* $p_k$ has not received $m_*$ from $p_j$ */
33 $\mathscr{E}(\mathbf{g_i}[p_*]) \leftarrow \mathscr{E}(\mathbf{g_i}[p_*]) \setminus \{(p_j, p_k)\}$
34 **foreach** $p \in \mathscr{V}(\mathbf{g_i}[p_*])$ *s.t.* $\nexists \pi_{p_*,p}$ *in* $\mathbf{g_i}[p_*]$ **do**
35 $\mathscr{V}(\mathbf{g_i}[p_*]) \leftarrow \mathscr{V}(\mathbf{g_i}[p_*]) \setminus \{p\}$ {prune: no input}
36 **if** $\forall p \in \mathscr{V}(\mathbf{g_i}[p_*]), (p, *) \in F_i$ **then**
37 $\mathscr{V}(\mathbf{g_i}[p_*]) \leftarrow \emptyset$ {prune: no dissemination}
38 TryToCompleteRound()

- $\langle FAIL, p_5, p_6 \rangle$ As a result, $p_6$ stops expecting $p_5$ to sent $m_0$. Note that, due to the previous failure notification, $p_5$ was already suspected to have $m_0$. Moreover, $p_6$ suspects that, before failing, $p_5$ sent $m_0$ to its successors; therefore, it adds both $p_7$ and $p_8$ to $\mathbf{g_6}[p_0]$.
- $\langle FAIL, p_5, p_8 \rangle$ As a result, $p_6$ stops suspecting that $p_8$ received $m_0$ directly from $p_5$; consequently, it removes the edge $(p_5, p_8)$ from $\mathbf{g_6}[p_0]$. Moreover, since $p_8$ is disconnected from the root (i.e., $p_0$), $p_6$ removes it from $\mathbf{g_6}[p_0]$. Intuitively, $p_6$ stops suspecting that $p_8$ has $m_0$.
- $\langle FAIL, p_0, p_3 \rangle$ As a result, $p_6$ removes $p_3$ from $\mathbf{g_6}[p_0]$ (similar as before).
- $\langle FAIL, p_5, p_7 \rangle$ As a result, $p_6$ removes $p_7$ from $\mathbf{g_6}[p_0]$ (similar as before).

In the end, $\mathbf{g_6}[p_0]$ contains only two vertices, i.e., $p_0$ and $p_5$, both marked as failed. Therefore, $p_6$ removes them both and stops waiting to receive $m_0$.

**Receiving A-broadcast messages.** When receiving $\langle BCAST, m_j \rangle$, a message A-broadcast by $p_j$ (line 12), server $p_i$ adds it to the set $M_i$ of known messages. Also, it A-broadcasts its own message $m_i$, in case it did not do so before. Then, it continues the dissemination of each known message through the network—$p_i$ sends all unique messages it has not already sent to its successors. Finally, $p_i$ removes all the vertices from the $\mathbf{g_i}[p_j]$ digraph; then, it checks whether the termination conditions are fulfilled.

**Receiving failure notifications.** When receiving $\langle FAIL, p_j, p_k \rangle$, a notification, R-broadcast by $p_k$, indicating $p_k$'s suspicion that $p_j$ has failed (line 19), $p_i$ disseminates it further. Then, it adds a tuple $(p_j, p_k)$ to the set $F_i$ of received failure notifications. Finally, it updates the tracking digraphs that contain $p_j$ as a vertex.

We distinguish between two cases, depending on whether this is the first notification of $p_j$'s failure received by $p_i$. If it is the first, $p_i$ updates all $\mathbf{g_i}[p_*]$ containing $p_j$ as a vertex by adding $p_j$'s successors (from $G$) together with the corresponding edges. The rationale is that $p_j$ may have sent $m_*$ to his successors, who are now in possession of it. However, there are two exceptions. First, $p_k$ could not have received $m_*$ directly from $p_j$ (cf. Proposition 5.2.1). Second, if a successor $p \notin \mathcal{V}(\mathbf{g_i}[p_*])$ is added, which is already known to have failed, it may have already received $m_*$ and sent it further. Hence, the successors of $p$ could be in possession of $m_*$ and are added to $\mathbf{g_i}[p_*]$ in the same way as described above (line 30).

If $p_i$ is already aware of $p_j$'s failure (i.e., the above process already took place), the new failure notification informs $p_i$, that $p_k$ (the owner of the notification) has not received $m_*$ from $p_j$—because $p_k$ would have sent it before sending the failure notification. Thus, the edge $(p_j, p_k)$ can be removed from $\mathbf{g_i}[p_*]$ (line 32). Removing edges from $\mathbf{g_i}[p_*]$ may disconnect some vertices from the root $p_*$. Intuitively, this means that those servers are no longer suspected to have $m_*$. Consequently, after removing an edge, $p_i$ removes every disconnected server $p$, i.e., $\nexists\, \pi_{p_*,p}$ in $\mathbf{g_i}[p_*]$ (line 34).

In the end, if $\mathbf{g_i}[p_*]$ contains only servers already marked as failed, $p_i$ prunes it entirely—no non-faulty server has $m_*$ (line 36).

**Iterating AllConcur.** Executing subsequent rounds of AllConcur requires the correct handling of failures. Since different servers may end and begin rounds at different times, AllConcur employs a consistent mechanism of tagging servers as failed: At the end of each round, all servers whose messages were not A-delivered are tagged as failed by all the other servers (line 8). As every non-faulty server agrees on the A-delivered messages, this ensures

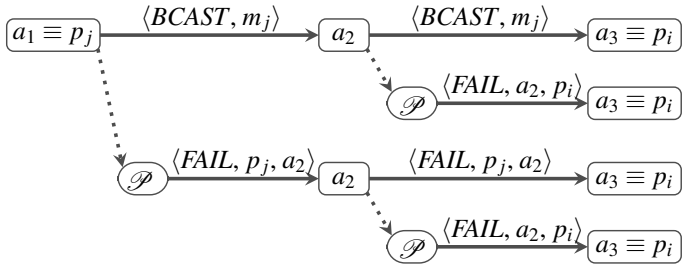



FIGURE 5.3: Possible messages along a three-server path. Dotted arrows indicate failure detection.

a consistent view of failed servers. In the next round, every server resends the failure notifications, except those of servers already tagged as failed (line 10). Thus, only the tags and the necessary resends need to be carried over from the previous round.

**Initial bootstrap and dynamic membership.** To bootstrap AllConcur, we require a centralized service, such as ZooKeeper [76]: The system must decide on the initial configuration—the identity of the $n$ servers, the fault tolerance $f$ and the digraph $G$. Once AllConcur starts, any further reconfigurations are agreed upon via atomic broadcast. This includes topology reconfigurations and membership changes, i.e., servers leaving and joining the system. In contrast to leader-based approaches, where such changes may necessitate a leader election, in AllConcur, dynamic membership is handled directly by the algorithm.

## 5.4 Informal proof of correctness

In this section, we prove that AllConcur, as described in Algorithm 5.1, solves the atomic broadcast problem: We show that, under the assumption of both $\kappa(G) > f$ and $\mathscr{P}$, the four properties on non-uniform atomic broadcast—validity, agreement, integrity and total order—are guaranteed (see Section 2.2.1). Then, we extend the proof to show that AllConcur solves the *uniform* atomic broadcast problem. Finally, we discuss the implications of assuming $\Diamond\mathscr{P}$.

The integrity property clearly holds: Every server $p_i$ executes *A-deliver*() only once for each message in its set $M_i$, which contains only messages A-broadcast by some servers. To show that the validity property holds, it is sufficient to prove the *round termination* property (see Lemma 5.4.4). To show that both agreement and total order hold, it is sufficient to prove the *set agreement* property (see Lemma 5.4.5). To prove round termination and set agreement, we introduce the following lemmas:

**Lemma 5.4.1.** *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be another server; let $\pi_{p_j,p_i} = (a_1, \ldots, a_\lambda)$ be a path (in digraph $G$) from $p_j$ to $p_i$. If $p_j$ knows a message $m$ (either its own or received), then, $p_i$ eventually receives either $\langle BCAST, m \rangle$ or $\langle FAIL, a_k, a_{k+1} \rangle$ with $1 \leq k < \lambda$.*

*Proof.* Server $p_j$ can either fail or send $m$ to $a_2$; note that $a_1 \equiv p_j$ and $a_d \equiv p_i$ (see Figure 5.3). Further, for each inner server $a_k \in \pi_{p_j,p_i}, 1 < k < \lambda$, we distinguish three scenarios: (1) $a_k$ fails; (2) $a_k$ detects the failure of its predecessor on the path; or (3) $a_k$ further sends the message received from its predecessor on the path. The message can be either $\langle BCAST, m \rangle$ or $\langle FAIL, a_l, a_{l+1} \rangle$ with $1 \leq l < k$. Thus, $p_i$ eventually receives either $\langle BCAST, m \rangle$ or $\langle FAIL, a_k, a_{k+1} \rangle$ with $1 \leq k < \lambda$. Figure 5.3 shows, in a tree-like fashion, what messages can be transmitted along a three-server path. □

**Lemma 5.4.2.** *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be another server. If $p_j$ knows a message m (either its own or received), then $p_i$ eventually receives either the message m or a notification of $p_j$'s failure.*

*Proof.* If $p_i$ receives $m$, then the proof is done. Otherwise, we assume $p_i$ does not receive a notification of $p_j$'s failure either. Due to $G$'s vertex-connectivity, there are at least $k(G)$ vertex-disjoint paths $\pi_{p_j,p_i}$. For each of these paths, $p_i$ must receive notifications of some inner vertex failures (cf. Lemma 5.4.1). Since the paths are vertex-disjoint, each notification indicates a different server failure. However, this contradicts the assumption that $f < k(G)$. □

**Corollary 5.4.2.1.** *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be another server. If $p_j$ receives a message, then $p_i$ eventually receives either the same message or a notification of $p_j$'s failure.*

**Lemma 5.4.3.** *Let $p_i$ be a server; let $\mathbf{g_i}[p_j]$ be a tracking digraph that can no longer be pruned. If $\mathcal{E}(\mathbf{g_i}[p_j]) \neq \emptyset$, then $p_i$ eventually removes an edge from $\mathcal{E}(\mathbf{g_i}[p_j])$.*

*Proof.* We assume that $p_i$ removes no edge from $\mathcal{E}(\mathbf{g_i}[p_j])$. Clearly, the following statements are true: (1) $\mathcal{V}(\mathbf{g_i}[p_j]) \neq \emptyset$ (since $\mathcal{E}(\mathbf{g_i}[p_j]) \neq \emptyset$); (2) $p_j \in \mathcal{V}(\mathbf{g_i}[p_j])$ (since $\mathbf{g_i}[p_j]$ can no longer be pruned); and (3) $p_j$ is known to have failed (since $\mathcal{V}(\mathbf{g_i}[p_j]) \neq \{p_j\}$). Let $p \in \mathcal{V}(\mathbf{g_i}[p_j])$ be a server such that $p_i$ receives no notification of $p$'s failure. The reason $p$ exists is twofold: (1) the maximum number of failures is bounded; and (2) $\mathbf{g_i}[p_j]$ can no longer be pruned (line 36). Then, we can construct a path $\pi_{p_j,p} = (a_1, \ldots, a_\lambda)$ in $\mathbf{g_i}[p_j]$ such that every server along the path, except for $p$, is known to have failed (line 34). Eventually, $p$ receives either $\langle BCAST, m_j \rangle$ or $\langle FAIL, a_k, a_{k+1} \rangle$ with $1 \leq k < \lambda$ (cf. Lemma 5.4.1). Since $p_i$ receives no notification of $p$'s failure, the message received by $p$ eventually arrives at $p_i$ (cf. Corollary 5.4.2.1). On the one hand, if $p_i$ receives $\langle BCAST, m_j \rangle$, then all edges are removed from $\mathcal{E}(\mathbf{g_i}[p_j])$; this leads to a contradiction. On the other hand, if $p_i$ receives $\langle FAIL, a_k, a_{k+1} \rangle$, then the edge $(a_k, a_{k+1})$ is removed from $\mathcal{E}(\mathbf{g_i}[p_j])$ (line 33); this also leads to a contradiction. □

**Lemma 5.4.4.** *(Round termination) Let r be a round started by some server; let $p_i$ be a non-faulty server. Then, $p_i$ eventually completes round r.*

*Proof.* If $\mathcal{V}(\mathbf{g_i}[p]) = \emptyset, \forall p$, then the proof is done (line 6). We assume $\exists p_j$ such that $\mathcal{V}(\mathbf{g_i}[p_j]) \neq \emptyset$ and $\mathbf{g_i}[p_j]$ can no longer be pruned. Clearly, $p_j \in \mathcal{V}(\mathbf{g_i}[p_j])$. Server $p_i$ receives either $m_j$ or a notification of $p_j$'s failure (cf. Lemma 5.4.2). If $p_i$ receives $m_j$, then all servers are removed from $\mathcal{V}(\mathbf{g_i}[p_j])$, which contradicts the assumption that $\mathcal{V}(\mathbf{g_i}[p_j]) \neq \emptyset$. We assume $p_i$ receives a notification of $p_j$'s failure; then, $p_j^+(\mathbf{g_i}[p_j]) \neq \emptyset$ (since $\mathbf{g_i}[p_j]$ can no longer be pruned); also, $\mathcal{E}(\mathbf{g_i}[p_j]) \neq \emptyset$. By repeatedly applying the result of Lemma 5.4.3, it results that $p_i$ eventually removes all edges from $\mathbf{g_i}[p_j]$. As a result, $\mathbf{g_i}[p_j]$ is eventually completely pruned, which contradicts the assumption that $\mathcal{V}(\mathbf{g_i}[p_j]) \neq \emptyset$. □

**Lemma 5.4.5.** *(Set agreement) Let $p_i$ and $p_j$ be two non-faulty servers that complete round r. Then, both servers agree on the same set of messages, i.e., $M_i^r = M_j^r$.*

*Proof.* It is sufficient to show that if $m_* \in M_i^r$ when $p_i$ completes round $r$, then also $m_* \in M_j^r$ when $p_j$ completes $r$. We assume that $p_j$ does not receive $m_*$. Let $\pi_{p_*,p_i} = (a_1, \ldots, a_\lambda)$ be one of the paths (in $G$) on which $m_*$ arrives at $p_i$. Let $k$, $1 \leq k \leq \lambda$ the smallest index such that $p_j$ receives no notification of $a_k$'s failure. The existence of $a_k$ is given by the existence of $p_i$, a server that is both non-faulty and on $\pi_{p_*,p_i}$. Then, according to the construction of the tracking digraphs, $a_k \in \mathcal{V}(\mathbf{g_j}[p_*])$. Since $p_j$ completes round $r$, it eventually removes $a_k$ from $\mathbf{g_j}[p_*]$. In general, $p_j$ can remove $a_k$ when it receives either $m_*$ or a notification of $a_k$'s

failure; yet, both alternatives lead to contradictions. In addition, for $k > 1$, $p_j$ can remove $a_k$ when there is no path $\pi_{p_*,a_k}$ in $\mathbf{g_j}[p_*]$. This requires $p_j$ to remove an edge on the $(a_1,\ldots,a_k)$ path. Thus, $p_j$ received $\langle FAIL, a_l, a_{l+1}\rangle$ with $1 \leq l < k$. Since $a_{l+1}$ is on $\pi_{p_*,p_i}$, it clearly received $m_*$ from $a_l$. As a result, before receiving $\langle FAIL, a_l, a_{l+1}\rangle$, $p_j$ must have received $\langle BCAST, m_*\rangle$, which leads to a contradiction. □

**Uniformity.** According to the proof above, AllConcur solves the non-uniform atomic broadcast problem (under the assumption of both $\kappa(G) > f$ and $\mathscr{P}$). In non-uniform atomic broadcast though, neither agreement nor total order holds for non-faulty servers. As a consequence, it is not necessary for all non-faulty servers to A-deliver the messages A-delivered by faulty servers. This may lead to inconsistencies in some applications, such as a persistent database (i.e., as a reaction of A-delivering a message, a faulty server issues a write on disk). Uniformity (see Section 2.2.1) can facilitate the development of such applications. In the context of AllConcur, proving uniformity translates into showing that *uniform set agreement* holds. Due to *message stability*—before a server A-delivers a message, it sends it to all its $d(G) > f$ successors—we can extend the proof of Lemma 5.4.5 to apply to all server, including faulty ones.

**Lemma 5.4.6.** *(Uniform set agreement) Let $p_i$ and $p_j$ be any two servers that complete round $r$. Then, both servers agree on the same set of messages, i.e., $M_i^r = M_j^r$.*

*Proof.* It is sufficient to show that if $m_* \in M_i^r$ when $p_i$ completes round $r$, then also $m_* \in M_j^r$ when $p_j$ completes $r$. We assume that $p_j$ does not receive $m_*$. Let $\pi_{p_*,p_i} = (a_1,\ldots,a_\lambda)$ be one of the paths (in $G$) on which $m_*$ arrives at $p_i$. Depending on whether $p_i$ is faulty, we distinguish two cases. If $p_i$ is non-faulty, the proof follows directly from the proof of Lemma 5.4.5. Therefore, we assume $p_i$ became faulty, but after completing round $r$.

We extend $\pi_{p_*,p_i}$ by appending to it $p_s$, one of $p_i$'s non-faulty successors; the existence of $p_s$ is ensured by the assumption that $\kappa(G) > f$ (since $d(G) \geq \kappa(G)$). As a result, we obtain the path $\pi_{p_*,p_s} = (a_1,\ldots,a_\lambda,a_{\lambda+1})$. Let $k$, $1 \leq k \leq \lambda+1$ be the smallest index such that $p_j$ receives no notification of $a_k$'s failure. The existence of $a_k$ is given by the existence of $p_s$, a server that is both non-faulty and on $\pi_{p_*,p_s}$. Then, according to the construction of the tracking digraphs, $a_k \in \mathscr{V}(\mathbf{g_j}[p_*])$. Since $p_j$ completes round $r$, it eventually removes $a_k$ from $\mathbf{g_j}[p_*]$. Similar to the proof of Lemma 5.4.5, removing $a_k$ as a result of receiving either $m_*$ or a notification of $a_k$'s failure, leads to contradictions. Therefore, $p_j$ removed $a_k$ (with $k > 1$) when there were no paths $\pi_{p_*,a_k}$ in $\mathbf{g_j}[p_*]$. This entails that $p_j$ removed an edge $(a_l,a_{l+1})$ on the $(a_1,\ldots,a_k)$ path after receiving $\langle FAIL, a_l, a_{l+1}\rangle$ with $1 \leq l < k$. If $a_{l+1} \neq p_s$ (i.e., $k \leq \lambda$), then it clearly received $m_*$ from $a_l$ (since is on $\pi_{p_*,p_i}$). Otherwise, due to message stability, $a_{l+1}$ receives $m_*$ from $p_i$. As a result, before receiving $\langle FAIL, a_l, a_{l+1}\rangle$, $p_j$ must have received $\langle BCAST, m_*\rangle$, which leads to a contradiction. □

**Eventual accuracy.** AllConcur's proof of correctness relies on the assumption of $\mathscr{P}$ (i.e., no false suspicion of failure). When using $\Diamond\mathscr{P}$ though, servers may falsely suspect each other to have failed. As a result, the digraph $G$ may become disconnected (even under the assumption of $\kappa(G) > f$), which may consequently lead to servers from different strongly connected components agreeing on different sets of messages. To guarantee the set agreement property under the assumption of $\Diamond\mathscr{P}$, AllConcur employs the forward-backward mechanism described in Section 5.2.1. The mechanism ensures that every server that A-delivers messages is part of the surviving partition—a strongly connected component that contains at least a majority of servers.

The servers that are not part of the surviving partition cannot make progress. In order to guarantee round termination (see Lemma 5.4.4), non-terminating servers need to be

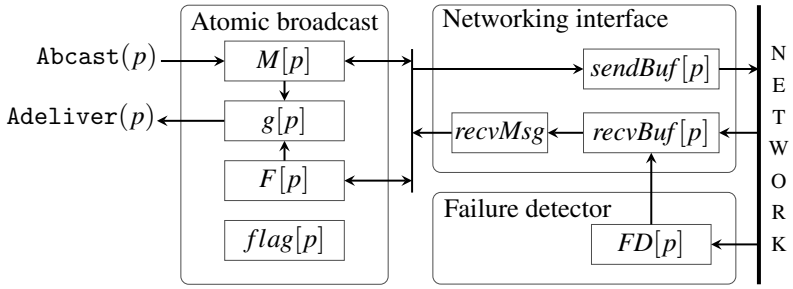



FIGURE 5.4: AllConcur system design from the perspective of a server $p$. Boxes depict the state (i.e., the variables), while arrows indicate the flow of information.

eventually removed from the system and consequently, be considered as faulty (i.e., process controlled crash). In practice, these servers could restart after a certain period of inactivity and then, try to rejoin the system, by sending a membership request to one of the non-faulty servers. Notice that this also includes scenarios where there is no surviving partition, which entails that AllConcur must be stopped and started again from the latest A-delivered round.

## 5.5 Formal specification and safety proof

In this section, we provide a formal specification of AllConcur during a single round and under the assumption of both $\kappa(G) > f$ and $\mathscr{P}$. Then, on the basis of this specification, we provide a mechanically verifiable proof of AllConcur's safety; the proof follows the steps of the informal proof described in Section 5.4.

### 5.5.1 Design specification

We use TLA+ [94] to provide a formal design specification of AllConcur[3]. In addition to the number of servers $n$ and the fault tolerance $f$, throughout this section, we define $S$ to be the set of servers and $E$ the set of directed edges describing the overlay network (i.e., $S = \mathscr{V}(G)$ and $E = \mathscr{E}(G)$); clearly, $E \subseteq S \times S$. To define a digraph, we use the *Graphs* module [94]. For example, $G$ is defined as a record whose *node* field is $S$ and *edge* field is $E$.

We split the design of AllConcur into three modules: (1) an atomic broadcast (AB) module; (2) a networking (NET) module; and (3) a failure detector (FD) module. Figure 5.4 illustrates the three modules together with the variables describing AllConcur's state; moreover, the arrows indicate the flow of information, e.g., receiving a failure notification updates the set $F[p]$, which leads to the update of the tracking digraphs in $g[p]$ (see Section 5.5.1.1). The AB module is the core of AllConcur's design (§ 5.5.1.1). It exposes two interfaces at every server $p \in S$: the input interface `Abcast`$(p)$, to A-broadcast a message; and the output interface `Adeliver`$(p)$, to A-deliver all known A-broadcast messages. The AB module relies on the other two modules for interactions between servers: the NET module (§ 5.5.1.3) provides an interface for asynchronous message-based communication; and the FD module (§ 5.5.1.4) provides information about faulty servers [34].

#### 5.5.1.1 The atomic broadcast module

Let $p \in S$ be any server. Then, the state of the AB module is described by the values of four variables (see Figure 5.4):

[3]The TLA+ specification is available at `https://github.com/mpoke/allconcur`

| | *M* | *g* | *F* | *nf* | *ab* | *done* | *sendBuf* | *recvBuf* | *recvMsg* | *FD* |
|---|---|---|---|---|---|---|---|---|---|---|
| `Abcast`$(p)$ | $[p]$ | $[p][p]$ | - | - | $[p]$ | - | - | - | - | - |
| `Adeliver`$(p)$ | - | - | - | - | - | $[p]$ | - | - | - | - |
| `RecvBCAST`$(p,m)$ | $[p]$ | $[p][m.o]$ | - | - | - | - | - | - | - | - |
| `RecvFAIL`$(p,m)$ | - | $[p]$ | $[p][m.t]$ | - | - | - | - | - | - | - |
| `Fail`$(p)$ | - | - | - | $[p]$ | - | - | - | - | - | - |
| `SendMsg`$(p,\ldots)$ | - | - | - | - | - | - | $[p]$ | - | - | - |
| `TXMsg`$(p)$ | - | - | - | - | - | - | $[p]$ | $[q \in p^+(G)]$ | - | - |
| `DeliverMsg`$(p)$ | - | - | - | - | - | - | - | $[p]$ | $[*]$ | - |
| `DetectFail`$(p,q)$ | - | - | - | - | - | - | - | $[p]$ | - | $[p][q]$ |

TABLE 5.1: The effect of the next-state relations on AllConcur's state. The brackets [] indicate what elements of the variables are modified; $[*]$ indicates that the entire variable is modified. Also, $p^+(G)$ denotes the set of successors of $p$ in $G$ (see Table 2.1); $m.o$ denotes the owner of a message $m$ (i.e., the server that first sent $m$); and $m.t$ denotes the server targeted by a failure notification $m$, i.e., the failed server.

- $M[p]$ is the set of A-broadcast messages known by $p$; actually, $M[p]$ contains the messages owners, i.e., $M[p] \subseteq S$.
- $g[p]$ is an array of $n$ tracking digraphs, one per server; the digraph $g[p][q]$ is used by $p$ to track the message A-broadcast by a server $q$.
- $F[p]$ is an array of $n$ sets, one per server; the set $F[p][q]$ contains all servers from which $p$ received notifications of $q$'s failure.
- *flag*$[p]$ is a record with three binary fields—*nf* indicates whether $p$ is non-faulty, *ab* indicates whether $p$ A-broadcast its message, and *done* indicates whether $p$ completed the round. For ease of notation, we omit the *flag* prefix and refer to the three flags as $nf[p]$, $ab[p]$ and $done[p]$, respectively.

**Initial state.** Initially, all sets in both $M[p]$ and $F[p]$ are empty—$p$ neither knows of any A-broadcast message nor has received any failure notifications. For any $q \in S$, $g[p][q]$ contains only $q$ as node—initially, $p$ expects to receive each message only from its owner. Also, $p$ is initially non-faulty and it has neither A-broadcast its message nor completed the round.

**Next-state relations.** The AB module defines six operators that specify all the possible state transitions (see Table 5.1). In addition to the two exposed interfaces, i.e., `Abcast`$(p)$ and `Adeliver`$(p)$, $p$ can perform the following four actions: ( 1) receive a message, i.e., `ReceiveMessage`$(p)$; (2) invoke the NET module for transmitting a message, i.e., `TXMsg`$(p)$ (§ 5.5.1.3);done (3) fail, i.e., `Fail`$(p)$; and (4) invoke the FD module for detecting the failure of a predecessor $q \in S$, i.e., `DetectFail`$(p,q)$ (§ 5.5.1.4).

The `Abcast`$(p)$ operator updates $M[p]$, $g[p]$ and $ab[p]$—it adds $p$ to $M[p]$; it removes all servers from $g[p][p].node$; and it sets the $ab[p]$ flag. Also, it sends $p$'s message further by invoking the `SendMsg` operator of the NET module (§ 5.5.1.3). The main precondition of the operator is that $p$ has not A-broadcast its message already; hence, a message can be A-broadcast at most once.

The `Adeliver`$(p)$ operator sets the $done[p]$ flag; as a result, $p$ can A-deliver the messages in $M[p]$ in a deterministic order. The main precondition of the operator is that all $p$'s tracking digraphs are empty, i.e., $g[p][q].node = \emptyset, \forall q \in S$. In Section 5.5.2, we show that this precondition is sufficient for safety.

The `ReceiveMessage`$(p)$ operator invokes the `DeliverMsg` operator of the NET module; as a result, the least recent message from the *recvBuf* is stored into *recvMsg* (§ 5.5.1.3).

AllConcur distinguishes between A-broadcast messages and failure notifications. For both, the $o$ field indicates the server that first sent the message, i.e., the *owner*; also, the $t$ field of a failure notification indicates the server suspected to have failed, i.e., the *target*. When $p$ receives a message $m$, it invokes either `RecvBCAST`$(p,m)$ or `RecvFAIL`$(p,m)$. To avoid resends, both operators are enabled only if $p$ has not already received $m$.

The `RecvBCAST`$(p,m)$ operator updates both $M[p]$ and $g[p]$—it adds $m.o$ to $M[p]$; and it removes all servers from $g[p][m.o].node$. Also, it sends $m$ further by invoking the `SendMessage` operator of the NET module (§ 5.5.1.3). In addition, if $p$ has not A-broadcast its message, receiving $m$ triggers the `Abcast`$(p)$ operator. One of the precondition of the operator is that $m$ was A-broadcast by its owner; although this condition is ensured by design, it facilitates the proof of the integrity property (§ 5.5.2).

The `RecvFAIL`$(p,m)$ operator updates both $F[p]$ and $g[p]$—it adds $m.o$ to $F[p][m.t]$ and updates every tracking digraph, in $g[p]$, that contains $m.t$. Updating the tracking digraphs is the core of AllConcur's early termination mechanism § 5.2.1 and we describe it in details in Section 5.5.1.2.

The `Fail`$(p)$ operator clears the $nf[p]$ flag. As a result, all of $p$'s operators are disabled—AllConcur assumes a fail-stop model. The main precondition of the operator is that less than $f$ servers have already failed.

The `SendMsg`, `DeliverMsg` and `TXMsg` operators are discussed in Section 5.5.1.3, while the `DetectFail` operator is discussed in Section 5.5.1.4.

#### 5.5.1.2 Updating the tracking digraphs

Tracking digraphs are trivially updated when $p$ adds an A-broadcast message to the set $M[p]$ and, as a result, removes all the servers from the digraph used (by $p$) to track this message. Updating the tracking digraphs becomes more involved when $p$ receives a failure notification. Let $p_t$ be the target and $p_o$ the owner of a received failure notification, i.e., $p_o$ detected $p_t$'s failure. Then, $p$ updates every tracking digraph, in $g[p]$, that contains $p_t$. Let $g[p][p_*]$ be such a digraph; if, after the update, $g[p][p_*]$ contains only servers known by $p$ to have failed, then it is completely pruned—$p$ is certain no non-faulty server can have $m_*$. We specify two approaches to update $g[p][p_*]$. The first approach follows the algorithm described in Section 5.3 (see Algorithm 5.1); however, due to its recursive specification, it is not suitable for the TLA+ Proof System (TLAPS) [42]. The second approach constructs the tracking digraph from scratch, using the failure notifications from $F[p]$; however, it requires the TLA+ Model Checker [94] to enumerate all paths of a digraph.

**First approach—recursive specification.** Let $p_t^+(g[p][p_*])$ be the set of $p_t$'s successors in $g[p][p_*]$ (see Table 2.1). Then, if $p_t^+(g[p][p_*]) \neq \emptyset$, then $p$ already received another notification of $p_t$'s failure, which resulted in $p$ suspecting $p_t$ to have sent (before failing) $m_*$ to its other successors, including $p_o$; thus, after receiving the failure notification, $p$ removes the edge $(p_t, p_o)$ from $g[p][p_*]$. Removing an edge, may disconnect some servers from the root $p_*$; intuitively, these servers cannot have received $m_*$ from any of the other suspected servers and thus, are removed from $g[p][p_*]$.

If $p_t^+(g[p][p_*]) = \emptyset$ (i.e., this is the first notification of $p_t$'s failure $p$ receives), then, we update $g[p][p_*]$ using a recursive function that takes two arguments—a FIFO queue $Q$ and a digraph $td$. Initially, $Q$ contains all the edges (in $G$) connecting $p_t$ to its successors (except $p_o$) and $td = g[p][p_*]$. Let $(x,y)$ be an edge extracted from $Q$. Then, if $y \in g[p][p_*]$, (i.e., $y$ is suspected by $p$ to have $m_*$), the edge $(x,y)$ is added to $td$ (i.e., $p$ suspects $y$ got $m_*$ from $x$). If $y \notin g[p][p_*]$, $y$ is also added to $td$ (i.e., $p$ suspects $y$ has $m_*$). In addition, if $F[p][y] \neq \emptyset$ (i.e., $y$ is known by $p$ to have failed), the edges connecting $y$ to its successors (expect those known

to have failed) are added to $Q$. Finally, the function is recalled with the updated $Q$ and $td$. The recursion ends when $Q$ is empty.

**Properties of tracking digraphs.** From the recursive specification, we deduce the following four invariants that uniquely define a non-empty tracking digraph, denoted by $\texttt{TD}(p, p_*)$:

- ($I_1$) it contains its root, i.e., $p_* \in \texttt{TD}(p, p_*).node$;
- ($I_2$) it contains all the successors of every server (in $G$) known to have failed, except those successors from which failure notifications were received, i.e.,

$$\forall \texttt{TD}(p, p_*).node : F[p][q] \neq \emptyset \Rightarrow \forall q_s \in q^+(G) \setminus F[p][q] \; : \; q_s \in \texttt{TD}(p, p_*).node;$$

- ($I_3$) it contains only edges that connect a server (in $G$) known to have failed to all its successors, except those successors from which failure notifications were received, i.e.,

$$\forall (e_1, e_2) \in \texttt{TD}(p, p_*).edge \; : \; (F[p][e_1] \neq \emptyset \wedge e_2 \in e_1^+(G) \setminus F[p][e_1]);$$

- ($I_4$) it contains only servers that are either the root or the successor of another server (in $G$) known to have failed, except those successors from which failure notifications were received, i.e.,

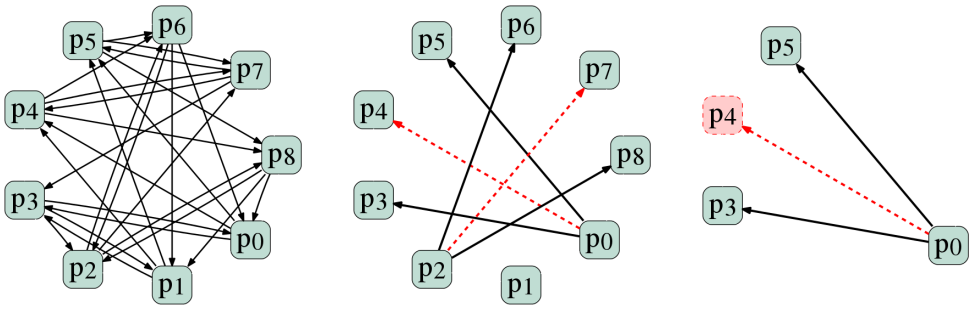
$$\forall q \in \texttt{TD}(p, p_*).node \; : \; (q = p_* \vee (\exists q_p \in \texttt{TD}(p, p_*).node \; : \wedge F[p][q_p] \neq \emptyset \\ \wedge q \in q_p^+(G) \setminus F[p][q_p])).$$

The intuition behind invariant $I_1$ is straightforward—while tracking $p_*$'s message, $p$ always suspects $p_*$ to have it. Invariants $I_2$ and $I_3$ describe how a tracking digraph expands. The successors of any server $q$, which is both suspected to have $m_*$ and known to have failed, are suspected (by $p$) to have received $m_*$ directly from $q$ before it failed. However, there is one exception—successors from which $p$ already received notifications of $q$'s failure. Receiving a notification of $q$'s failure before receiving $m_*$ entails the sender of the notification could not have received $m_*$ directly from $q$ (cf. Proposition 5.2.1).

$I_1$, $I_2$ and $I_3$ are necessary but not sufficient for a non-empty digraph to be a tracking digraph. As an example, we consider nine servers connected through a $G_S(9,3)$ digraph [150] (see Figure 5.5a). While $p_6$ is tracking $m_0$, it receives two notification, one from $p_4$ indicating $p_0$'s failure and another from $p_7$ indicating $p_2$'s failure; i.e., $F[p_6][p_0] = \{p_4\}$ and $F[p_6][p_2] = \{p_7\}$. Clearly, the digraph $\bar{K}_9(p_6)$ illustrated in Figure 5.5b satisfies the first three invariants. However, there is not reason for $p_6$ to suspect that $p_2$ has $m_0$.

For sufficiency, invariant $I_4$ is needed. Together with $I_3$, $I_4$ requires that $p$ suspects only those servers that are connected through failures to the root $p_*$. In other words, for $p$ to suspect a server $q$, there must be a sequence of servers starting with $p_*$ and ending with $q$ such that every server preceding $q$ is both known to have failed and suspected to have sent $m_*$ to the subsequent server. Note that a server is always connected through failures to itself. Figure 5.5c shows the actual digraph used by $p_6$ to track $m_0$ in the above example.

**Second approach—TLAPS specification.** We use the above invariants to provide a non-recursive specification of a non-empty tracking digraph. Let $K_n$ be a complete digraph with $n$ nodes; clearly, $K_n$ satisfies $I_1$ and $I_2$. Let $\bar{K}_n(p)$ be a digraph obtained from $K_n$ by removing all edges not satisfying $I_3$ (see Figure 5.5b for nine servers connected through a $G_S(9,3)$




(A) $G_S(9,3)$ (B) $\bar{K}_9(p_6)$ (C) $g[p_6][p_0]$

FIGURE 5.5: (a) An overlay network connecting nine servers; (b) a digraph that satisfies $I_1$, $I_2$ and $I_3$; and (c) the digraph used by $p_6$ to track $m_0$. Both digraphs consider nine servers connected through a $G_S(9,3)$ digraph [150] and are based on two failure notifications received by $p_6$ (indicated by dashed edges), one from $p_4$ indicating $p_0$'s failure and another from $p_7$ indicating $p_2$'s failure. Note that $p_4 \notin g[p_6][p_0]$.

digraph). Then, the set $\texttt{TD}(p, p_*).node$ contains any node in $\bar{K}_n(p)$ that is either $p_*$ or (according to $p$) is connected to $p_*$ through failures, i.e.,

$$\{q \in \bar{K}_n(p).node : \vee\, q = p_* \\ \vee\, \exists \pi_{p_*,q}(\bar{K}_n(p)) : F[p][x] \neq \emptyset, \forall x \in \pi_{p_*,q}(\bar{K}_n(p)) \setminus \{q\}\}, \tag{5.1}$$

where $\pi_{p_*,q}(\bar{K}_n(p))$ is a path in $\bar{K}_n(p)$ from $p_*$ to $q$ (see Table 2.1). Note that when removing a node from $\bar{K}_n(p)$ we also remove all the edges incident on that node. Using TLAPS, we prove that this specification satisfies all four invariants.

#### 5.5.1.3 The networking module

The NET module specifies an interface for asynchronous message-based communication; the module assumes that servers communicate through an overlay network. The interface considers three constants: (1) $S$, the set of servers; (2) $G$, the digraph that describes the overlay network; and (3) *Message*, the set of existing messages. We assume that every message has a field $o$ indicating the server that first sent it.

Let $p \in S$ be any server. Then, the state of the NET module is described by the values of three variables (see Figure 5.4): (1) $sendBuf[p]$, is $p$'s sending buffer; (2) $recvBuf[p]$, is $p$'s receiving buffer; and (3) *recvMsg*, is the latest received message. Note that while $recvBuf[p]$ is a sequence of received messages, $sendBuf[p]$ is a sequence of tuples, with each tuple consisting of a message and a sequence of destination servers. Also, both buffers act as FIFO queues. In the initial state, the buffers are empty sequences; the initial value of *recvMsg* is irrelevant.

**Next-state relations.** The NET module defines three operators that specify all the possible state transitions (see Table 5.1). The operators consists of the three main actions performed in message-based communication—sending, transmitting and delivering a message. To describe the three operators, let *msgs* be a sequence of messages and *nf* a mapping $S \to \{0,1\}$ indicating the non-faulty servers.

The $\texttt{SendMsg}(p, msgs, nf)$ operator, updates $sendBuf[p]$ by appending tuples consisting of messages from *msgs* with their destinations. For every message $m$, the set of destinations consist of $p$'s non-faulty successors, except for $m.o$. Note that the `SendMsg` operator has no precondition.

The `TXMsg`$(p)$ operator sends $m$, the next message from $sendBuf[p]$, to $q$, one of $m$'s destinations. The sequence of destinations of $m$ is updated by removing $q$. When there are no more destinations, $m$ is removed from $sendBuf[p]$. Also, $m$ is appended to $recvBuf[q]$. As a precondition, the send buffer of $p$ must not be empty.

The `DeliverMsg`$(p)$ operator updates $recvBuf[p]$ by removing a message (i.e., the least-recent received) and storing it in *recvMsg*. Note that *recvMsg* is only a temporary variable used by the AB module to access the delivered message (i.e., the `ReceiveMessage` operator). As a precondition, the receive buffer of $p$ must not be empty.

#### 5.5.1.4 The failure detector module

The FD module provides information about faulty servers. It specifies a failure detector that guarantees both completeness and accuracy, i.e., $\mathscr{P}$ [34]. The specification assumes a heartbeat-based failure detector with local detection: Every server sends heartbeats to its successors; once it fails, its successors detect the lack of heartbeats. The module considers two constants: (1) $S$, the set of servers; and (2) $G$, the digraph that describes the overlay network.

Let $p \in S$ be any server. Then, the state of the FD module is described by the values of one variable (see Figure 5.4)—$FD[p]$ is an array of $n$ flags, one per server; $FD[p][q]$ indicates whether $p$ has detected $q$'s failure. Clearly, $FD[p][q] = 1 \Rightarrow q \in p^+(G)$. Initially, all flags in $FD[p]$ are cleared.

**Next-state relation.** The FD module defines only one operator, `DetectFail`$(p,q)$, that specifies the state transition when $p$ detects $q$'s failure; i.e., the $FD[p][q]$ flag is set (see Table 5.1). The operator has a set of preconditions. First, $p$ must be both non-faulty and a successor of $q$. Second, $q$ must be faulty, i.e., $nf[q] = 0$; this condition guarantees the accuracy property required by $\mathscr{P}$.

Once a failure is detected, the AB module must be informed. The FD module invokes the NET module to append a notification of $q$'s failure to $recvBuf[p]$ (see Figure 5.4). This ensures that any A-broadcast messages sent by $q$ to $p$ that were already transmitted (i.e., added to $recvBuf[p]$) are delivered by $p$ before its own notification of $q$'s failure. Thus, Proposition 5.2.1 holds: If $p$ receives from $q_s \in q^+(G)$ a notification of $q$'s failure, then $q_s$ has not received from $q$ any message that $p$ did not already receive. In the above scenario, $q_s = p$.

#### 5.5.1.5 Safety and liveness properties

Using the above specification, we define both safety and liveness properties. First, AllConcur relies on a perfect failure detector for detecting faulty servers; hence, it guarantees both accuracy and completeness (see Section 2.2.4). *Accuracy* is a safety property: It requires that no server is suspected to have failed before actually failing, i.e.,

$$\forall q \in S : nf[q] = 1 \Rightarrow \forall p \in S : FD[p][q] = 0.$$

*Completeness* is a liveness property: It requires that all failures are eventually detected, i.e.,

$$\forall q \in S : nf[q] = 0 \rightsquigarrow (\forall p \in q^+(G) : nf[p] = 1 \Rightarrow FD[p][q] = 1),$$

where $X \rightsquigarrow Y$ asserts that whenever $X$ is true, $Y$ is eventually true [94].

Second, any non-uniform atomic broadcast algorithm must satisfy four properties—validity, agreement, integrity, and total order (see Section 2.2.1). Integrity and total order are safety properties. *Integrity* requires for any message $m$, every non-faulty server to A-deliver

$m$ at most once, and only if $m$ was previously A-broadcast by its owner $q$, i.e.,

$$\forall p \in S : nf[p] = 1 \Rightarrow \forall q \in M[p] \, : \, ab[q] = 1.$$

Note that the requirement that a server A-delivers $m$ at most once is ensured by construction, i.e., $M[p]$ is a set.

*Total order* asserts that if two non-faulty servers $p$ and $q$ A-deliver messages $m_1$ and $m_2$, then $p$ A-delivers $m_1$ before $m_2$, if and only if $q$ A-delivers $m_1$ before $m_2$. Since $p$ A-delivered messages in $M[p]$ in a deterministic order, we replace total order with *set agreement*: Let $p$ and $q$ be any two non-faulty servers, then, after completing the round, $M[p] = M[q]$, i.e.,

$$\forall p,q \in S : (nf[p] = 1 \wedge done[p] = 1 \wedge nf[q] = 1 \wedge done[q] = 1) \Rightarrow M[p] = M[q].$$

Validity and agreement are liveness properties. *Validity* asserts that if a non-faulty server A-broadcasts a message, then it eventually A-delivers it, i.e.,

$$\forall p \in S : (\Box(nf[p] = 1) \wedge ab[p] = 1) \rightsquigarrow a\text{-}deliver(p,p),$$

where $\Box X$ asserts that $X$ is always true [94]; also,

$$a\text{-}deliver(p,q) = q \in M[p] \wedge done[p] = 1$$

asserts the conditions necessary for $p$ to A-deliver the message A-broadcast by $q$.

*Agreement* asserts that if a non-faulty server A-delivers a message A-broadcast by any server, then all non-faulty servers eventually also A-deliver the message.

$$\forall p,q,s \in S :(\Box(nf[p] = 1) \wedge a\text{-}deliver(p,s)) \rightsquigarrow (nf[q] = 1 \Rightarrow a\text{-}deliver(q,s)).$$

To verify that all the above properties hold, we use the TLA+ Model Checker [94], hence, the need for a tracking digraph specification that does not enumerate all paths of a digraph (§ 5.5.1.2). For a small number of servers, e.g., $n = 3$, the model checker can do an exhaustive search of all reachable states. Yet, for larger values of $n$ the exhaustive search becomes intractable. As an alternative, we use the model checker to randomly generate state sequences that satisfy both the initial state and the next-state relations. In the model, we consider the overlay network is described by a $G_S(n,d)$ digraph [150]. When choosing $G_S(n,d)$'s degree, we require a reliability target of 6-nines; the reliability is estimated over a period of 24 hours and a server $MTTF \approx 2$ years [67].

In addition, we use TLAPS to formally prove the safety properties[4]—the failure detector's accuracy and the atomic broadcast's integrity and set agreement (§ 5.5.2).

### 5.5.2 Proof of safety

Atomic broadcast has two safety properties—integrity and total order. In AllConcur, total order can be replaced by set agreement (§ 5.5.1.5). Note that the formal proof of safety considers non-uniform properties, i.e., set agreement applies only to non-faulty servers. In addition, AllConcur relies on a perfect failure detector for information about faulty servers; thus, for safety, accuracy must also hold. For all three safety properties, we use TLAPS [94] to provide mechanically verifiable proofs. All three proofs follow the same pattern: We consider each property to be an invariant that holds for the initial state and is preserved by the next-state relations.

[4]TLAPS does not allow for liveness proofs.

**Accuracy.** The failure detector's accuracy is straightforward to prove. Initially, all flags in $FD$ are cleared; hence, the property holds. Then, according to the specification of the `DetectFail` operator (§ 5.5.1.4), setting the flag $FD[p][q]$ for $\forall p,q \in S$ is preconditioned by $q$ previously failing, i.e., $nf[q] = 0$. Moreover, due to the fail-stop assumption, a faulty server cannot become subsequently non-faulty. As a result, accuracy is preserved by the next-state relations.

**Integrity.** Integrity is also straightforward to prove. Initially, all the sets in $M$ are empty; hence, the property holds. Then, the only two operators that update $M$ are `Abcast` and `RecvBCAST` (see Table 5.1). The `Abcast`$(p)$ operator adds $p$ to $M[p]$ and also sets the $ab[p]$ flag; hence, integrity is preserved by `Abcast`. The `RecvBCAST`$(p,m)$ operator adds $m.o$ to $M[p]$. Clearly, receiving a message $m$ entails the existence of a path from $m.o$ to $p$ such that each server on the path has $m$ in its $M$ set (see Lemma 5.5.1 below); this also includes $m.o$. When $m.o$ adds its message $m$ to $M[m.o]$, it also sets the $ab[m.o]$ flag (according to the `Abcast` operator). Thus, integrity is preserved also by `RecvBCAST`. In practice, to simplify the TLAPS proof, we introduce an additional precondition for the `RecvBCAST`$(p,m)$ operator—$ab[m.o] = 1$ (§ 5.5.1.1).

#### 5.5.2.1 Set agreement

The set agreement property is the essence of AllConcur—it guarantees the total order of broadcast messages. Clearly, set agreement holds in the initial state, since all *done* flags are cleared. Moreover, the *done* flags are set only by the `Adeliver` operator (see Table 5.1); thus, we only need to prove set agreement is preserved by `Adeliver`. We follow the informal proof provided in Section 5.4. We introduce the following lemmas:

**Lemma 5.5.1.** *Let $p$ be a server that receives $p_*$'s message $m_*$; then, there is a path (in $G$) from $p_*$ to $p$ such that every server on the path has received $m_*$ from its predecessor on the path, i.e.,*

$$\begin{aligned}
&\forall p, p_* \in S : p_* \in M[p] \Rightarrow \exists \pi_{p_*,p}(G) = (a_1, \ldots, a_\lambda) : \\
&\quad \wedge \forall k \in \{1, \ldots, \lambda\} : p_* \in M[a_k] && \text{(5.2a)} \\
&\quad \wedge \forall q \in S : (\exists k \in \{1, \ldots, (\lambda - 1)\} : a_{k+1} \in F[q][a_k]) \Rightarrow p_* \in M[q]. && \text{(5.2b)}
\end{aligned}$$

*Proof.* Equation (5.2a) is straightforward. Equation (5.2b) ensures that every server on the path (except $p_*$) received $m_*$ from its predecessor. Any server $q$ that received a failure notification from a server on the path (except $p_*$) targeting its predecessor on the path also received $m_*$ (cf. Proposition 5.2.1). □

**Lemma 5.5.2.** *Let $p$ be a non-faulty server that does not receive $p_*$'s message $m_*$; let $\pi_{p_*,a_i}(G) = (a_1, \ldots, a_i)$ be a path (in $G$) on which $a_i$ receives $m_*$; let $(a_1, \ldots, a_i)$ be also a path in $g[p][p_*]$. Then,*

$$done[p] = 1 \Rightarrow (\forall q \in g[p][p_*].node : F[p][q] \neq \emptyset).$$

*Proof.* A necessary condition for $p$ to complete the round is to remove every server $a_j, \forall 1 \leq j \leq i$ from $g[p][p_*]$. According to AllConcur's specification (§ 5.5.1.2), a server $a_j$ can be removed from $g[p][p_*]$ in one of the following scenarios: (1) $p_* \in M[p]$; (2) $\nexists \pi_{p_*,a_j}(g[p][p_*])$; and (3) $\forall q \in g[p][p_*].node : F[p][q] \neq \emptyset$. Clearly, $p_* \notin M[p]$. Also, $\nexists \pi_{p_*,a_j}(g[p][p_*])$ entails at least the removal of an edge from the $(a_1, \ldots, a_j)$ path. Let $(a_l, a_{l+1}), 1 \leq l < j$ be one of the removed edges. Then, $a_{l+1} \in F[p][a_l]$, which entails $p_* \in M[p]$ (cf. Equation (5.2b)). Thus, $p$ completes the round only if $\forall q \in g[p][p_*].node : F[p][q] \neq \emptyset$. □

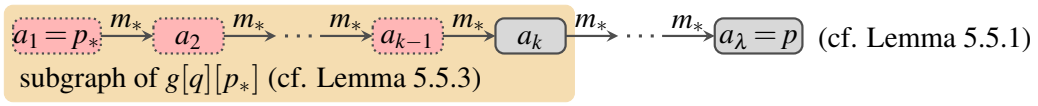



FIGURE 5.6: The path $\pi_{p_*,p}$ on which a non-faulty server $p$ first receives $m_*$ from $p_*$; every server on the path (except $p_*$) has received $m_*$ from its predecessor. Dotted red boxes indicate faulty servers; in case of no failures, $a_k = p_*$. Solid gray boxes indicate non-faulty servers. If another non-faulty server $q$ does not receive $m_*$, then it suspects all servers on the subpath from $a_1$ to $a_k$ to have $m_*$ and to have received $m_*$ from their predecessors on the path (except $p_*$).

**Lemma 5.5.3.** *Let $p,q$ be two non-faulty servers such that $p_* \in M[p]$, but $p_* \notin M[q]$; let $\pi_{p_*,p}(G) = (a_1,\ldots,a_\lambda)$ be the path on which $p$ receives $m_*$; let $a_k$ be a server on $\pi_{p_*,p}(G)$ such that $nf[a_k] = 1$ and $nf[a_i] = 0, \forall 1 \leq i < k$. Then, $(a_1,\ldots,a_i), \forall 1 \leq i \leq k$ is a path in $g[q][p_*]$, i.e., $(a_1,\ldots,a_i) \in g[q][p_*]$.*

*Proof.* We use mathematical induction: The basic case is given by invariant $I_1$ (§ 5.5.1.2), $a_1 = p_* \in g[q][p_*].node$. For the inductive step, we assume $(a_1,\ldots,a_i) \in g[q][p_*]$ for some $1 \leq i < k$. Due to the FD's completeness property (§ 5.5.1.5), the failure of $a_j, \forall 1 \leq j \leq i$ is eventually detected. Due to the vertex-connectivity of $G$, $q$ eventually receives the failure notification of every $a_j$. Moreover, $q$ cannot remove $a_j, \forall 1 \leq j \leq i$ from $g[q][p_*]$ before it receive failure notifications of every $a_j$ (cf. Lemma 5.5.2). Thus, eventually $F[q][a_i] \neq \emptyset$ and, since $a_{i+1} \notin F[q][a_i]$ (cf. Lemma 5.5.1), $a_{i+1} \in g[q][p_*].node$ and $(a_i,a_{i+1}) \in g[q][p_*].edge$ (according to invariants $I_2$ and $I_3$). □

**Theorem 5.5.4.** *AllConcur's specification guarantees set agreement.*

*Proof.* Let $p_* \in M[p]$, but $p_* \notin M[q]$. Then, $\exists \pi_{p_*,p} = (a_1,\ldots,a_\lambda)$ on which $m_*$ first arrives at $p$ (cf. Lemma 5.5.1). Let $a_k$ be a server on $\pi_{p_*,p}$ such that $nf[a_k] = 1$ and $nf[a_i] = 0, \forall 1 \leq i < k$; the existence of $a_k$ is given by the existence of $p$, a server that is both non-faulty and on $\pi_{p_*,p}$. The path $\pi_{p_*,p}$ is illustrated in Figure 5.6; the faulty servers are indicated by dotted red boxes. Before $q$ terminates, $a_k \in g[q][p_*].node$ (cf. Lemma 5.5.3). However, due to the precondition of `Adeliver`$(q)$ (§ 5.5.1.1), i.e.,

$$done[q] = 1 \Rightarrow g[q][s].node = \emptyset, \forall s \in S,$$

$a_k$ is subsequently removed from $g[q][p_*]$. Since $m_* \notin M[q]$ and $nf[a_k] = 1$, $a_k$ must have been disconnected from $p_*$ (§ 5.5.1.2). Thus, $q$ removed at least one edge from $(a_1,\ldots,a_k)$; let $(a_i,a_{i+1}), 1 \leq i < k$ be one of the removed edges. Then, $a_{i+1} \in F[q][a_i]$ (cf. $I_3$), which entails $p_* \in M[q]$ (cf. Lemma 5.5.1). □

**Reconstructed tracking digraphs.** The set agreement proof relies on the following property of tracking digraphs (§ 5.5.1.2): If a server $q$ was removed from $g[p][p_*]$, then one of the following is true—(1) $p_* \in M[p]$; (2) $F[p][q] \neq \emptyset$[5]; and (3) $\nexists \pi_{p_*,q}(g[p][p_*])$. However, when the `Adeliver`$(p)$ operator is enabled, $g[p][p_*].node = \emptyset$; checking the existence of a path $\pi_{p_*,q}(g[p][p_*])$ is not possible. Therefore, we reconstruct $g[p][p_*]$ from the failure notifications received by $p$; we denote the resulted digraph by `RTD`$(p,p_*)$.

[5] $F[p][q] \neq \emptyset$ is necessary, but not sufficient to remove $q$ from $g[p][p_*]$.

Constructing the set $\texttt{RTD}(p,p_*).node$ is similar to the TLAPS specification of updating tracking digraphs (§ 5.5.1.2). The difference is twofold. First, if $p$ receives from $p_o$ a notification of $p_t$'s failure, with $p_t \in \texttt{RTD}(p,p_*)$, then $p$ adds all $p_t$'s successors to $\texttt{RTD}(p,p_*)$—including $p_o$. Second, servers are never removed from $\texttt{RTD}(p,p_*)$. Clearly, every server added at any point to $g[p][p_*]$ is also in $\texttt{RTD}(p,p_*)$

To construct the set $\texttt{RTD}(p,p_*).edge$, we first connect (in $\texttt{RTD}(p,p_*)$) each server known to have failed to its successors. Then, we remove all the edges on which we are certain $p_*$'s message $m_*$ was not transmitted. To identify these edges we use Proposition 5.2.1: If $p$ received from $e_2$ a notification of $e_1$ failure without previously receiving $m_*$, then the edge $(e_1,e_2)$ is not part of $\texttt{RTD}(p,p_*)$, i.e.,

$$\forall e_1,e_2 \in \texttt{RTD}(p,p_*).node : e_2 \in F[p][e_1] \wedge p_* \notin M[p] \Rightarrow (e_1,e_2) \notin \texttt{RTD}(p,p_*).edge\,. \tag{5.3}$$

Some of the remaining edges in $\texttt{RTD}(p,p_*)$ were still not used for transmitting $m_*$, i.e., the edges for which $e_2 \in F[p][e_1]$ occurred before $p_* \in M[p]$. Since the $\texttt{Adeliver}(p)$ operator has no access to the history of state updates modifying either $F[p][e_1]$ or $M[p]$, we cannot identify these edges. Yet, since $p_* \in M[p]$, these edges play no role in proving set agreement. Clearly, every edge added at any point to $g[p][p_*]$ is also in $\texttt{RTD}(p,p_*)$.

Using reconstructed tracking digraphs, we redefine the above property of tracking digraphs as an invariant, referred to as the *RTD invariant*:

$$\begin{aligned}\forall p,q,p_* \in S : (q \in \texttt{RTD}(p,p_*).node \wedge q \notin g[p][p_*].node) \Rightarrow &\vee p_* \in M[p]\\ &\vee F[p][q] \neq \emptyset\\ &\vee \nexists \pi_{p_*,q}(\texttt{RTD}(p,p_*))\,.\end{aligned}$$

The RTD invariant enables us to formally prove set agreement using TLAPS. Clearly, when $\texttt{Adeliver}(q)$ is enabled, $a_k$ (from the proof of Theorem 5.5.4) is in $\texttt{RTD}(q,p_*)$, but not in $g[q][p_*]$. According to the initial assumptions, $p_* \notin M[q]$ and $F[q][a_k] = \emptyset$. Moreover, $\pi_{p_*,a_k} = (a_1,\ldots,a_k)$ was at some point a path in $g[q][p_*]$ (cf. Lemma 5.5.3); hence, $\pi_{p_*,a_k}$ is also a path in $\texttt{RTD}(p,p_*)$ (follows from RTD's construction), which contradicts the RTD invariant.

**Proving the RTD invariant.** To prove the RTD invariant, we follow the same pattern as before: We prove that the invariant holds for the initial state (since $\texttt{RTD}(p,p_*) = g[p][p_*], \forall p,p_*$) and is preserved by the only three operators that update the $M$, $G$ or $F$ variables—`Abcast`; `RecvBCAST`; `RecvFAIL` (see Table 5.1). In the following proofs, $X'$ denotes the updated value of a variable $X$ after applying an operator [94].

**Theorem 5.5.5.** *Both* `Abcast` *and* `RecvBCAST` *operators preserve the RTD invariant.*

*Proof.* According to the specifications of both `Abcast` and `RecvBCAST` (§ 5.5.1.1),

$$g[p][p_*] \neq g'[p][p_*] \Rightarrow p_* \in M'[p]\,;$$

therefore, we assume $g[p][p_*] = g'[p][p_*]$. In addition, since $F[p] = F'[p]$, $\texttt{RTD}(p,p_*).node = \texttt{RTD}(p,p_*)'.node$. Thus, since $M[p] \subseteq M'[p]$, the only possibility for the RTD invariant to not be preserved by either operators is $\exists \pi_{p_*,q}(\texttt{RTD}(p,p_*)')$. Let $(e_1,e_2)$ be one of the edges that enables such a path, i.e.,

$$(e_1,e_2) \notin \texttt{RTD}(p,p_*).edge \wedge (e_1,e_2) \in \texttt{RTD}(p,p_*)'.edge\,.$$

As a result,

$$e_2 \in F[p][e_1] \wedge p_* \notin M[p] \wedge (e2 \notin F'[p][e_1] \vee p_* \in M'[p]) \quad \text{(cf. Equation (5.3))},$$

which is equivalent to $p_* \in M'[p]$. □

**Theorem 5.5.6.** *The* `RecvFAIL` *operator preserves the RTD invariant.*

*Proof.* We consider $\texttt{RecvFAIL}(s,m)$, with $s \in S$; clearly, $M = M'$ (see Table 5.1). Also, if $s \neq p$, the RTD invariant is preserved, since also $F[p] = F'[p]$ and $g[p] = g'[p]$. Thus, the proof is concerned only with $\texttt{RecvFAIL}(p,m)$. Aside from adding $m.o$ to $F[p][m.t]$, $F = F'$. Clearly, $q = m.t \Rightarrow F'[p][q] \neq \emptyset$. Thus, we make the following assumptions, i.e., the only non-trivial case (cf. RTD invariant):

| | | | | | |
|---|---|---|---|---|---|
| $(A_1)$ | $q \neq m.t$; | $(A_3)$ | $q \notin g'[p][p_*].node$; | $(A_5)$ | $F'[p][q] = \emptyset$; |
| $(A_2)$ | $q \in \texttt{RTD}(p,p_*)'.node$; | $(A_4)$ | $p_* \notin M'[p]$; | $(A_6)$ | $\exists \pi_{p_*,q}(\texttt{RTD}(p,p_*)')$. |

We split the remainder of the proof in three steps.

*Step 1.* We show that $q$ is in $\texttt{TD}(p,p_*)'$ (§ 5.5.1.2). Let

$$\pi_{p_*,q}(\texttt{RTD}(p,p_*)') = (a_1,\ldots,a_\lambda) \quad \text{(cf. } (A_6)\text{)};$$

then

$$(a_k,a_{k+1}) \in \texttt{TD}(p,p_*)'.edge, \forall 1 \leq k < \lambda\,,$$

since $p_* \notin M'[p]$ (cf. $(A_4)$) and no edges are removed from $\texttt{RTD}(p,p_*)'$ (cf. Equation (5.3)). Therefore,

$$a_k \in \texttt{TD}(p,p_*)'.node, \forall 1 \leq k \leq \lambda \quad \text{(cf. Equation (5.1))},$$

which entails the following assumption:

$$(A_7) \quad q \in \texttt{TD}(p,p_*)'.node.$$

*Step 2.* We show that $g[p][p_*]$ is not updated. If $g'[p][p_*] \neq g[p][p_*]$, then, $g'[p][p_*]$ is either $\texttt{TD}(p,p_*)'$ or an empty digraph obtained after completely pruning $\texttt{TD}(p,p_*)'$. Since the latter option requires $F'[p][s] \neq \emptyset, \forall s \in \texttt{TD}(p,p_*)'.node$ (§ 5.5.1.2), which contradicts $(A_5)$ and $(A_7)$, we assume $g'[p][p_*] = \texttt{TD}(p,p_*)'$. Yet, this entails $q \in g'[p][p_*].node$ (cf. $A_7$), which contradicts $(A_3)$. Thus, we make the following two equivalent assumptions (§ 5.5.1.2):

$$(A_8) \quad g[p][p_*] = g'[p][p_*]; \qquad (A_9) \quad m.t \notin g[p][p_*].node.$$

*Step 3.* We distinguish between three scenarios—according to the specification (§ 5.5.1.1), $g[p][p_*]$ can be in one of the following three states: (1) an initial state, i.e., $g[p][p_*].node = \{p_*\}$, with $F[p][p_*] = \emptyset$; (2) a final state, i.e., $g[p][p_*].node = \emptyset$; and (3) an intermediary state, i.e., $g[p][p_*] = \texttt{TD}(p,p_*)$.

The *initial state* entails $g'[p][p_*].node = \{p_*\}$ (cf. $(A_8)$); thus, $p_*$ is neither $m.t$ (cf. $(A_9)$) nor $q$ (cf. $(A_3)$). Clearly,

$$p_* \neq m.t \Rightarrow F[p][p_*] = F'[p][p_*]\,.$$

Moreover, according to $(A_6)$ and the construction of $\texttt{RTD}(p,p_*)'$, $p_* \neq q$ entails $F'[p][p_*] \neq \emptyset$. Yet, this contradicts the condition of the initial state, i.e., $F[p][p_*] = \emptyset$.

The *final state*, i.e., $g[p][p_*].node = \emptyset$, can be reached only by completely pruning the tracking digraph (cf. ($A_4$) and $M[p] \subseteq M'[p]$); i.e.,

$$F[p][s] \neq \emptyset, \forall s \in \mathtt{TD}(p, p_*).node\,.$$

According to invariant $I_2$, no server can be added to $\mathtt{TD}(p, p_*)$, regardless of the received failure notification. As a result, $q \in \mathtt{TD}(p, p_*).node$ (cf. ($A_7$)) and, thus, $F[p][q] \neq \emptyset$. Yet, since $F[p][q] = F'[p][q]$ (cf. ($A_1$)), this contradicts ($A_5$).

The *intermediary state* entails $g'[p][p_*] = \mathtt{TD}(p, p_*)$ (cf. ($A_8$)); thus, $\nexists \pi_{p_*,q}(\mathtt{TD}(p, p_*))$ (cf. ($A_3$)). Nonetheless, according to ($A_7$) and Equation (5.1),

$$\exists \pi_{p_*,q}(\mathtt{TD}(p, p_*)') = (b_1, \ldots, b_\gamma) : F'[p][b_k] \neq \emptyset, \forall 1 \leq k < \gamma.$$

Moreover,

$$\exists 1 \leq k \leq \gamma : (b_k, b_{k+1}) \notin \mathtt{TD}(p, p_*).edge\,;$$

let $k$ be the smallest index satisfying this property. Then, either $F[p][b_k] = \emptyset$ or $b_{k+1} \in F[p][b_k]$ (cf. invariant $I_3$). If $F[p][b_k] = \emptyset$, then $b_k = m.t$ (since $F'[p][b_k] \neq \emptyset$). Also, $(b_1, \ldots, b_k)$ is a path in $\mathtt{TD}(p, p_*)$ (cf.invariants $I_1$, $I_2$ and $I_3$) and, thus, $b_k \in \mathtt{TD}(p, p_*).node$, which entails $m.t \in g'[p][p_*]$. Yet, this contradicts ($A_8$) and ($A_9$). If $b_{k+1} \in F[p][b_k]$, then $b_{k+1} \in F'[p][b_k]$ (i.e., $F[p][b_k] \subseteq F'[p][b_k]$), which contradicts $(b_k, b_{k+1}) \notin \mathtt{TD}(p, p_*)'.edge$. □

## 5.6 Performance analysis

AllConcur is designed as a high-performance atomic broadcast algorithm that enables large-scale state-machine replication. Its performance is given by three metrics: (1) work per server; (2) communication time; and (3) storage requirements. Our analysis focuses on Algorithm 5.1, which assumes both $\kappa(G) > f$ and $\mathscr{P}$, and it uses the LogP model [44]. The LogP model is described by four parameters: the latency $L$; the overhead $o$; the gap between messages $g$; and the number of servers $P$, which we denote by $n$. We make the common assumption that $o > g$ [7]; also, the model assumes short messages. AllConcur's performance depends on $G$'s parameters: the degree $d$, the diameter $D$, and the fault diameter $D_f$ (see Section 2.1.1). A discussion on how to choose $G$ is provided in Section 5.6.4.

### 5.6.1 Work per server

The amount of work a server performs is given by the number of messages it receives and sends. AllConcur distinguishes between A-broadcast messages and failure notifications. First, without failures, every server receives an A-broadcast message from all of its $d$ predecessors, i.e., $(n-1)\,d$ messages received by every server. This is consistent with the $\Omega(n^2 f)$ worst-case message complexity for synchronous consensus algorithms that tolerate up to $f$ failures [50]. Second, every failed server is detected by up to $d$ servers, each sending a failure notification to its $d$ successors. Therefore, every server receives up to $d^2$ notifications of each failure. Overall, every server receives at most $(n-1)\,d + f\,d^2$ messages. Since $G$ is regular, every server sends the same number of messages.

In order to complete a round, in a non-failure scenario, a server needs to receive at least $(n-1)$ messages and send them further to $d$ successors. We estimate the time of sending or receiving a message by the overhead $o$ of the LogP model [44]. Therefore, a lower bound on the time needed to complete a round (due to work) is given by $2(n-1)\,d\,o$. Moreover, in a non-failure scenario, completing a round results in A-delivering $n$ messages; this entails that the work per A-broadcast message is sublinear, i.e., $\mathscr{O}(d)$.

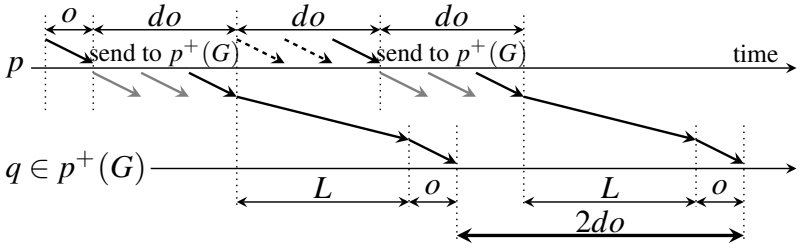



FIGURE 5.7: LogP model of message transmission in AllConcur for $d = 3$. Dashed arrows indicate already sent messages.

### 5.6.2 Communication time

In general, the time to transmit a message (between two servers) is estimated by

$$T(msg) = L + 2o\,.$$

We consider only the scenario of a single non-empty message $m$ being A-broadcast and we estimate the time between $sender(m)$ A-broadcasting $m$ and A-delivering $m$.

#### 5.6.2.1 Non-faulty scenario

We split the A-broadcast of $m$ in two parts: (1) $sender(m)$ R-broadcasting $m$; and (2) the empty messages $m_\emptyset$ traveling back to $sender(m)$. In a non-failure scenario, messages are R-broadcast in $D$ steps, i.e., $T_D(msg) = DT(msg)$. Moreover, to account for contention while sending to $d$ successors, we add to the sending overhead the expected waiting time, i.e.,

$$o_s = o + \frac{d-1}{2}\,o\,.$$

Note that for R-broadcasting $m$, there is no contention while receiving (every server, except $sender(m)$, is idle until it receives $m$). Thus, the time to R-broadcast $m$ is estimated by

$$T_D(m) = (L + o_s + o)\,D\,.$$

When the empty messages $m_\emptyset$ are transmitted to $sender(m)$, the servers are no longer idle; $T_D(m_\emptyset)$ needs to account for contention while receiving. On average, servers send further one in every $d$ received messages; thus, a server $p$ sends messages to the same successor $q$ at a period of $2do$ (see Figure 5.7). In general, once a message arrives at a server, it needs to contend with other received messages. During a round, a server receives $n-1$ messages, each from $d$ predecessors; this entails a receiving overhead of $do$ per message. Moreover, the server sends each message once to all its successors, which adds a sending overhead of $do$ per message. Overall, since servers handle incoming connections in a round-robin fashion, processing a round of messages from all $d$ predecessors takes (on average) $2do$. Thus, on average, the message in-rate on a connection matches the out-rate: There is no contention while receiving empty messages, i.e.,

$$T_D(m_\emptyset) = T_D(m)\,.$$

#### 5.6.2.2 Faulty scenario—probabilistic analysis

Let $\pi_m$ be the longest path a message $m$ has to travel before it is completely disseminated. If $m$ is lost (due to failures), $\pi_m$ is augmented by the longest path the failure notifications have to travel before reaching all non-faulty servers. Let $\mathscr{D}$ be a random variable that denotes the

length of the longest path $\pi_m$, for any A-broadcast message $m$, i.e.,

$$\mathscr{D} = \max_m |\pi_m|, \forall m.$$

We refer to $\mathscr{D}$ as AllConcur's *depth*. Therefore, $\mathscr{D}$ ranges from $D$, if no servers fail, to $f + D_f$ in the worst case scenario (see Section 5.2). However, $\mathscr{D}$ is not uniformly distributed. A back-of-the-envelope calculation shows that it is very unlikely for AllConcur's depth to exceed $D_f$.

We consider a single AllConcur round, with all $n$ servers initially non-faulty. Also, we estimate the probability $p_f$ of a server to fail, by using an exponential lifetime distribution model, i.e., over a period of time $\Delta$,

$$p_f(\Delta) = 1 - e^{-\Delta/MTTF}, \tag{5.4}$$

where *MTTF* is the mean time to failure. If $sender(m)$ succeeds in sending $m$ to all of its successors, then $D \leq \pi_m \leq D_f$. We estimate the probability of such an event by

$$Pr[D \leq \mathscr{D} \leq D_f] = (1 - p_f(d\,o))^n = e^{-n d o/MTTF},$$

where $o$ is the sending overhead [44]. Note that this probability increases if the round starts with previously failed servers.

For typical values of *MTTF* ($\approx$ 2 years [67]) and $o$ ($\approx 1.8\mu s$ for TCP on our InfiniBand cluster described in Section § 5.7), a system of 256 servers connected via a digraph of degree 7 (see Table 5.3) would finish 1 million AllConcur rounds with $\mathscr{D} \leq D_f$ with a probability larger than 99.99%. This demonstrates why early termination is essential for efficiency, as for most rounds no failures occur and even if they do occur, the probability of $\mathscr{D} > D_f$ is very small. Note that a practical deployment of AllConcur should include regularly replacing failed servers and/or updating $G$ after failures.

#### 5.6.2.3 Estimating the fault diameter

The fault diameter of any digraph $G$ has the following trivial bound [38, Theorem 6]

$$D_f(G,f) \leq \left\lfloor \frac{n-f-2}{k(G)-f} \right\rfloor + 1.$$

However, this bound is neither tight nor does it relate the fault diameter to the digraph's diameter. In general, the fault diameter is unbounded in terms of the digraph diameter [38]. Yet, if the first $f+1$ shortest vertex-disjoint paths from $u$ to $v$ are of length at most $\delta_f$ for $\forall u, v \in \mathscr{V}(G)$, then $D_f(G,f) \leq \delta_f$ [89]. To compute $\delta_f$, we need to solve the min-max $(f+1)$-disjoint paths problem for every pair of vertices: Find $(f+1)$ vertex-disjoint paths $\pi_0, \ldots, \pi_f$ that minimize the length of the longest path; hence,

$$\delta_f = \max_i |\pi_i|, 0 \leq i \leq f.$$

Unfortunately, the problem is known to be strongly NP-complete [104]. As a heuristic to find $\delta_f$, we minimize the sum of the lengths instead of the maximum length, i.e., the min-sum disjoint paths problem. This problem can be expressed as a minimum-cost flow problem; thus, it can be solved polynomially with well known algorithms, e.g., successive shortest path [5, Chapter 9]. Let $\widehat{\pi}_0, \ldots, \widehat{\pi}_f$ be the paths obtained from solving the min-sum disjoint paths problem; also, let

$$\widehat{\delta}_f = \max_i |\widehat{\pi}_i|, 0 \leq i \leq f.$$

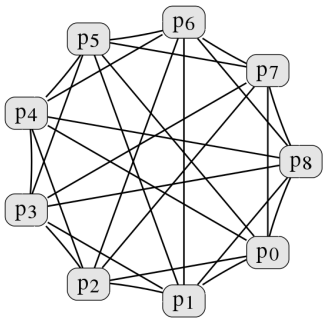



FIGURE 5.8: A binomial graph with $n = 12$ vertices.

Then, from the minimality condition of both min-max and min-sum problems, we deduce the following chain of inequalities:

$$\frac{\sum_{i=0}^{f} |\widehat{\pi}_i|}{f+1} \leq \frac{\sum_{i=0}^{f} |\pi_i|}{f+1} \leq \delta_f \leq \widehat{\delta}_f. \tag{5.5}$$

Thus, we approximate the fault diameter bound by $\widehat{\delta}_f$, i.e.,

$$D_f(G,f) \leq \widehat{\delta}_f.$$

Then, we use Equation (5.5) to check the accuracy of our approximation: We check the difference between the maximum and the average length of the paths obtained from solving the tractable min-sum problem.

As an example, we consider a binomial graph [11]—a generalization of 1-way dissemination [71]. In binomial graphs, two servers $pi$ and $pj$ are connected if

$$j = i \pm 2^l \pmod{n}, \forall 0 \leq l \leq \lfloor \log_2 n \rfloor.$$

Figure 5.8 shows a binomial graph with $n = 12$ vertexes and $p_i^+ = p_i^- = \{p_j : j = i \pm \{1,2,4\}\}$. The graph has connectivity $k = 6$ and diameter $D = 2$. After solving the min-sum problem, we can estimate the fault diameter bound, i.e., $3 \leq \delta_f \leq 4$. After a closer look, we can see that one of the six vertex-disjoint paths from $p_0$ to $p_3$ has length four, i.e., $p_0 - p_{10} - p_6 - p_5 - p_3$.

### 5.6.3 Storage requirements

Every server $p_i$ stores five data structures (see Algorithm 5.1): (1) the digraph $G$; (2) the set of known messages $M_i$; (3) the set of received failure notifications $F_i$; (4) the array of tracking digraphs $\mathbf{g_i}$; and (5) the internal FIFO queue $Q$. Table 5.2 shows the space complexity of each data structure. In general, for regular digraphs, $p_i$ needs to store $d$ edges per node; however, some digraphs require less storage, e.g., binomial graphs [11] require only the graph size. Also, each tracking digraph has at most $fd$ vertices; yet, only $f$ of these digraphs may have more than one vertex. The space complexity of the other data structures is straightforward (see Table 5.2).

### 5.6.4 Choosing the digraph $G$

AllConcur's performance depends on the parameters of $G$—degree, diameter, and fault diameter. Binomial graphs [11] have both diameter and fault diameter lower than other commonly

| Notation | Description | Space complexity per server |
|---|---|---|
| $G$ | digraph | $\mathcal{O}(n \cdot d)$ |
| $M_i$ | messages | $\mathcal{O}(n)$ |
| $F_i$ | failure notifications | $\mathcal{O}(f \cdot d)$ |
| $\mathbf{g_i}$ | tracking digraphs | $\mathcal{O}(f^2 \cdot d)$ |
| $Q$ | FIFO queue | $\mathcal{O}(f \cdot d)$ |

TABLE 5.2: Space complexity per server for data structures used by Algorithm 5.1. The space complexity for $G$ holds for regular digraphs, such as $G_S(n,d)$ § 5.6.4.

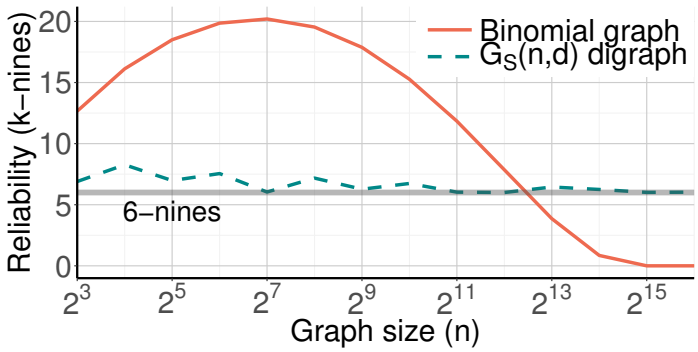



FIGURE 5.9: AllConcur's reliability estimated over a period of 24 hours and a server $MTTF \approx 2$ years.

used graphs, such as the binary Hypercube. Also, they are optimally connected, hence, offering optimal work for the provided connectivity. However, their connectivity depends on the number of vertices, which reduces their flexibility: Binomial graphs provide either too much or not enough connectivity.

We estimate AllConcur's reliability over a period of time $\Delta$ by

$$\rho_G(\Delta) = \sum_{i=0}^{k(G)-1} C(n,i)\, p_f(\Delta)^i\, (1-p_f(\Delta))^{n-i},$$

with $p_f$ defined in Equation 5.4. Figure 5.9 plots $\rho_G(\Delta)$ as a function of $n$, with $\Delta = 24$ hours. For a reliability target of 6-nines, we can see that the binomial graph offers either too much reliability, resulting in unnecessary work, or not sufficient reliability.

As an alternative, AllConcur uses $G_S(n,d)$ digraphs, for any $d \geq 3$ and $n \geq 2d$ [150]. In a nutshell, the construction of $G_S(n,d)$ entails constructing the line digraph of a generalized de Bruijn digraph [55] with the self-loops replaced by cycles. A more detailed description follows the steps provided in the original paper [150].

**Construction.** Let $m$ be the quotient and $t$ the remainder of the division of $n$ by $d$, i.e.,

$$n = md + t, m \geq 2.$$

Let $G_B(m,d)$ be a generalized de Bruijn digraph with $m$ vertices and degree $d$, i.e.,

$$\mathcal{V}(G_B(m,d)) = \{0,\ldots,m-1\},$$
$$\mathcal{E}(G_B(m,d)) = \{(u,v) : v = ud + a (\mathrm{mod} m), a = 0,\ldots,d-1\}.$$

Every vertex of $G_B(m,d)$ has at least $\lfloor d/m \rfloor$ self-loops; moreover, at least two vertices (i.e., 0 and $m-1$), have $\lceil d/m \rceil$ self-loops. Therefore, we can remove the self-loops and replaced them with $\lfloor d/m \rfloor$ cycles connecting all the vertices and an additional cycle connecting only the vertices with $\lceil d/m \rceil$ self-loops; we denote the resulting $d$-regular digraph by $G_B^*(m,d)$.

Further, we construct the line digraph of $G_B^*(m,d)$, which we denoted by $L(G_B^*(m,d))$, i.e.,

$$\begin{aligned}\mathcal{V}(L(G_B^*(m,d))) &= \{uv : (u,v) \in \mathcal{E}(G_B^*(m,d))\},\\ \mathcal{E}(L(G_B^*(m,d))) &= \{(uv,wz) : v = w\}.\end{aligned}$$

Note that $L(G_B^*(m,d))$ has $md$ vertices. If $t = 0$, then $L(G_B^*(m,d))$ is the $G_S(n,d)$ digraph. If $t > 0$, then we choose an arbitrary vertex $v \in \mathcal{V}(G_B^*(m,d))$. Let $X = \{x_0,\ldots,x_{d-1}\}$ be a subset of $\mathcal{V}(L(G_B^*(m,d)))$ with $d$ vertices $uv, \forall u \in \mathcal{V}(G_B^*(m,d))$; similarly, let $Y = \{y_0,\ldots,y_{d-1}\}$ be a subset of $\mathcal{V}(L(G_B^*(m,d)))$ with $d$ vertices $vu, \forall u \in \mathcal{V}(G_B^*(m,d))$. Clearly, $X$ and $Y$ exist since $G_B^*(m,d)$ is $d$-regular. Moreover, let

$$M = \{(x,y) : \forall x \in X, \forall y \in Y\}.$$

Clearly, $M$ is a subset of $\mathcal{E}(L(G_B^*(m,d)))$. Then, $G_S(n,d)$ is constructed by adding a set of $t$ vertices, i.e., $W = \{w_0,\ldots,w_{t-1}\}$, to $L(G_B^*(m,d))$ as follows:

$$\begin{aligned}\mathcal{V}(G_S(n,d)) &= \mathcal{V}(L(G_B^*(m,d))) \cup W,\\ \mathcal{E}(G_S(n,d)) &= \mathcal{E}(L(G_B^*(m,d))) \cup \{(w_i,w_j) : i \neq j\}\\ &\quad \bigcup_{i=0}^{t-1} \{(x,w_i),(w_i,y) : x \in X_i, y \in Y_i\} \setminus \bigcup_{i=0}^{t-1} M_i,\end{aligned}$$

where

$$\begin{aligned}M_i &= \{(x_{i+p},y_{i+q}) : q = i + p \pmod{d-t+1}, 0 \leq p \leq d-t\},\\ X_i &= \{x_i,\ldots,x_{i+d-t}\},\\ Y_i &= \{y_i,\ldots,y_{i+d-t}\},\end{aligned}$$

for $i = 0,\ldots,t-1$.

**Properties.** Similarly to binomial graphs [11], $G_S(n,d)$ digraphs are optimally connected. Contrary to binomial graphs though, they can be adapted to various reliability targets (see Figure 5.9 for a reliability target of 6-nines). Moreover, $G_S(n,d)$ digraphs have a quasi-minimal diameter for $n \leq d^3 + d$: The diameter is at most one larger than the lower bound obtained from the Moore bound, i.e.,

$$DL(n,d) = \lceil \log_d (n(d-1)+d) \rceil - 1. \tag{5.6}$$

In addition, $G_S(n,d)$ digraphs have low fault diameter bounds (experimentally verified). Table 5.3 shows the parameters of $G_S(n,d)$ for different number of vertices and 6-nines reliability; the reliability is estimated over a period of 24 hours according to the data from the TSUBAME2.5 system failure history [67], i.e., server $MTTF \approx 2$ years.

### 5.6.5 AllConcur vs. leader-based atomic broadcast

For a theoretical comparison to leader-based atomic broadcast, we consider the leader-based approach described in Section 5.1: To establish total order, $n$ servers use a leader-based

| $G_S(n,d)$ | $D$ | $DL(n,d)$ | $G_S(n,d)$ | $D$ | $DL(n,d)$ | $G_S(n,d)$ | $D$ | $DL(n,d)$ |
|---|---|---|---|---|---|---|---|---|
| $G_S(6,3)$ | 2 | 2 | $G_S(32,4)$ | 3 | 3 | $G_S(256,7)$ | 4 | 3 |
| $G_S(8,3)$ | 2 | 2 | $G_S(45,4)$ | 4 | 3 | $G_S(512,8)$ | 3 | 3 |
| $G_S(11,3)$ | 3 | 2 | $G_S(64,5)$ | 4 | 3 | $G_S(1024,11)$ | 4 | 3 |
| $G_S(16,4)$ | 2 | 2 | $G_S(90,5)$ | 3 | 3 | | | |
| $G_S(22,4)$ | 3 | 3 | $G_S(128,5)$ | 4 | 3 | | | |

TABLE 5.3: The parameters—vertex count $n$, degree $d$ and diameter $D$—of $G_S(n,d)$ for 6-nines reliability (estimated over a period of 24 hours and a server $MTTF \approx 2$ years). The lower bound for the diameter is given by Equation 5.6.

group, such as Paxos [92], as a reliable sequencer (see Figure 5.1a); moreover, all the servers interact directly with the leader. In general, in such a leader-based approach, not all servers need to send a message. This is an advantage over AllConcur, where the early termination mechanism requires every server to send a message. However, for the typical scenarios targeted by AllConcur—the data to be agreed upon is well balanced—we can realistically assume that all servers have a message to send (e.g., distributed ledgers [10]).

**Trade-off between work and total message count.** The work required to A-deliver a message from every server in a leader-based approach is unbalanced. On the one hand, every server sends one message and receives $n-1$ messages, resulting in $O(n)$ work. On the other hand, the leader requires quadratic work, i.e., $\mathcal{O}(n^2)$: it receives one message from every server and it sends every received message to all servers. Note that every message is also replicated, adding a constant amount of work per message.

To avoid overloading a single server (i.e., the leader), AllConcur distributes the work evenly among all servers—every server performs $\mathcal{O}(nd)$ work (see Section 5.6.1). This decrease in complexity comes at the cost of introducing more messages to the network. A leader-based approach introduces $n(n-1)$ messages to the network (not counting the messages needed for replication). In AllConcur, every message is sent $d$ times; thus, the total number of messages in the network is $n^2d$.

**Removing and adding servers.** For both AllConcur and leader-based atomic broadcast, the cost of intentionally removing and adding servers can be hidden by using a two-phase approach similar to the transitional configuration in Raft [130]. Thus, we focus only on the cost of unintentionally removing a server, i.e., a failure. Also, we consider a worst-case analysis—we compare the impact of a leader failure to that of a AllConcur server. The consequence of a leader failure is threefold: (1) every server receives one failure notification; (2) a leader election is triggered; and (3) the new leader needs to reestablish the connections to the $n$ servers. Note that the cost of reestablishing the connection can be hidden if the servers connect from the start to all members of the group. In AllConcur, there is no need for leader election. A server failure causes every server to receive up to $d^2$ failure notifications (see Section 5.6.1). Also, the depth may increase (see Section 5.6.2.2).

**Redundancy.** The amount of redundancy (i.e., $d$) needed by AllConcur is given by the reliability of the agreeing servers. Therefore, $d$ can be seen as a performance penalty for requiring a certain level of reliability—AllConcur provides a trade off between performance and reliability. Using more reliable hardware increases AllConcur's performance. In contrast, in a leader-based deployment, more reliable hardware increases only the performance of

message replication (i.e., less replicas are needed), leaving both the quadratic work and the quadratic total message count unchanged.

## 5.7 Evaluation

We evaluate AllConcur on two production systems: (1) an InfiniBand cluster with 96 nodes; and (2) the Hazel Hen Cray XC40 system (7712 nodes). We refer to the two systems as IB-hsw and XC40, respectively. On both systems, each node has 128GB of physical memory and two Intel Xeon E5-2680v3 12-core CPUs with a base frequency of 2.5GHz. The IB-hsw system nodes are connected through a Voltair 4036 Fabric (40Gbps); each node uses a single Mellanox ConnectX-3 QDR adapter (40GBps). Moreover, each node is running ScientificLinux version 6.4. The XC40 system nodes are connected through the Cray Aries network.

We implemented AllConcur[6] in C; the implementation relies on *libev*, a high-performance event loop library. Each instance of AllConcur is deployed on a single physical node. The nodes communicate via either standard sockets-based TCP or high-performance InfiniBand Verbs (IBV); we refer to the two variants as AllConcur-TCP and AllConcur-IBV, respectively. On the IB-hsw system, to take advantage of the high-performance network, we use TCP/IP over InfiniBand ("IP over IB") for AllConcur-TCP. The failure detector is implemented over unreliable datagrams. To compile the code, we use GCC version 5.2.0 on the IB-hsw system and Cray Programming Environment 5.2.82 on the XC40 system.

We evaluate AllConcur as a distributed agreement algorithm—the non-faulty servers agree on a common set of A-broadcast messages, which are then A-delivered in a deterministic order. Therefore, during the evaluation, we focus on two common performance metrics: (1) the *agreement latency*, i.e., the time needed to agree on a common set; and (2) the *agreement throughput*, i.e., the amount of data agreed upon per second. In addition, we introduce the *aggregated throughput*, a performance metric defined as the agreement throughput times the number of servers.

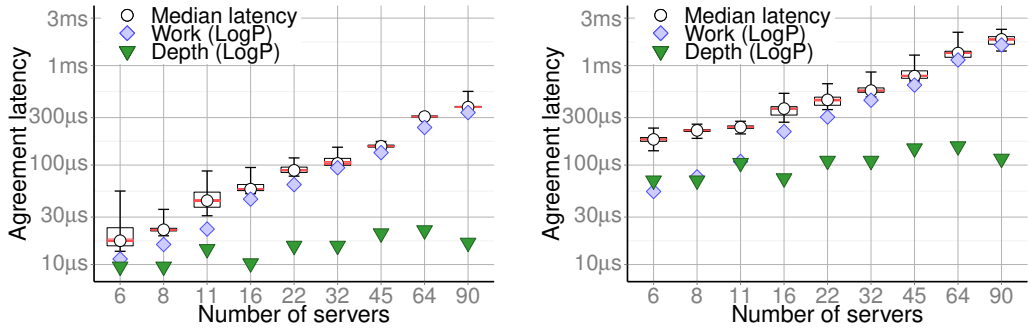
In the following benchmarks, the servers are interconnected via $G_S(n,d)$ digraphs (see Table 5.3). If not specified otherwise, each server generates requests[7] at a certain rate. The requests are buffered until the current agreement round is completed; then, they are packed into a message that is A-broadcast in the next round. All the experiments assume $\mathscr{P}$. An evaluation of the forward-backward mechanism required by $\Diamond\mathscr{P}$ is provided in Chapter 6. All the figures report both the median and the 95% nonparametric confidence interval around it [73]. Moreover, for each figure, the system used to obtain the measurements is specified in square brackets.

### 5.7.1 Single request agreement

To evaluate the LogP models described in Section 5.6, we consider a benchmark where the servers agree on one single request. Clearly, such a scenario is not the intended use case of AllConcur, as all servers, except one, A-broadcast empty messages. Figure 5.10 plots the agreement latency as a function of system size for both AllConcur-IBV and AllConcur-TCP on the IB-hsw system and it compares it with the LogP models for both work and

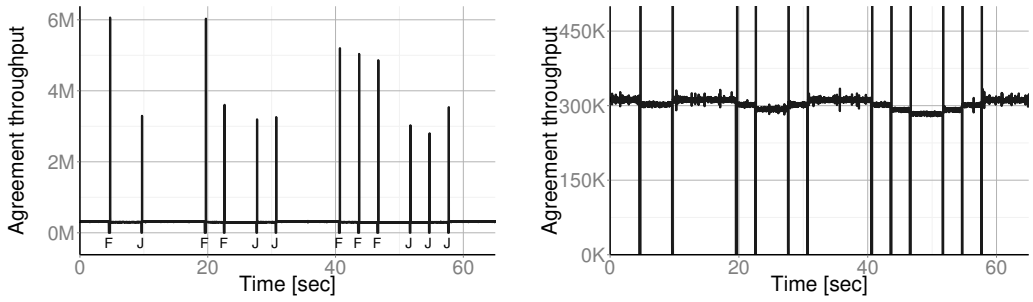
[6] Source code: `https://github.com/mpoke/allconcur`

[7] AllConcur guarantees strong consistency. In particular, we focus on the strong consistency of write requests. For strongly consistent reads, queries also need to be serialized via atomic broadcast. Serializing queries is costly, especially for read-heavy workloads. Typical coordination services [76] relax the consistency model: Queries are performed locally and, hence, can return stale data. AllConcur ensures that a server's view of the shared state cannot fall behind more than one round, i.e., one instance of concurrent atomic broadcast; thus, locally performed queries cannot be outdated by more than one round.




(A) AllConcur-IBV [IB-hsw] (B) AllConcur-TCP [IB-hsw]

FIGURE 5.10: Agreement latency for a single (64-byte) request. The LogP parameters are $L = 1.25\mu s$ and $o = 0.38\mu s$ over IBV and $L = 12\mu s$ and $o = 1.8\mu s$ over TCP.




(A) AllConcur-IBV [IB-hsw] (B) Zoom-in of (a)

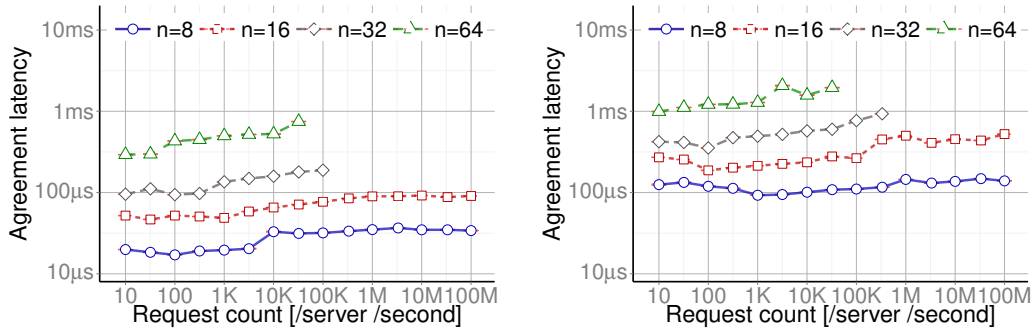
FIGURE 5.11: Agreement throughput during membership changes—servers failing, indicated by F, and servers joining, indicated by J. Deployment over 32 servers, each generating 10,000 (64-byte) requests per second. The failure detector has $\Delta_{hb} = 10ms$ and $\Delta_{to} = 100ms$. The spikes in throughput are due to the accumulated requests during unavailability periods.

depth (§ 5.6). The LogP parameters for the IB-hsw system are $L = 1.25\mu s$ and $o = 0.38\mu s$ over IBV and $L = 12\mu s$ and $o = 1.8\mu s$ over TCP. The models are good indicators of AllConcur's performance; e.g., with increasing the system size, work becomes dominant.

### 5.7.2 Membership changes

To evaluate the effect of membership changes on performance, we deploy AllConcur-IBV on the IB-hsw system. In particular, we consider 32 servers each generating 10,000 (64-byte) requests per second. Servers rely on a heartbeat-based failure detector with a heartbeat period $\Delta_{hb} = 10ms$ and a timeout period $\Delta_{to} = 100ms$. Figure 5.11 shows AllConcur's agreement throughput (binned into $10ms$ intervals) during a series of events, i.e., servers failing, indicated by F, and servers joining, indicated by J. Initially, one server fails, causing a period of unavailability ($\approx 190ms$); this is followed by a rise in throughput, due to the accumulated requests (see Figure 5.11a). Shortly after, the system stabilizes, but at a lower throughput since one server is missing. Next, a server joins the system causing another period of unavailability ($\approx 80ms$) followed by another rise in throughput. Similarly, this scenario repeats for two and three subsequent failures[8]. Note that both unavailability periods can be reduced. First, by improving the failure detector implementation, $\Delta_{to}$ can be significantly decreased [54].

[8]The $G_S(32,4)$ has vertex-connectivity four; therefore, in general, it cannot safely sustain more than three failures




(A) AllConcur-IBV [IB-hsw] (B) AllConcur-TCP [IB-hsw]

FIGURE 5.12: Constant (64-byte) request rate per server.

Second, new servers can join the system as non-participating members until they established all necessary connections [130].

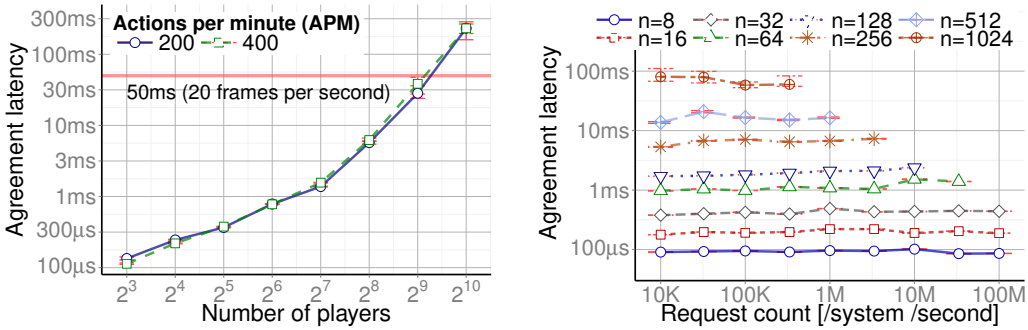
### 5.7.3 Real-world applications

AllConcur enables decentralized coordination services that require strong consistency at high request rates; therefore, it allows for a novel approach to several real-world applications. In this subsection, we evaluate AllConcur using a set of benchmarks, representative of three such applications: (1) travel reservation systems; (2) multiplayer video games; and (3) distributed exchanges.

**Travel reservation systems.** These are typical scenarios where updates are preceded by a large number of local queries, e.g., clients check many flights before choosing a ticket. To increase performance, most existing systems either adopt weaker consistency models, such as eventual consistency [46], or partition the state [152], not allowing transactions spanning multiple partitions. AllConcur offers strong consistency by distributing queries over multiple servers that agree on the entire state.

We consider a benchmark where 64-byte requests are generated with a constant rate per server, i.e., a server's rate of generating requests is bounded by its rate of answering queries. Since the *batching factor* (i.e., the amount of requests packed into a message) is not bounded, the system becomes unstable once the rate of generating requests exceeds the agreement throughput; this leads to a cycle of larger messages, leading to longer times, leading to larger messages etc. A practical deployment would bound the message size and reduce the inflow of requests. Figures 5.12a and 5.12b plot the agreement latency as a function of the rate of generating requests; the measurements were obtained on the IB-hsw system. By using AllConcur-IBV, 8 servers, each generating 100 million requests per second, reach agreement in $35\mu s$; while 64 servers, each generating 32,000 requests per second, reach agreement in less than $0.75ms$. AllConcur-TCP has $\approx 3\times$ higher latency.

**Multiplayer video games.** These are an example of applications where the shared state satisfies two conditions—it is too large to be frequently transferred through the network and it is periodically updated. For example, modern video games update the state once every $50ms$ (i.e., 20 frames per second) by only sending changes since the previous state [17, 18]. Therefore, such applications are latency sensitive [16]. To decrease latency, existing systems either limit the number of players, e.g., $\approx 8$ players in real time strategy games, or limit the players' view to only a subset of the state, such as the area of interest in first person shooter




(A) AllConcur-TCP [XC40] (B) AllConcur-TCP [XC40]

FIGURE 5.13: (a) Agreement latency in multiplayer video games for different APM and 40-byte requests. (b) Constant (40-byte) request rate per system.

games [17, 18]. AllConcur allows hundreds of servers to share a global state view at low latency.

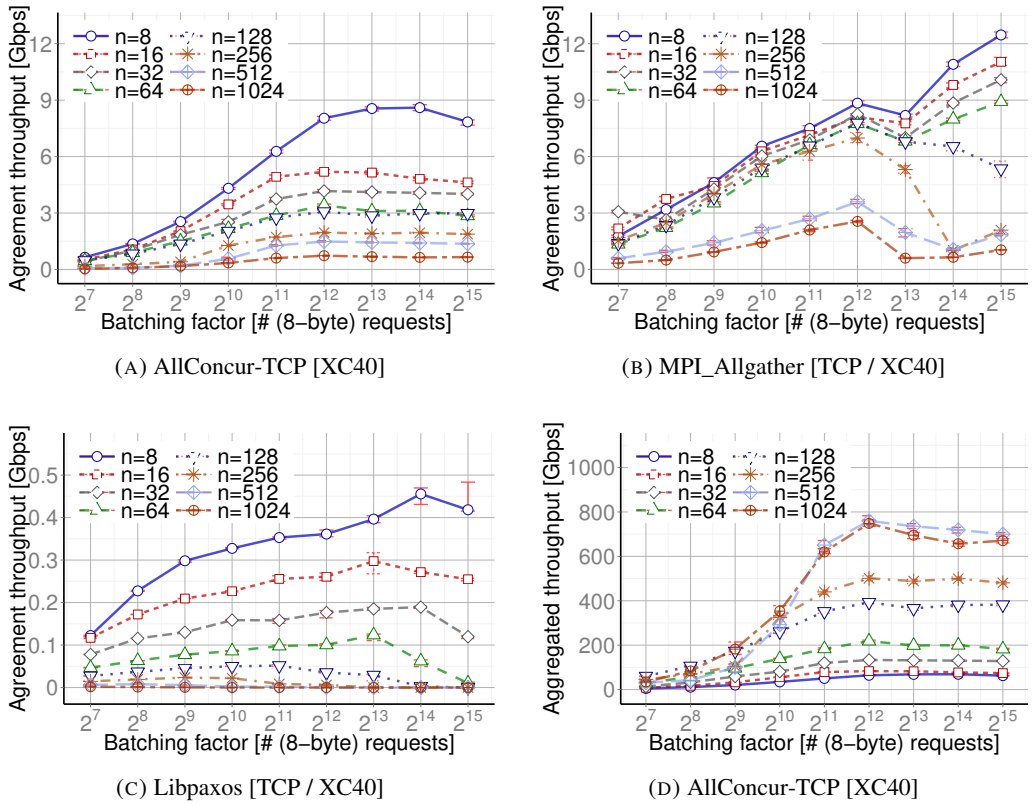
To emulate such a scenario, we deploy AllConcur on the XC40 system; although not designed for video games, the system enables large-scale deployments. Similarly to travel reservation systems, each server's rate of generating requests is bounded: In multiplayer video games, each player performs a limited number of actions per minute (APM), i.e., usually 200 APM, although expert players can exceed 400 APM [103]. Each action causes a state update with a typical size of 40 bytes [17]. Figure 5.13a plots the agreement latency as a function of the number of players, for 200 and 400 APM. AllConcur-TCP supports the simultaneous interaction among 512 players with an agreement latency of 28*ms* for 200 APM and 38*ms* for 400 APM. Therefore, AllConcur enables so called epic battles [22].

**Distributed exchanges.** These are a typical example of systems where fairness is essential. For example, to connect to an exchange service, such as the New York Stock Exchange, clients must obtain so called co-locations (i.e., servers with minimal latency to the exchange). To ensure fairness, such co-locations are usually standardized; every client subscribing to the same co-location service has the same latency, ensured by standardized hardware [148]. Therefore, centralized systems cannot support geographically distributed clients. AllConcur enables the deployment of exchange services over geographically distributed servers: As long as all clients have an equal latency to any of the servers, fairness is provided.

To emulate such a scenario, we deploy AllConcur on the XC40 system; although this system is not geographically distributed, we believe the results are good indicators of AllConcur's behavior at large scale. In distributed exchanges, the servers are handling a globally constant rate of requests. Therefore, in Figure 5.13b, we plot the agreement latency as a function of the system's rate of generating request. An AllConcur-TCP deployment across 8 servers handles 100 million (40-byte) requests per second with latencies below 90$\mu s$; while, across 512 servers, it can handle one million requests per second with latencies below 20*ms*. The 4$\times$ increase in agreement latency for 1,024 servers is due to the 11$\times$ redundancy of the $G_S$ digraph, necessary for achieving our reliability target of 6-nines (see Table 5.3).

### 5.7.4 Comparison to other algorithms

In this subsection, we compare AllConcur's throughput with the throughput achieved by other agreement algorithms. First, we compare AllConcur against an unreliable approach; second, we compare AllConcur against a state of the art atomic broadcast algorithm.




(A) AllConcur-TCP [XC40]

(B) MPI_Allgather [TCP / XC40]

(C) Libpaxos [TCP / XC40]

(D) AllConcur-TCP [XC40]

FIGURE 5.14: (a) AllConcur vs. (b) unreliable agreement vs. (c) leader-based agreement—batching factor effect on the agreement throughput. (d) Batching factor effect on the aggregated throughput.

**AllConcur vs. unreliable agreement.** To evaluate the overhead of providing fault tolerance, we compare AllConcur to an implementation of unreliable agreement. In particular, we use MPI_Allgather [123] to disseminate all messages to every server. We consider a benchmark where every server A-broadcasts a fixed-size message per round (fixed number of requests). Figures 5.14b and 5.14a plot the agreement throughput as a function of the batching factor (i.e., the amount of requests packed into a message). The measurements were obtained on the XC40 system; for a fair comparison, we used Open MPI [63] over TCP to run the benchmark. AllConcur provides a reliability target of 6-nines with an average overhead of 58%. Moreover, for messages of at least 2,048 (8-byte) requests, the overhead does not exceed 75%.

**AllConcur vs. leader-based agreement.** We conclude AllConcur's evaluation by comparing it to Libpaxos [147], an open-source implementation of Paxos [92] over TCP. In particular, we use Libpaxos as the leader-based group in the deployment described in Section 5.1. The size of the Paxos group is five, sufficient for our reliability target of 6-nines. We consider the same benchmark used to compare to unreliable agreement—every server A-broadcasts a fixed-size message per round. In the case of Libpaxos, every server has one outstanding A-broadcast message, i.e., before A-broadcasting another message, it waits for the message to be A-delivered. Figures 5.14a and 5.14c plot the agreement throughput as a function of the batching factor; the measurements were obtained on the XC40 system. The throughput peaks at a certain message size, indicating the optimal batching factor to be used. AllConcur-TCP reaches an agreement throughput of 8.6*Gbps*, equivalent to $\approx$ 135 million (8-byte) requests per second (see Figures 5.14a). As compared to Libpaxos, AllConcur achieves at least 17$\times$

higher throughput (see Figure 5.14c). The drop in throughput (after reaching the peak), for both AllConcur and Libpaxos, is due to the TCP congestion control mechanism.

AllConcur's agreement throughput decreases with increasing the number of servers. The reason for this performance drop is twofold. First, to maintain the same reliability, more servers entail a higher degree for $G$ (see Table 5.3) and therefore, more redundancy. Second, agreement among more servers entails more synchronization. However, the number of participating servers is application-dependent. Thus, a better metric to measure AllConcur's actual performance is the aggregated throughput. Figure 5.14d plots the aggregated throughput corresponding to the agreement throughput from Figures 5.14a. AllConcur-TCP's aggregated throughput increases with the number of servers and it peaks at $\approx 750 Gbps$ for 512 and 1,024 servers.

# Chapter 6

# AllConcur+

The digraph-based communication model of AllConcur requires a resilient digraph in order to reliably disseminate messages. However, this resiliency comes at the cost of redundancy, which imposes a lower bound on the work per A-broadcast message. In this chapter, we present AllConcur+, a leaderless concurrent atomic broadcast algorithm that adopts a dual digraph approach, with the aim of enabling redundancy-free state-machine replication during intervals with no failures. Servers in AllConcur+ communicate via two overlay networks described by two digraphs—an *unreliable* digraph, with a vertex-connectivity of one, and a *reliable* digraph, with a vertex-connectivity larger than the maximum number of tolerated failures. The unreliable digraph enables minimal-work distributed agreement during intervals with no failures—every server both receives and sends every broadcast message at most once. When failures do occur, AllConcur+ falls back to the reliable digraph. Moreover, the fault tolerance is given by the reliable digraph's vertex-connectivity and can be adapted to system-specific requirements. Thus, similar to AllConcur, AllConcur+ trades off reliability against performance. Our evaluation at the end of this chapter is based on OMNeT++, a discrete-event simulator [156]. When no failures occur, AllConcur+ outperforms other state of the art atomic broadcast algorithms both in terms of throughput and latency. Moreover, an analysis of failure scenarios shows that AllConcur+'s expected performance is robust with regard to occasional failures.

The remainder of this chapter states the problem in Section 6.1; presents the dual digraph approach by both describing the two modes of execution and specifying how to safely transition from one mode to another in Section 6.2; provides a thorough description of the design of AllConcur+ in Section 6.3; shows AllConcur+'s correctness through an informal proof in Section 6.4; and evaluates the performance of AllConcur+ in Section 6.5.

Most of the content of this chapter is reproduced from the following paper:

- Marius Poke, Colin W. Glass. *A Dual Digraph Approach for Leaderless Atomic Broadcast (Extended Version)* [136]

## 6.1 Problem statement

As seen in Chapter 5, by distributing the workload evenly among all servers, AllConcur enables high-performance state-machine replication, while scaling out to hundreds of servers. A key concept of AllConcur is that servers exchange messages through an overlay network described by a regular and sparse digraph. As a consequence, the workload is evenly balanced among the participating servers and the work per broadcast message is sublinear. Moreover, for the message dissemination to be reliable (while tolerating up to $f$ failures), the digraph must also be resilient, i.e., its vertex-connectivity must exceed $f$. The resiliency of the digraph comes at the cost of redundancy. Every server receives each message from all its predecessors and sends each message to all its successors. Therefore, for a regular digraph with degree $d$, the amount of messages introduced into the network increases by a factor of $d$.

This redundancy introduces a lower bound on the work per broadcast message, i.e., although sublinear, the work is bounded by $\Omega(d)$.

The resilience is unavoidable: Given the frequency of failures in today's distributed systems [67, 70, 52], providing consistency through non-fault-tolerant algorithms would be unfeasible. Nonetheless, for many real-world systems, intervals with no failures are common enough that the constant overhead of redundancy leads to suboptimal performance. For instance, consider the multiplayer video game scenario from AllConcur's evaluation, where 512 players perform each 400 actions per minute, with an action causing a 40 bytes state update (see Section 5.7). As a result, an AllConcur rounds takes around 38*ms* to complete (see Figure 5.13a). If we assume the players are distributed across a system with a mean time between failures per server of around 2 years [67], then, there is a 99.9% probability of no failures occurring during an interval of two minute. This means that AllConcur conservatively provides resiliency during more than 3,000 rounds, even though there is a high chance of no failures to happen. AllConcur+ is designed to leverage redundancy-free agreement during these intervals with no failures, thereby pushing the limits of scalable, high-performance state-machine replication even further.

## 6.2 A dual digraph approach

AllConcur+ switches between two modes—unreliable and reliable. During intervals with no failures, AllConcur+ uses the unreliable mode, which enables minimal-work distributed agreement. When failures do occur, AllConcur+ automatically switches to the reliable mode, that uses the early termination mechanism of AllConcur (see Section 5.2.1). In this section, we first describe the two modes (§ 6.2.1). Then, we model AllConcur+ as a state-machine, specifying the possible transitions from one mode to another (§ 6.2.2). Afterwards, we discuss the conditions necessary to A-deliver messages (§ 6.2.3). Finally, we reason about the possible concurrent states AllConcur+ servers can be in (§ 6.2.4).

### 6.2.1 Round-based algorithm

AllConcur+ is a round-based algorithm. Based on the two modes, it distinguishes between *unreliable* and *reliable* rounds. Every round is described by its round number $r$. In round $r$, every server A-broadcasts a (possibly empty) message and collects (in a set) all the messages received for this round (including its own). The goal is for all non-faulty servers to eventually agree on a common set of messages; we refer to this as the *set agreement* property (see Section 5.4). Then, all non-faulty servers A-deliver the messages in the common set in a deterministic order. For brevity, we say a round is A-delivered if its messages are A-delivered.

**Unreliable rounds.** Unreliable rounds enable minimal-work distributed agreement while no failures occur. The overlay network is described by an unreliable digraph $G_U$ with vertex-connectivity $\kappa(G_U) = 1$ (see Table 2.1 for digraph notations). A server completes an unreliable round once it has received a message from every server (A-broadcast in that round). Since $G_U$ is connected (i.e., $\kappa(G_U) = 1$), the completion of an unreliable round is guaranteed under the condition of no failures. To ensure set agreement holds, the completion of an unreliable round does not directly lead to that round being A-delivered. Yet, completing two successive unreliable rounds guarantees the first one can be A-delivered (see Section 6.2.3 for details).

**Reliable rounds.** Receiving a failure notification in an unreliable round triggers both a rollback to the latest A-delivered round and a switch to the reliable mode. Therefore, the first

round that has not yet been A-delivered (i.e., succeeding the latest A-delivered round) is rerun reliably. The overlay network is described by a reliable digraph $G_R$ with vertex-connectivity $\kappa(G_R) > f$. To complete a reliable round, every server uses AllConcur's early termination mechanism. In addition to completion, early termination guarantees set agreement (see Section 5.4). Thus, once a server completes a reliable round, it can safely A-deliver it (§ 6.2.3). In summary, a rollback entails (reliably) rerunning unreliable rounds, which have not been previously A-delivered. For validity to hold, it is necessary for the same messages to be A-broadcast when a round is rerun. In practice, this requirement can be dropped.

Similar to AllConcur, once a reliable round completes, all servers for which no messages were A-delivered are removed. Therefore, all non-faulty servers have a consistent view of the system (similar to group membership services [33, 43]). After removing servers from the system, every non-faulty server needs to update both $G_U$, in order for it to be connected, and the set $F$ of received failure notifications; also, the servers may choose to update $G_R$. We defer the discussion on updating digraphs to Sections 6.3.1 and 6.3.3.2, respectively. Updating $F$ entails removing all the *invalid failure notifications*—notifications that are targeting, or were detected by, removed servers. Invalid notifications are no longer required for further message tracking, since removed servers cannot be suspected of having any messages.

### 6.2.2 State machine approach

It is common to describe a distributed algorithm as a state machine: Every server's behavior is defined by an ordered sequence of states. In the case of AllConcur+, as long as no failures occur, it is an ordered sequence of unreliable rounds, i.e., with strictly increasing round numbers. This sequence consists of a sub-sequence of A-delivered rounds followed (potentially) by a round not yet A-delivered (§ 6.2.3) and by the ongoing round. The occurrence of a failure leads to a rollback. This breaks the order of the sequence, i.e., the round numbers are no longer strictly increasing. To enforce a strict order, we introduce the concept of an *epoch*. The definition of an epoch is threefold: (1) it starts with a reliable round; (2) it contains only one completed reliable round; and (3) it contains at most one sequence of unreliable rounds. An epoch is described by an epoch number $e$, which corresponds to the number of reliable rounds completed so far, plus the ongoing round, if it is reliable.

The state of each server is defined by the epoch number, the round number, the round type and, for unreliable rounds, the type of the previous round. We denote states with unreliable rounds by $[e,r]$ and states with reliable rounds by $[\![e,r]\!]$. Also, when necessary, we use $\triangleright$ to indicate the first unreliable round following a reliable round. Initially, all servers are in state $[\![1,0]\!]$, essentially assuming a reliable round 0 has already been completed, without any messages being A-broadcast. State $[\![1,0]\!]$ is required by the definition of an epoch.

Servers can move from one state to another; we define three types of state transitions.

**No-fail transitions.** Denoted by $\rightarrow$, no-fail transitions move servers to the next unreliable round without increasing the epoch. We identify two no-fail transitions:

$$(\mathrm{T_{UU}}) \quad [e,r] \rightarrow [e,r+1]; \qquad\qquad (\mathrm{T_{R\triangleright}}) \quad [\![e,r]\!] \rightarrow [e,r+1]_{\triangleright}.$$

$\mathrm{T_{UU}}$ continues a sequence of unreliable rounds, while $\mathrm{T_{R\triangleright}}$, in the absence of failure notifications, starts a sequence of unreliable rounds. Note that the initial state $[\![1,0]\!]$ is always followed by a $\mathrm{T_{R\triangleright}}$ transition.

**Fail transitions.** Denoted by $\xrightarrow{\text{fail}}$, fail transitions move servers to a reliable round while increasing the epoch. Fail transitions are caused by failure notifications: In unreliable rounds, any failure notification immediately triggers a fail transition. In reliable rounds, remaining

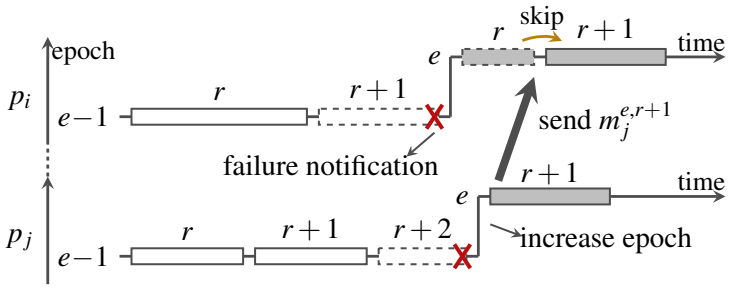



FIGURE 6.1: Skip transition (i.e., $T_{Sk}$). Empty rectangles indicate unreliable rounds; filled (gray) rectangles indicate reliable rounds. Rectangles with solid edges indicate completed rounds, while dashed edges indicate interrupted rounds.

failure notifications that are still valid at the end of the round result in a fail transition to the next round. Overall, we identify three fail transitions:

$$(T_{UR}) \quad [e,r] \xrightarrow{\text{fail}} [[e+1,r-1]]; \qquad (T_{\triangleright R}) \quad [e,r]_{\triangleright} \xrightarrow{\text{fail}} [[e+1,r]];$$
$$(T_{RR}) \quad [[e,r]] \xrightarrow{\text{fail}} [[e+1,r+1]].$$

$T_{UR}$ and $T_{\triangleright R}$ both interrupt the current unreliable round and rollback to the latest A-delivered round. The difference is $T_{UR}$ is preceded by $T_{UU}$, while $T_{\triangleright R}$ is preceded by $T_{R\triangleright}$, and therefore, the latest A-delivered round differs (§ 6.2.3). $T_{RR}$ continues a sequence of reliable rounds, due to failure notifications in $F$ that remain valid at the end of the round and therefore cannot be removed (§ 6.2.1).

**Skip transitions.** Denoted by $\xrightarrow{\text{skip}}$, skip transitions move servers to a reliable round without increasing the epoch, i.e.,

$$(T_{Sk}) \quad [[e,r]] \xrightarrow{\text{skip}} [[e,r+1]].$$

A server $p_i$ performs a skip transition if it receives, in a reliable round, a message A-broadcast by a server $p_j$ in the same epoch, but in a subsequent reliable round. This happens only if $p_j$ had one more $T_{UU}$ transition than $p_i$ before receiving the failure notification that triggered the fail transition to the current epoch. Figure 6.1 illustrates the skip transition of $p_i$ after receiving $m_j^{e,r+1}$, the message A-broadcast by $p_j$ while in state $[[e,r+1]]$. This message indicates (to $p_i$) that $p_j$ rolled back to round $r$, the latest A-delivered round; thus, it is safe (for $p_i$) to also A-deliver $[e,r]$ and *skip* to $[[e,r+1]]$. Note that $[[e,r]]$ is not completed (we say a state is completed or A-delivered, if its round is completed or A-delivered, respectively).

In summary, $T_{UU}$ and $T_{R\triangleright}$ lead to unreliable rounds, while $T_{UR}$, $T_{\triangleright R}$, $T_{RR}$ and $T_{Sk}$ lead to reliable rounds.

### 6.2.3 A-delivering messages

In Figure 6.1, $p_j$ already A-delivered round $r$ when it receives the failure notification that triggers the transition from $[e-1,r+2]$ to $[[e,r+1]]$. To explain the intuition behind $p_j$ A-delivering $r$, we first introduce the following proposition:

**Proposition 6.2.1.** *A necessary condition for a server to complete a state is for all non-faulty servers to start the state.*

*Proof.* For a server to complete a state (with either an unreliable or a reliable round), it must receive from every non-faulty server a message A-broadcast in that state (§ 6.2.1). □

Since $p_j$ started $[e-1, r+2]$, it completed $[e-1, r+1]$ and thus, every non-faulty server started $[e-1, r+1]$ (cf. Proposition 6.2.1). This entails that every non-faulty server completed $[e-1, r]$. Consequently, $p_j$ knows round $r$ was safely completed by all non-faulty servers and it can A-deliver it. Moreover, in the example, the message $m_j^{e,r+1}$ A-broadcast by $p_j$ while in state $[\![e, r+1]\!]$ carries (implicitly) the information that all non-faulty servers completed $[e-1, r]$. As a result, upon receiving $m_j^{e,r+1}$, $p_i$ can also A-deliver $[e-1, r]$.

In general, an AllConcur+ server can A-deliver an unreliable round $r$ in two ways: (1) it completes the subsequent unreliable round $r+1$; or (2) after it interrupts the subsequent round $r+1$ due to a failure notification, it receives a message A-broadcast by another server in the same epoch, but in a subsequent reliable round, i.e., a skip transition. In other words, a server A-delivers $[e, r]$, if one of the following transition sequences occurs:

$$[e, r] \to [e, r+1] \to [e, r+2];$$
$$[e, r] \to [e, r+1] \xrightarrow{\text{fail}} [\![e+1, r]\!] \xrightarrow{\text{skip}} [\![e+1, r+1]\!].$$

Reliable rounds use early termination; therefore, they can be A-delivered directly after completion (i.e., before either $\mathrm{T}_{\mathrm{R}\triangleright}$ or $\mathrm{T}_{\mathrm{RR}}$). In other words, a server A-delivers $[\![e, r]\!]$, if one of the following transition sequences occurs:

$$[\![e, r]\!] \to [e, r+1]_{\triangleright};$$
$$[\![e, r]\!] \xrightarrow{\text{fail}} [\![e+1, r+1]\!].$$

### 6.2.4 Concurrent states

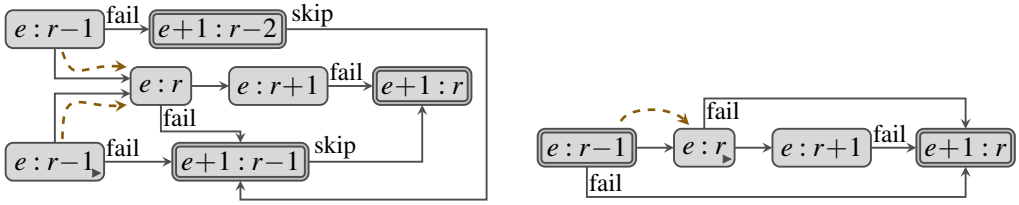
The necessary condition stated in Proposition 6.2.1 enables us to reason about the possible concurrent states, i.e., the states a non-faulty server $p_i$ can be, given that a non-faulty server $p_j$ is either in $[e, r]$ or in $[\![e, r]\!]$ (§ 6.2.4.1). Knowing the possible concurrent states enables us to deduce a set of properties that aid the design of AllConcur+ (§ 6.2.4.2).

#### 6.2.4.1 Description of concurrent states

We are interested in what states a non-faulty server $p_i$ can be, given that a non-faulty server $p_j$ is either in $[e, r]$ or in $[\![e, r]\!]$.

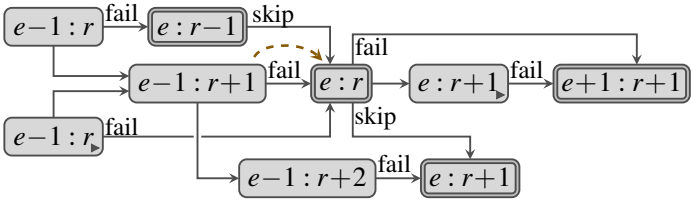
**Server $p_j$ is in an unreliable round.** Let $p_j$ be in $[e, r]$. Then, $[e, r]$ is preceded by either $\mathrm{T}_{\mathrm{UU}}$ or $\mathrm{T}_{\mathrm{R}\triangleright}$ (see the curved dashed arrows in Figures 6.2a and 6.2b). In both cases, $p_j$ completed the previous state; therefore, $\mathrm{T}_{\mathrm{UU}}$ entails $p_i$ has started $[e, r-1]$, while $\mathrm{T}_{\mathrm{R}\triangleright}$ entails $p_i$ has started $[\![e, r-1]\!]$ (cf. Proposition 6.2.1). Using both this information and the fact that $p_j$ already started $[e, r]$, we can deduce all the possible states of $p_i$ (see the boxes in Figures 6.2a and 6.2b).

Since $p_j$ started $[e, r]$, $p_i$ can be in one of the following states with unreliable rounds: $[e, r-1]$ (if $p_j$ is not in $[e, r]_{\triangleright}$); $[e, r]$; or $[e, r+1]$. Moreover, if $p_i$ receives a failure notification, it moves (from any of these states) to a state with a reliable round (indicated by double-edged boxes). Note that a fail transition from $[e, r-1]$ depends on whether round $r-1$ is the first in a sequence of unreliable rounds (see the lower-right corner triangle in Figure 6.2a). Finally, $p_i$ can skip a reliable round, i.e., either $r-2$ or $r-1$ (see Figure 6.2a); the precondition is for at least one other server to have A-delivered the corresponding unreliable round. Notice though that $p_i$ cannot skip both rounds—this would imply both $[e, r-2]$ and $[e, r-1]$ to be A-delivered, which is not possible since $p_i$ does not start $[e, r]$.

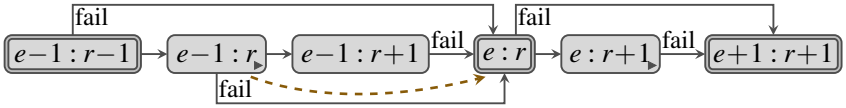



(A) $p_j$ in $[e,r]$ (B) $p_j$ in $[e,r]_{\triangleright}$

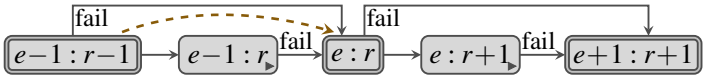
FIGURE 6.2: The possible states of a non-faulty server $p_i$, while a non-faulty server $p_j$ is in epoch $e$ and unreliable round $r$. Boxes indicate states—single-edged for unreliable rounds and double-edged for reliable rounds. A triangle in the lower-right corner indicates the first state in a sequence of unreliable rounds. Straight arrows indicate $p_i$'s possible transitions; curved dashed arrows indicate $p_j$'s latest transition.



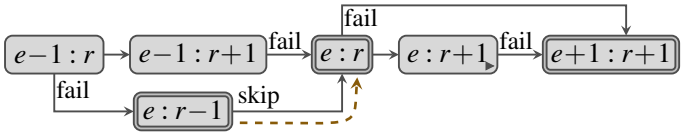

(A) $p_j$'s $[[e,r]]$ state is preceded by $\mathrm{T_{UR}}$




(B) $p_j$'s $[[e,r]]$ state is preceded by $\mathrm{T_{\triangleright R}}$




(C) $p_j$'s $[[e,r]]$ state is preceded by $\mathrm{T_{RR}}$




(D) $p_j$'s $[[e,r]]$ state is preceded by $\mathrm{T_{Sk}}$

FIGURE 6.3: The possible states of a non-faulty server $p_i$, while a non-faulty server $p_j$ is in epoch $e$ and reliable round $r$. Boxes indicate states—single-edged for unreliable rounds and double-edged for reliable rounds. A triangle in the lower-right corner indicates the first state in a sequence of unreliable rounds. Straight arrows indicate $p_i$'s possible transitions; curved dashed arrows indicate $p_j$'s latest transition.

**Server $p_j$ is in a reliable round.** Let $p_j$ be in $[[e,r]]$. Then, $p_i$ can clearly be in $[[e,r]]$ as well; moreover, since $p_j$ started $[[e,r]]$, $p_i$ can be ahead in either $[e,r+1]_{\triangleright}$ or $[[e+1,r+1]]$. To infer the other possible states of $p_i$, we consider the four transitions that can precede $p_j$'s $[[e,r]]$ state (see the curved dashed arrows in Figures 6.3a, 6.3b, 6.3c, and 6.3d). Note that $[[e,r]]$, $[e,r+1]_{\triangleright}$, and $[[e+1,r+1]]$ are present in all four figures.

$\mathrm{T_{UR}}$ entails $p_j$ has started $[e-1,r+1]$ and hence, $p_i$ started either $[e-1,r]$ or $[e-1,r]_{\triangleright}$ (cf. Proposition 6.2.1). Receiving a failure notification while in either of these states triggers

(eventually) a transition to $[\![e,r]\!]$ (either directly from $[e-1,r]_\triangleright$ or through a skip transition from $[e-1,r]$ via $[\![e,r-1]\!]$). Moreover, if the failure notification is delayed, $p_i$ can move to $[e-1,r+1]$ or even to $[e-1,r+2]$. Finally, receiving a failure notification while in $[e-1,r+2]$ triggers a transition to $[\![e,r+1]\!]$; $[\![e,r+1]\!]$ can also be reached by a skip transition from $[\![e,r]\!]$ (see Figure 6.1). Both $\mathrm{T}_{\triangleright\mathrm{R}}$ and $\mathrm{T}_{\mathrm{RR}}$ entail $p_i$ has started $[\![e-1,r-1]\!]$, since $p_j$ completed it (cf. Proposition 6.2.1). In addition, $\mathrm{T}_{\triangleright\mathrm{R}}$ entails $p_j$ started $[e-1,r]$ before moving to $[\![e,r]\!]$; therefore, in this case, $p_i$ can also start $[e-1,r+1]$ (after completing $[e-1,r]$). $\mathrm{T}_{\mathrm{Sk}}$ entails at least one server completed $[e-1,r]$ (see Figure 6.1) and thus, $p_i$ has started $[e-1,r]$. Note that the state transitions illustrated in Figure 6.3d are also included in Figure 6.3a.

In summary, while $p_j$ is in $[e,r]$, $p_i$ can be in four states with unreliable rounds, i.e.,

$$(1)\ [e,r-1] \qquad (2)\ [e,r-1]_\triangleright \qquad (3)\ [e,r] \qquad (4)\ [e,r+1]\,,$$

and in four states with reliable rounds, i.e.,

$$(1)\ [\![e,r-1]\!] \qquad (2)\ [\![e+1,r-2]\!] \qquad (3)\ [\![e+1,r-1]\!] \qquad (4)\ [\![e+1,r]\!]\,.$$

While $p_j$ is in $[\![e,r]\!]$, $p_i$ can be in five states with unreliable rounds, i.e.,

$$(1)\ [e-1,r] \qquad (2)\ [e-1,r]_\triangleright \qquad (3)\ [e-1,r+1] \qquad (4)\ [e-1,r+2] \qquad (5)\ [e,r+1]_\triangleright\,,$$

and in five states with reliable rounds, i.e.,

$$(1)\ [\![e-1,r-1]\!] \qquad (2)\ [\![e,r-1]\!] \qquad (3)\ [\![e,r]\!] \qquad (4)\ [\![e,r+1]\!] \qquad (5)\ [\![e+1,r+1]\!]\,.$$

#### 6.2.4.2 Concurrency properties

The following propositions facilitate the design of of AllConcur+ (see Section 6.3). In a nutshell, Proposition 6.2.2 asserts the uniqueness of a state, i.e., a state is uniquely identified by only its epoch and round; Propositions 6.2.3, 6.2.5, and 6.2.6 specify what messages a non-faulty server can receive.

**Proposition 6.2.2.** *Let $p_i$ and $p_j$ be two non-faulty servers, both in epoch $e$ and round $r$. Then, both are either in $[e,r]$ or in $[\![e,r]\!]$.*

*Proof.* W.l.o.g., we assume $p_i$ is in state $[e,r]$ and $p_j$ is in state $[\![e,r]\!]$. Clearly, $p_i$'s state $[e,r]$ is preceded by either a $\mathrm{T}_{\mathrm{UU}}$ transition (and thus, by $[e,r-1]$) or a $\mathrm{T}_{\mathrm{R}\triangleright}$ transition (and thus, by $[\![e,r-1]\!]$). This entails, $p_j$ at least started either $[e,r-1]$ or $[\![e,r-1]\!]$ (cf. Proposition 6.2.1). Moreover, $p_j$'s state $[\![e,r]\!]$ can only be preceded by either a fail or a skip transition. However, a fail transition from either $[e,r-1]$ or $[\![e,r-1]\!]$ is not possible since it would entail an increase in epoch; also, a skip transition cannot be preceded by an unreliable round. Therefore, $p_j$ went through $[\![e,r-1]\!] \xrightarrow{\text{skip}} [\![e,r]\!]$ and, as a result, $p_i$ went through $[\![e,r-1]\!] \to [e,r]$.

The skip transition requires at least one non-faulty server to have the following fail transition $[e-1,r+1] \xrightarrow{\text{fail}} [\![e,r]\!]$ (see Figure 6.1). Let $p_k$ be such a server; clearly, $p_k$ has not started $[\![e,r-1]\!]$. Yet, this contradicts $p_i$'s transition $[\![e,r-1]\!] \to [e,r]$ (cf. Proposition 6.2.1). □

**Proposition 6.2.3.** *Let $p_i$ and $p_j$ be two non-faulty servers. Let $m_j^{(e,r)}$ be the message A-broadcast by $p_j$ while in $[e,r]$ and received by $p_i$ in epoch $\widehat{e}$ and round $\widehat{r}$. Then, $\widehat{e} \geq e$. Also, $(\widehat{e} = e \wedge \widehat{r} < r) \Rightarrow \widehat{r} = r-1$.*

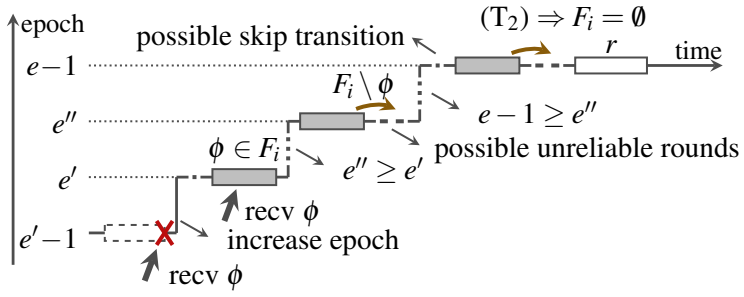



FIGURE 6.4: A non-faulty server $p_i$ is in state $[e-1,r]$; $p_i$ received a failure notification $\phi$ in a preceding state. While in the completed reliable round of epoch $e'$, $\phi \in F_i$; after the reliable round of epoch $e''$ is completed, $\phi$ is removed from $F_i$; after the reliable round of epoch $e-1$ is completed, $F_i = \emptyset$. If $e' = e'' = e-1$, the three reliable rounds refer to the same actual round. Empty rectangles indicate unreliable rounds; filled (gray) rectangles indicate reliable rounds. Rectangles with solid edges indicate completed round, while with dashed edges indicate interrupted rounds.

*Proof.* First, we assume $\widehat{e} < e$. Clearly, $e \neq 1$. Therefore, $p_j$'s state $[e,r]$ is preceded by a state $[\![e,r' < r]\!]$ that increased the epoch to $e$. This entails $p_i$ started state $[\![e,r']\!]$ (cf. Proposition 6.2.1). Moreover, $p_i$ started this state before $p_j$ sent $m_j^{(e,r)}$. Yet, this contradicts the assumption that $p_i$ received the message in epoch $\widehat{e} < e$.

Then, we assume $\widehat{e} = e \wedge \widehat{r} < r \not\Rightarrow \widehat{r} = r-1$; consequently, $\widehat{e} = e$ and $\widehat{r} < r-1$. However, $p_j$'s state $[e,r]$ is preceded by either $[e,r-1]$ or $[\![e,r-1]\!]$. In both cases, $p_i$ was in epoch $e$ and round $r-1$ before receiving $m_j^{(e,r)}$ in epoch $e$ and round $\widehat{r}$, which contradicts $\widehat{r} < r-1$. □

To prove Proposition 6.2.5 we introduce the following lemma:

**Lemma 6.2.4.** *Let $p_i$ be a non-faulty server in state $[e-1,r]$; let $p_j$ be another non-faulty server in epoch $e$. Let $\phi(a,b)$ be a notification sent by $b$ indicating $a$'s failure; $\phi(a,b)$ triggered $p_j$'s transition from epoch $e-1$ to $e$. Then, $p_i$ cannot receive $\phi(a,b)$ in any state that precedes $[e-1,r]$.*

*Proof.* We assume $p_i$ receives $\phi(a,b)$ in a state preceding $[e-1,r]$ (see Figure 6.4). Let $[\![e',*]\!]$, with $e' \leq e-1$ and $*$ a placeholder for any round, be either the state in which $p_i$ receives $\phi(a,b)$ or the state in which $p_i$ moves after receiving $\phi(a,b)$ (i.e., a fail transition). Clearly, while $p_i$ is in $[\![e',*]\!]$, $\phi(a,b) \in F_i$. Moreover, $[e-1,r]$ must be preceded by a reliable round followed by a $T_{R\triangleright}$ transition, which requires $F_i = \emptyset$ (§ 6.2.2). Thus, in the sequence of states from $[\![e',*]\!]$ to $[e-1,r]$ there is a state $[\![e'',*]\!]$, with $e' \leq e'' \leq e-1$, whose completion results in the removal of $\phi(a,b)$ from $F_i$. This means that when $p_i$ A-delivers state $[\![e'',*]\!]$, it removes from the system at least one of the servers $a$ and $b$ (§ 6.2.1). Since $p_i$ completes $[\![e'',*]\!]$ and $e'' < e$, $p_j$ also completes (and A-delivers) $[\![e'',*]\!]$. Thus, $p_j$ removes the same servers since set agreement is ensured by AllConcur's early termination mechanism. Yet, this leads to a contradiction—$\phi(a,b)$ was invalidated during epoch $e-1$ and therefore, cannot trigger $p_j$'s transition to epoch $e$. □

**Proposition 6.2.5.** *Let $p_i$ and $p_j$ be two non-faulty servers. Let $m_j^{(e,r)}$ be the message A-broadcast by $p_j$ while in $[\![e,r]\!]$ and received by $p_i$ in epoch $\widehat{e}$ and round $\widehat{r}$. Then, $\widehat{e} < e \Rightarrow (\widehat{e} = e-1 \wedge [\![\widehat{e},\widehat{r}]\!] \wedge \widehat{r} = r-1)$.*

*Proof.* We assume $\widehat{e} < e \not\Rightarrow (\widehat{e} = e-1 \wedge [\![\widehat{e},\widehat{r}]\!] \wedge \widehat{r} = r-1)$; thus, $\widehat{e} < e-1 \vee [\widehat{e},\widehat{r}] \vee \widehat{r} \neq r-1$. First, we assume $\widehat{e} < e-1$. Yet, $p_j$'s state $[\![e,r]\!]$ is preceded by a state in epoch $e-1$; note

that $[[e,r]]$ cannot be the initial state since $m_j^{(e,r)}$ is A-broadcast in $[[e,r]]$. As a result, $p_j$ completes at least one state in epoch $e-1$ before sending $m_j^{(e,r)}$. This means, $p_i$ starts at least one state in epoch $e-1$ before receiving $m_j^{(e,r)}$ in epoch $\widehat{e}$, which contradicts $\widehat{e} < e-1$. Therefore, $\widehat{e} = e-1$.

Second, we assume $p_i$ receives $m_j^{(e,r)}$ in $[e-1,\widehat{r}]$. Let $\phi$ be the failure notification that triggers $p_j$'s transition from epoch $e-1$ to epoch $e$. Then, $p_j$ sends $\phi$ to $p_i$ before it sends $m_j^{(e,r)}$; consequently, $p_i$ receives $\phi$ either in $[e-1,\widehat{r}]$ or in a preceding state. On the one hand, a failure notification received in $[e-1,\widehat{r}]$ increases the epoch, which contradicts the assumption that $p_i$ receives $m_j^{(e,r)}$ in epoch $e-1$. On the other hand, receiving $\phi$ in a state preceding $[e-1,\widehat{r}]$ contradicts the result of Lemma 6.2.4. Thus, $p_i$ receives $m_j^{(e,r)}$ in $[[e-1,\widehat{r}]]$.

Finally, we assume $\widehat{r} \neq r-1$. According to the transitions leading to reliable rounds, $p_j$'s state $[[e,r]]$ is preceded by a completed state with epoch $e-1$ and round either $r$ or $r-1$ (§ 6.2.2). Thus, $p_i$ starts a state with epoch $e-1$ and round either $r$ or $r-1$ before receiving $m_j^{(e,r)}$ in $[[e-1,\widehat{r}]]$. For $[[e-1,\widehat{r}]]$ to be preceded by a state with epoch $e-1$, $[[e-1,\widehat{r}]]$ must be preceded by a skip transition; yet, in such a case the preceding state cannot be completed by any non-faulty server (§ 6.2.2). Thus, $p_j$'s state $[[e,r]]$ is preceded by $[[e-1,r]]$ (since $\widehat{r} \neq r-1$). However, there is no sequence of transitions from $[[e-1,r]]$ to $[[e,r]]$; in other words, when $p_j$ completes $[[e-1,r]]$ it A-delivers round $r$ and thus, it cannot rerun round $r$ in a subsequent epoch. Therefore, $\widehat{r} = r-1$. □

**Proposition 6.2.6.** *Let $p_i$ and $p_j$ be two non-faulty servers. Let $m_j^{(e,r)}$ be a message A-broadcast by $p_j$ while in $[[e,r]]$. If $p_i$ receives $m_j^{(e,r)}$ in epoch $\widehat{e} = e$ and round $\widehat{r} \leq r$, then it is in either $[[e,r-1]]$ or $[[e,r]]$.*

*Proof.* First, we assume $p_i$ is in state $[e,\widehat{r}]$. Then, $[e,\widehat{r}]$ is preceded by a completed state $[[e,r' < \widehat{r}]]$. Yet, this implies that $p_j$ started $[[e,r']]$ (cf. Proposition 6.2.1). Moreover, $p_j$ could not have skipped $[[e,r']]$, since $p_i$ completed it. Thus, $p_j$'s state $[[e,r]]$ is preceded by a completed state $[[e,r']]$ (since $r' < r$), which contradicts the definition of an epoch. As a result, $p_i$ is in state $[[e,\widehat{r}]]$.

Second, we assume $\widehat{r} < r-1$. Therefore, $p_i$ and $p_j$ are in the same epoch, both in reliable rounds, but with at least another round between them. We show that this is not possible. According to the transitions leading to reliable rounds, $p_j$'s $[[e,r]]$ is preceded by either a completed state, i.e., $[e-1,r+1]$, $[e-1,r]$ or $[[e-1,r-1]]$, or a skipped state, i.e., $[[e,r-1]]$ (§ 6.2.2). On the one hand, a completed preceding state $s$ entails that $p_i$ started $s$ (cf. Proposition 6.2.1); moreover, since $p_i$ is in epoch $e$ and the state $s$ has epoch $e-1$, $p_i$ also completed $s$. Yet, if $p_i$ completed any of the $[e-1,r+1]$, $[e-1,r]$ and $[[e-1,r-1]]$ states, there is no reason to rerun (in epoch $e$) round $\widehat{r} < r-1$. On the other hand, if $p_j$'s $[[e,r]]$ is preceded by a skip transition, then at least one non-faulty server completed $[e-1,r]$ and thus, $p_i$ completed $[e-1,r-1]$ (see Figure 6.1). As a result, again there is no reason for $p_i$ to rerun (in epoch $e$) round $\widehat{r} < r-1$. □

## 6.3 The design of the AllConcur+ algorithm

In this section, we provide the details of AllConcur+'s design as a non-uniform atomic broadcast algorithm. We first outline the design of AllConcur+ through a concise event-based description (§ 6.3.1). Then, we provide a more detailed description, including pseudocode (§ 6.3.1). Initially, we assume both $\mathscr{P}$ and no more than $f$ failures during the whole

| # | State | Event | Actions |
|---|---|---|---|
| 1 | $[\widehat{e},\widehat{r}]$ or $[\widehat{e},\widehat{r}]_{\triangleright}$ | recv. $m_{j,[\widehat{e},\widehat{r}+1]}$ | postpone sending and delivery for $[\widehat{e},\widehat{r}+1]$ |
| 2 | $[\widehat{e},\widehat{r}]$ or $[\widehat{e},\widehat{r}]_{\triangleright}$ | recv. $m_{j,[\widehat{e},\widehat{r}]}$ | (1) send $m_{j,[\widehat{e},\widehat{r}]}$ further (via $G_U$)<br>(2) A-broadcast $m_{i,[\widehat{e},\widehat{r}]}$ (if not done already)<br>(3) try to complete $[\widehat{e},\widehat{r}]$ |
| 3 | $[\widehat{e},\widehat{r}]$ | recv. (valid) $fn_{j,k}$ | move to $[[\widehat{e}+1,\widehat{r}-1]]$ and re-handle $fn_{j,k}$ (see #9) |
| 4 | $[\widehat{e},\widehat{r}]_{\triangleright}$ | recv. (valid) $fn_{j,k}$ | move to $[[\widehat{e}+1,\widehat{r}]]$ and re-handle $fn_{j,k}$ (see #9) |
| 5 | $[[\widehat{e},\widehat{r}]]$ | recv. $m_{j,[\widehat{e},\widehat{r}+1]}$ | postpone sending and delivery for $[\widehat{e},\widehat{r}+1]$ |
| 6 | $[[\widehat{e},\widehat{r}]]$ | recv. $m_{j,[[\widehat{e}+1,\widehat{r}+1]]}$ | (1) send $m_{j,[[\widehat{e}+1,\widehat{r}+1]]}$ further (via $G_R$)<br>(2) postpone delivery for $[[\widehat{e}+1,\widehat{r}+1]]$ |
| 7 | $[[\widehat{e},\widehat{r}]]$ | recv. $m_{j,[[\widehat{e},\widehat{r}+1]]}$ | (1) A-deliver $[\widehat{e}-1,\widehat{r}]$<br>(2) move to $[[\widehat{e},\widehat{r}+1]]$ and re-handle $m_{j,[[\widehat{e},\widehat{r}+1]]}$ (see #8) |
| 8 | $[[\widehat{e},\widehat{r}]]$ | recv. $m_{j,[[\widehat{e},\widehat{r}]]}$ | (1) send $m_{j,[[\widehat{e},\widehat{r}]]}$ further (via $G_R$)<br>(2) A-broadcast $m_{i,[[\widehat{e},\widehat{r}]]}$ (if not done already)<br>(3) try to complete $[[\widehat{e},\widehat{r}]]$ |
| 9 | $[[\widehat{e},\widehat{r}]]$ | recv. (valid) $fn_{j,k}$ | (1) send $fn_{j,k}$ further (via $G_R$)<br>(2) update tracking digraphs<br>(3) try to complete $[[\widehat{e},\widehat{r}]]$ |

TABLE 6.1: The actions performed by a server $p_i$ when different events occur while in epoch $\widehat{e}$ and round $\widehat{r}$. $m_{j,[e,r]}$ denotes an unreliable message sent by $p_j$ while in $[e,r]$; $m_{j,[[e,r]]}$ denotes a reliable message sent by $p_j$ while in $[[e,r]]$; and $fn_{j,k}$ denotes a notification sent by $p_k$ indicating $p_j$'s failure. $p_i$ drops all other messages as well as invalid failure notifications received.

deployment (i.e., it is not necessary to update $G_R$). Later, we discuss both how to adapt AllConcur+ to use $\Diamond\mathscr{P}$ instead of $\mathscr{P}$ (see Section 6.3.3.1) and how to update $G_R$ in order to maintain reliability despite failures having occurred (see Section 6.3.3.2).

### 6.3.1 Event-based description

Table 6.1 summarizes the actions performed by a server $p_i$ when different events occur. We assume $p_i$ is in epoch $\widehat{e}$ and round $\widehat{r}$. We distinguish between the three states described in Section 6.2.2: $[\widehat{e},\widehat{r}]_{\triangleright}$; $[\widehat{e},\widehat{r}]$; and $[[\widehat{e},\widehat{r}]]$. We consider also the three events that can occur:

- $p_i$ receives $m_{j,[e,r]}$, an unreliable message sent by $p_j$ while in $[e,r]$;
- $p_i$ receives $m_{j,[[e,r]]}$, a reliable message sent by $p_j$ while in $[[e,r]]$;
- $p_i$ receives $fn_{j,k}$, a notification sent by $p_k$ indicating $p_j$'s failure.

Note that messages are uniquely identified by the tuple (*source id*, *epoch number*, *round number*, *round type*), while failure notifications by the tuple (*target id*, *owner id*).

**Handling unreliable messages.** If $p_i$ receives in any state an unreliable message $m_{j,[e,r]}$, then it cannot have been sent from a subsequent epoch (i.e., $e \leq \widehat{e}$, cf. Proposition 6.2.3). Moreover, unreliable messages from either previous epochs or previous rounds can be dropped. Therefore, $p_i$ must handle unreliable messages sent only from the current epoch (i.e., $e = \widehat{e}$). If $m_{j,[\widehat{e},r]}$ was sent from a subsequent round, then $r = \widehat{r}+1$ (cf. Proposition 6.2.3). In this case, $p_i$ postpones both the sending and the delivery of $m_j$ for $[\widehat{e},\widehat{r}+1]$ (see #1 and #5 in Table 6.1). Otherwise, $m_{j,[\widehat{e},r]}$ was sent from the current round (i.e., $r = \widehat{r}$)

and thus, $p_i$ can only be in an unreliable round (cf. Proposition 6.2.2). Handling $m_{j,[\widehat{e},\widehat{r}]}$ while in $[\widehat{e},\widehat{r}]$ consists of three operations (see #2 in Table 6.1): (1) send $m_j$ further via $G_U$; (2) A-broadcast own message (if not done already); and (3) try to complete the unreliable round $\widehat{r}$. The necessary and sufficient condition for $p_i$ to complete an unreliable round is to receive a message (sent in that round) from every server. Once $p_i$ completes $[\widehat{e},\widehat{r}]$ (and not $[\widehat{e},\widehat{r}]_{\triangleright}$), it A-delivers $[\widehat{e},\widehat{r}-1]$. The completion of either $[\widehat{e},\widehat{r}]$ or $[\widehat{e},\widehat{r}]_{\triangleright}$ is followed by a $\mathrm{T_{UU}}$ transition to $[\widehat{e},\widehat{r}+1]$, which entails handling any postponed unreliable messages (see #1 in Table 6.1).

**Handling reliable messages.** If $p_i$ receives in any state a reliable message $m_{j,[\![e,r]\!]}$ sent from a subsequent epoch, then it is also from a subsequent round (i.e., $e > \widehat{e} \Rightarrow r > \widehat{r}$ (cf. Proposition 6.2.5)). Therefore, unreliable messages from either previous epochs or previous rounds can be dropped (i.e., clearly, messages from preceding epochs are outdated and, since $e > \widehat{e} \Rightarrow r > \widehat{r}$, messages from preceding rounds are outdated as well). As a result, $p_i$ must handle reliable messages sent from either the current or the subsequent epoch (i.e., $e \geq \widehat{e}$); in both cases, $p_i$ can only be in an reliable round (cf. Propositions 6.2.5 and 6.2.6). If $m_{j,[\![e,r]\!]}$ was sent from a subsequent epoch, then $e = \widehat{e}+1$ and $r = \widehat{r}+1$ (cf. Proposition 6.2.5); in this case, $p_i$ postpones the delivery of $m_j$ for $[\![\widehat{e}+1,\widehat{r}+1]\!]$; yet, it sends $m_j$ further via $G_R$[1] (see #6 in Table 6.1). Otherwise, $m_{j,[\![\widehat{e},r]\!]}$ was sent from either the subsequent or the current round (cf. Proposition 6.2.6). Receiving $m_{j,[\![\widehat{e},\widehat{r}+1]\!]}$ while in $[\![\widehat{e},\widehat{r}]\!]$ triggers a $\mathrm{T_{Sk}}$ transition (see Figure 6.1). $\mathrm{T_{Sk}}$ consists of two operations(see #7 in Table 6.1): (1) A-deliver the last completed state (i.e., $[\widehat{e}-1,\widehat{r}]$); and (2) move to $[\![\widehat{e},\widehat{r}+1]\!]$ and re-handle $m_{j,[\![\widehat{e},\widehat{r}+1]\!]}$. Finally, receiving $m_{j,[\![\widehat{e},\widehat{r}]\!]}$ while in $[\![\widehat{e},\widehat{r}]\!]$ consists of three operations (see #8 in Table 6.1): (1) send $m_j$ further via $G_R$; (2) A-broadcast own message (if not done already); and (3) try to complete the reliable round $\widehat{r}$.

To complete a reliable round, AllConcur+ uses early termination—the necessary and sufficient condition for $p_i$ to complete $[\![\widehat{e},\widehat{r}]\!]$ is to stop tracking all messages. Consequently, once $p_i$ completes $[\![\widehat{e},\widehat{r}]\!]$, it can safely A-delivered it. Moreover, servers, for which no message was A-delivered, are removed. This entails both updating $G_U$ to ensure connectivity[2] and removing the invalid failure notifications (§ 6.2.1). Depending on whether all failure notifications are removed, we distinguish between a no-fail transition $\mathrm{T_{R\triangleright}}$ and a fail transition $\mathrm{T_{RR}}$. A $\mathrm{T_{R\triangleright}}$ transition to $[\widehat{e},\widehat{r}+1]_{\triangleright}$ entails handling any postponed unreliable messages (see #2 in Table 6.1). A $\mathrm{T_{RR}}$ transition to $[\![\widehat{e}+1,\widehat{r}+1]\!]$ entails delivering any postponed reliable messages (see #8 in Table 6.1).

**Handling failure notifications.** A failure notification $fn_{j,k}$ is valid only if both the owner (i.e., $p_k$) and the target (i.e., $p_j$) are not removed (§ 6.2.1). Receiving a valid notification while in an unreliable round, triggers a rollback to the latest A-delivered round and the reliable rerun of the subsequent round. Consequently, if $p_i$ is in $[\widehat{e},\widehat{r}]$, then it moves to $[\![\widehat{e}+1,\widehat{r}-1]\!]$ (see #3 in Table 6.1), while from $[\widehat{e},\widehat{r}]_{\triangleright}$ it moves to $[\![\widehat{e}+1,\widehat{r}]\!]$ (see #4 in Table 6.1). In both cases, $p_i$ re-handles $fn_{j,k}$ in the new reliable round. Handling a valid notification while in a reliable round consists of three operations (see #9 in Table 6.1): (1) send the notification further via $G_R$; (2) update the tracking digraphs; and (3) try to complete the reliable round $\widehat{r}$.

Updating the tracking digraphs after receiving a valid notification $fn_{j,k}$ follows the procedure described in the AllConcur chapter (see Section 5.3). For any tracking digraph $\mathbf{g}[p_*]$ that contains $p_j$, we identify two cases. First, if $p_j$ has no successors in $\mathbf{g}[p_*]$, then $\mathbf{g}[p_*]$ is recursively expanded by (uniquely) adding the successors of any vertex $p_p \in \mathcal{V}(\mathbf{g}[p_*])$ that

[1] AllConcur+ does not postpone the sending of failure notifications, and thus, to not break the message tracking mechanism that relies on message ordering, it does not postpone sending reliable messages either (see Section 6.3.2.6).

[2] During every reliable round, the servers agree on the next $G_U$.

is the target of a received failure notification, except for those successors that are the owner of a failure notification targeting $p_p$. Second, if one of $p_j$ successors in $\mathbf{g}[p_*]$ is $p_k$, then the edge $(p_j, p_k)$ is removed from $\mathbf{g}[p_*]$. In addition, $\mathbf{g}[p_*]$ is pruned by first removing the servers with no path from $p_*$ to themselves and then, if all remaining servers are targeted by received failure notifications, by removing all of them. When starting a reliable round, the tracking digraphs are reset and the valid failure notifications are redelivered. Thus, this procedure needs to be repeated for all valid failure notifications.

**Initial bootstrap and dynamic membership.** Similarly to AllConcur, bootstrapping AllConcur+ requires a reliable centralized service, such as Chubby [27], that enables the servers to agree on the initial configuration (i.e., the identity of the participating servers and the two digraphs). Once AllConcur+ starts, it is completely decentralized—any further reconfigurations (including also membership changes) are agreed upon via atomic broadcast.

### 6.3.2 Design details

Here, we provide a more detailed description of AllConcur+'s design, including pseudocode (see Algorithms 6.1–6.6) . First, we specify the variables used and their initial values; also, we outline the main loop and the main communication primitives, i.e., *broadcast*(), *R-broadcast*() and *A-broadcast*(). Then, we split the description of the design into the following points: (1) handling of the three possible events—receiving an unreliable message, receiving a reliable message, and receiving a failure notification; (2) conditions for completing and A-delivering a round; (3) handling of premature messages; and (4) updating the tracking digraphs.

#### 6.3.2.1 Overview

AllConcur+'s design is outlined in Algorithm 6.1; the code is executed by a server $p_i$. The variables used are described in the Input section. Apart from $m_j^{(e,r)}$, which denotes a message A-broadcast by $p_j$ in epoch $e$ and round $r$, all variables are local to $p_i$:

- $\widehat{e}$ denotes $p_i$'s epoch;
- $\widehat{r}$ denotes $p_i$'s round;
- $M_i$ contains messages received (by $p_i$) during the current state;
- $M_i^{prev}$ contains messages received (by $p_i$) in the previous completed, but not yet A-delivered round (if any);
- $M_i^{next}$ contains messages received prematurely (by $p_i$), i.e.,

$$m_j^{(e,r)} \in M_i^{next} \Rightarrow e > \widehat{e} \vee (e = \widehat{e} \wedge r > \widehat{r});$$

- $F_i$ contains $p_i$'s valid failure notifications;
- $\mathbf{g_i}$ are $p_i$'s tracking digraphs.

Initially, $p_i$ is in state $[[1,0]]$; before starting the main loop, it moves to $[1,1]$ (line 1). In the main loop (line 2), $p_i$ A-broadcasts its own message $m_i^{(\widehat{e},\widehat{r})}$, if it is neither empty, nor was already A-broadcast. Depending on AllConcur+'s mode, the A-broadcast primitive is either a broadcast that uses $G_U$ or an R-broadcast that uses $G_R$ (lines 10–19). Also, in the main loop, $p_i$ handles received messages (unreliable and reliable) and failure notifications.

**Algorithm 6.1:** The AllConcur+ algorithm under the assumption of both $\diamond\mathcal{P}$ and no more than $f$ failures; code executed by server $p_i$; see Table 2.1 for digraph notations.

**Input:** $G_U$; $G_R$ {$p_i$'s unreliable and reliable digraphs}
$[[\widehat{e} \leftarrow 1, \widehat{r} \leftarrow 0]]$ {$p_i$'s initial state}
$M_i \leftarrow \emptyset$ {messages received by $p_i$ in current state}
$M_i^{prev} \leftarrow \emptyset$ {messages received by $p_i$ in previous not A-delivered (unreliable) round}
$M_i^{next} \leftarrow \emptyset$ {messages received prematurely by $p_i$}
$F_i \leftarrow \emptyset$ {$p_i$'s known failure notifications}
$\mathcal{V}(\mathbf{g_i}[p_*]) \leftarrow \{p_*\}, \forall p_* \in \mathcal{V}(G_R)$ {tracking digraphs}
$m_j^{(e,r)}$ {$p_j$'s message while in $[e,r]$ or $[[e,r]]$}

1 $[[\widehat{e},\widehat{r}]] \rightarrow [\widehat{e},\widehat{r}+1]$ {transition from initial state $[[1,0]]$}
2 **loop**
3 **if** $m_i^{(\widehat{e},\widehat{r})}$ *not empty* and $m_i^{(\widehat{e},\widehat{r})} \notin M_i$ **then** *A-broadcast*$(m_i^{(\widehat{e},\widehat{r})})$
4 **if** *receive* $\langle BCAST, m_j^{(e,r)} \rangle$ **then**
5 HandleBCAST($m_j^{(e,r)}$) {handle unreliable message (§ 6.3.2.2)}
6 **if** *receive* $\langle RBCAST, m_j^{(e,r)} \rangle$ **then**
7 HandleRBCAST($m_j^{(e,r)}$) {handle reliable message (§ 6.3.2.3)}
8 **if** *receive* $\langle FAIL, p_j, p_k \in p_j^+(G_R) \rangle$ **then**
9 HandleFAIL($p_j$,$p_k$) {handle failure notification (§ 6.3.2.4)}
10 **def** ***A-broadcast***$(m_j^{(e,r)})$**:**
11 **if** $[\widehat{e},\widehat{r}]$ **then** *broadcast*$(m_j^{(e,r)})$
12 **else if** $[[\widehat{e},\widehat{r}]]$ **then** *R-broadcast*$(m_j^{(e,r)})$
13 **def** ***broadcast***$(m_j^{(e,r)})$**:**
14 **if** $m_j^{(e,r)} \notin M_i$ **then send** $\langle BCAST, m_j^{(e,r)} \rangle$ **to** $p_i^+(G_U)$
15 $M_i \leftarrow M_i \cup \{m_j^{(e,r)}\}$
16 **def** ***R-broadcast***$(m_j^{(e,r)})$**:**
17 **if** $m_j^{(e,r)} \notin M_i$ **then send** $\langle RBCAST, m_j^{(e,r)} \rangle$ **to** $p_i^+(G_R)$
18 $M_i \leftarrow M_i \cup \{m_j^{(e,r)}\}$
19 $\mathcal{V}(\mathbf{g_i}[p_j]) \leftarrow \emptyset$

#### 6.3.2.2 Handling unreliable messages

Algorithm 6.2 shows the code executed by $p_i$ when it receives an unreliable message $m_j^{(e,r)}$, i.e., $m_j^{(e,r)}$ was sent by $p_j$ while in $[e,r]$. Unreliable messages cannot originate from subsequent epochs (cf. Proposition 6.2.3). Also, unreliable messages from preceding states are dropped (line 21), because messages from preceding epochs are outdated and, since $e \leq \widehat{e}$, messages from preceding rounds are outdated as well. As a result, $m_j^{(e,r)}$ was sent from either $[\widehat{e},\widehat{r}+1]$ (cf. Proposition 6.2.3) or $[\widehat{e},\widehat{r}]$ (cf. Proposition 6.2.2). We defer the handling of premature messages, i.e., sent from subsequent states, to Section 6.3.2.6. Handling an unreliable message sent from $[\widehat{e},\widehat{r}]$ consists of three operations (lines 25–27): (1) send $m_j^{(e,r)}$ further (using $G_U$); (2) A-broadcast own message (if not done so already); and (3) try to complete round $\widehat{r}$ (see Section 6.3.2.5).

**Algorithm 6.2:** Handling an unreliable message—while in epoch $\widehat{e}$ and round $\widehat{r}$, $p_i$ receives $m_j^{(e,r)}$ sent by $p_j$ while in $[e,r]$.

```
20 def HandleBCAST(m_j^(e,r)):
       /* e ≤ ê (cf. Propositions 6.2.3)                                  */
21     if e < ê or r < r̂ then drop m_j^(e,r)
22     else if r > r̂ then                  {r = r̂ + 1 (cf. Proposition 6.2.3)}
           /* postpone handling m_j^(e,r) for [ê, r̂ + 1]                  */
23         if ∀m_*^(e',r') ∈ M_i^next : e' = ê then M_i^next ← M_i^next ∪ {m_j^(e,r)}
24     else                  {e = ê ∧ r = r̂ ⇒ [ê, r̂] (cf. Proposition 6.2.2)}
25         broadcast(m_j^(e,r))                  {send m_j^(e,r) further via G_U}
26         A-broadcast(m_i^(ê,r̂))                  {A-broadcast own message}
27         TryToComplete()              {try to complete round r̂ (§ 6.3.2.5)}
```

#### 6.3.2.3 Handling reliable messages

Algorithm 6.3 shows the code executed by $p_i$ when it receives a reliable message $m_j^{(e,r)}$, i.e., $m_j^{(e,r)}$ was sent by $p_j$ while in $[\![e,r]\!]$. Reliable messages from preceding states are dropped (line 29), because messages from preceding epochs are outdated and, since $e > \widehat{e} \Rightarrow r > \widehat{r}$ (cf. Proposition 6.2.5), messages from preceding rounds are outdated as well. As a result, we identify three scenarios (cf. Propositions 6.2.5 and 6.2.6):

- $m_j^{(e,r)}$ was sent from the subsequent epoch (i.e., from $[\![\widehat{e}+1,\widehat{r}+1]\!]$);
- $m_j^{(e,r)}$ was sent from the current state $[\![\widehat{e},\widehat{r}]\!]$;
- $m_j^{(e,r)}$ was sent from the current epoch, but from the subsequent reliable round (i.e., from $[\![\widehat{e},\widehat{r}+1]\!]$).

In all three scenarios, $p_i$ is in a reliable round (cf. Propositions 6.2.5 and 6.2.6).

First, we defer the handling of premature messages (i.e., sent from $[\![\widehat{e}+1,\widehat{r}+1]\!]$) to Section 6.3.2.6. Second, handling a reliable message sent from the current state $[\![\widehat{e},\widehat{r}]\!]$ consists of three operations (lines 43–45): (1) send $m_j^{(e,r)}$ further (using $G_R$); (2) A-broadcast own message (if not done so already); and (3) try to complete round $\widehat{r}$ (see Section 6.3.2.5). Third, receiving a reliable message from the same epoch and a subsequent round while in a reliable round triggers a $\mathrm{T_{Sk}}$ transition (see Figure 6.1). $\mathrm{T_{Sk}}$ consists of three operations (lines 35–42): (1) A-deliver last completed state (i.e., $[\widehat{e}-1,\widehat{r}]$); (2) initialize the next round, including updating the tracking digraphs (see Section 6.3.2.7); and (3) move to $[\![\widehat{e},\widehat{r}+1]\!]$. Then, $m_j^{(e,r)}$ is handled as if it was received in the same state (lines 43–45).

#### 6.3.2.4 Handling failure notifications

Algorithm 6.4 shows the code executed by $p_i$ when it receives a failure notification sent by $p_k$ targeting one of its predecessors $p_j$. Note that if $k = i$, then the notification is from the local failure detector. The notification is valid only if both the owner (i.e., $p_k$) and the target (i.e., $p_j$) are part of the reliable digraph (§ 6.2.1). Handling a valid notification consists of five operations: (1) send the notification further using $G_R$ (line 48); (2) if in an unreliable round, rollback to the latest A-delivered round and start a reliable round (lines 49–52); (3) update the tracking digraphs (see Section 6.3.2.7); (4) add the failure notification to $F_i$ (line 55); and (5) try to complete current reliable round $\widehat{r}$ (see Section 6.3.2.5).

**Algorithm 6.3:** Handling a reliable message—while in epoch $\widehat{e}$ and round $\widehat{r}$, $p_i$ receives $m_j^{(e,r)}$ sent by $p_j$ while in $[[e,r]]$; see Table 2.1 for digraph notations.

```
28 def HandleRBCAST(m_j^(e,r)):
       /* e > ê ⇒ r > r̂ (cf. Proposition 6.2.5) */
29     if e < ê or r < r̂ then drop m_j^(e,r)
30     else if e > ê then {e = ê+1 ∧ r = r̂+1 (cf. Proposition 6.2.5)}
           /* send m_j^(e,r); deliver later in [[ê+1, r̂+1]] */
31         send ⟨RBCAST, m_j^(e,r)⟩ to p_i^+(G_R)
32         if ∃m_*^(e',r') ∈ M_i^next : e' = ê then M_i^next ← ∅
33         M_i^next ← M_i^next ∪ {m_j^(e,r)}
34     else {r = r̂ ∨ r = r̂+1 (cf. Proposition 6.2.6)}
35         if r = r̂+1 then {skip transition}
36             foreach m ∈ M_i^prev do {A-deliver [ê−1, r̂]}
37                 A-deliver(m)
38             M_i^prev ← ∅; M_i ← ∅; M_i^next ← ∅
39             foreach p_* ∈ V(G_R) do
40                 V(g_i[p_*]) ← {p_*}
41                 UpdateTrackingDigraph(g_i[p_*], ∅, F_i)
42             [[ê, r̂]] —skip→ [[ê, r̂+1]] {T_Sk}
43         R-broadcast(m_j^(e,r)) {send m_j^(e,r) further via G_R}
44         A-broadcast(m_i^(ê,r̂)) {A-broadcast own message}
45         TryToComplete() {try to complete round r̂ (§ 6.3.2.5)}
```

**Algorithm 6.4:** Handling a failure notification—while in epoch $\widehat{e}$ and round $\widehat{r}$, $p_i$ receives from $p_k$ a notification of $p_j$'s failure; see Table 2.1 for digraph notations.

```
46 def HandleFAIL(p_j, p_k):
       /* if k = i then notification from local FD */
47     if p_j ∉ V(G_R) or p_k ∉ V(G_R) then return
48     send ⟨FAIL, p_j, p_k⟩ to p_i^+(G_R) {send further via G_R}
49     if [ê, r̂] then {rollback}
50         M_i ← ∅; M_i^next ← ∅
51         if M_i^prev ≠ ∅ then [ê, r̂] —fail→ [[ê+1, r̂−1]] {T_UR}
52         else [ê, r̂]_▷ —fail→ [[ê+1, r̂]] {T_▷R}
53     foreach p_* ∈ V(G_R) do
54         UpdateTrackingDigraph(g_i[p_*], F_i, {(p_j, p_k)}) {(§ 6.3.2.7)}
55     F_i ← F_i ∪ {(p_j, p_k)}
56     TryToComplete() {try to complete round r̂ (§ 6.3.2.5)}
```

#### 6.3.2.5 Round completion

After handling either a message or a failure notification, $p_i$ tries to complete the current round. If successful, it tries to safely A-deliver messages and finally, it moves to the next state. We distinguish between completing an unreliable round (lines 58–66) and completing a reliable round (lines 67–91).

**Completing an unreliable round.** The necessary and sufficient condition for $p_i$ to complete $[\widehat{e},\widehat{r}]$ is to receive a message (sent in $[\widehat{e},\widehat{r}]$) from every server (line 59). The completion

**Algorithm 6.5:** Round completion—while in epoch $\widehat{e}$ and round $\widehat{r}$, $p_i$ tries to complete the current round and safely A-deliver messages; see Table 2.1 for digraph notations.

```
57 def TryToComplete():
58     if [ê, r̂] then
59         if |M_i| = |V(G_U)| then
60             foreach m ∈ M_i^prev do          {A-deliver [ê, r̂ − 1] (if it exists)}
61                 A-deliver(m)
62             [ê, r̂] → [ê, r̂ + 1]                                          {T_UU}
               /* handle postponed unreliable messages                        */
63             foreach m_*^(e,r) ∈ M_i^next do
64                 send ⟨BCAST, m_*^(e,r)⟩ to p_i^+(G_U)
65             M_i^prev ← M_i; M_i ← M_i^next; M_i^next ← ∅
66             if M_i ≠ ∅ then A-broadcast(m_i^(ê,r̂))
67     else if V(g_i[p_*]) = ∅, ∀p_* ∈ V(G_R) then
68         foreach m ∈ M_i do A-deliver(m)                          {A-deliver [[ê, r̂]]}
69         if {p_* : m_* ∉ M_i} ≠ ∅ then                                {remove servers}
70             UpdateUnreliableDigraph(G_U, {p_* : m_* ∉ M_i})
71             foreach p ∈ {p_* : m_* ∉ M_i} do
72                 V(G_R) ← V(G_R) \ {p}                                  {adjust G_R}
73                 E(G_R) ← E(G_R) \ {(x, y) : x = p ∨ y = p}
74                 F_i ← F_i \ {(x, y) : x = p ∨ y = p}                    {adjust F_i}
75         V(g_i[p_*]) ← {p_*}, ∀p_* ∈ V(G_R)
76         M_i^prev ← ∅
77         if F_i = ∅ then
78             [[ê, r̂]] → [ê, r̂ + 1]_▷                                      {T_R▷}
               /* handle postponed unreliable messages                        */
79             foreach m_*^(e,r) ∈ M_i^next do
80                 send ⟨BCAST, m_j^(e,r)⟩ to p_i^+(G_U)
81             M_i ← M_i^next; M_i^next ← ∅
82         else
83             [[ê, r̂]] —fail→ [[ê + 1, r̂ + 1]]                             {T_RR}
84             if ∃m_*^(e,r) ∈ M_i^next : e = ê − 1 then                    {discard}
85                 M_i^next ← ∅; M_i ← ∅
86             else                          {deliver postponed reliable messages}
87                 M_i ← M_i^next; M_i^next ← ∅
88                 foreach m_* ∈ M_i do V(g_i[p_*]) ← ∅
89             foreach p_* ∈ V(G_R) do
90                 UpdateTrackingDigraph(g_i[p_*], ∅, F_i)                {(§ 6.3.2.7)}
91         if M_i ≠ ∅ then A-broadcast(m_i^(ê,r̂))
```

is followed by a $\mathrm{T}_{\mathrm{UU}}$ transition to $[\widehat{e},\widehat{r}+1]$ (line 62). Starting $[\widehat{e},\widehat{r}+1]$ entails handling the messages (sent in $[\widehat{e},\widehat{r}+1]$) received (by $p_i$) prematurely (lines 63–65); we defer this discussion to Section 6.3.2.6. Also, if any messages were already received for $[\widehat{e},\widehat{r}+1]$, then $p_i$ A-broadcast its own message (line 66). In addition, if $[\widehat{e},\widehat{r}]$ is not the first in a sequence of unreliable rounds, then $p_i$ A-delivers $[\widehat{e},\widehat{r}-1]$ (line 61).

**Completing a reliable round.** Due to early termination, the necessary and sufficient condition for $p_i$ to complete $[\![\widehat{e},\widehat{r}]\!]$ is for all its tracking digraphs to be empty (line 67). Consequently, a completed reliable round can be safely A-delivered (line 68). Moreover, servers, for which no message was A-delivered, are removed. This entails updating both $G_U$ to ensure connectivity (§ 6.3.1) and $F_i$ (line 74). Depending on whether $F_i$ is empty after the update, we distinguish between a no-fail transition $\mathrm{T}_{\mathrm{R}\triangleright}$ (lines 77–81) and a fail transition $\mathrm{T}_{\mathrm{RR}}$ (lines 82–90). Starting $[\widehat{e},\widehat{r}+1]_{\triangleright}$ after $\mathrm{T}_{\mathrm{R}\triangleright}$ entails handling the messages (sent in $[\widehat{e},\widehat{r}+1]_{\triangleright}$) received (by $p_i$) prematurely (lines 79–81). Similarly, starting $[\![\widehat{e}+1,\widehat{r}+1]\!]$ after $\mathrm{T}_{\mathrm{RR}}$ entails handling the messages (sent in $[\![\widehat{e}+1,\widehat{r}+1]\!]$) received (by $p_i$) prematurely (lines 84–88). We defer the handling of premature messages to Section 6.3.2.6. In addition, $\mathrm{T}_{\mathrm{RR}}$ requires the tracking digraphs to be updated (see Section 6.3.2.7); note that before starting either transition, the tracking digraphs are reset (line 75). Finally, if any messages were already received for the new state (i.e., $[\widehat{e},\widehat{r}+1]_{\triangleright}$ or $[\![\widehat{e}+1,\widehat{r}+1]\!]$), then $p_i$ A-broadcast its own message (line 91).

#### 6.3.2.6 Handling premature messages

In AllConcur+, servers can be in different states (§ 6.2.4). This entails that a server can receive both A-broadcast messages and failure notifications sent by another server from a future state; we refer to these as premature messages. When handling a premature message, the following two conditions must be satisfied. First, the digraph on which the message has arrived needs to be consistent with the digraph on which the message is sent further. In other words, if the server sending the message has previously updated the digraph, then the server receiving the message must postpone sending it further until it also updates the digraph. Second, changing the message order (with respect to other messages) must not affect early termination; note that message order is not relevant for unreliable messages.

The way premature messages are handled depends on the scope of the failure notifications. In general, to avoid inconsistencies, failure notifications need to be specific to $G_R$: Once $G_R$ is updated, all failure notifications are discarded. Thus, we avoid scenarios where the failure notification's owner is not a successor of the target. To trivially enforce this, failure notifications can be made specific to an epoch, as in AllConcur (see Section 5.3). In this case, premature failure notifications (i.e., specific to a subsequent epoch) are postponed: Both their sending and delivering is delayed until the subsequent epoch. Since reliable messages are specific to an epoch[3], they can also be postponed; thereby, message order is preserved. Once a new epoch starts, the failure notifications specific to the previous epoch are outdated and hence, discarded. This entails detecting again the failures of the faulty servers that were not removed in the previous epoch (§ 6.2.1).

As long as $G_R$ remains unchanged, re-detecting failures can be avoided, by only discarding outdated failure notifications that are invalid and resending the valid ones (i.e., with both owner and target not removed in the previous epoch). Even more, resending failure notifications can be avoided by not postponing them, i.e. they are sent further and delivered immediately. This requires that the valid failure notifications are redelivered at the beginning of each epoch, i.e., updating the tracking digraphs. Note that for message order to be preserved, premature reliable messages are also sent further immediately and only their delivery is postponed to their specific epoch.

In AllConcur+, failure notifications are handled immediately. While $p_i$ is in epoch $\widehat{e}$ and round $\widehat{r}$, it can receive the following premature messages: (1) unreliable messages sent from $[\widehat{e},\widehat{r}+1]$ (cf. Proposition 6.2.3), which are handled in $[\widehat{e},\widehat{r}+1]$ (lines 63–65 and 79–81); and (2) reliable messages sent from $[\![\widehat{e}+1,\widehat{r}+1]\!]$ (cf. Proposition 6.2.5), which are sent immediately (line 31), but delivered in $[\![\widehat{e}+1,\widehat{r}+1]\!]$ (lines 84–88). Postponed messages

[3]Reliable messages that trigger skip transitions are not premature messages.

**Algorithm 6.6:** Updating a tracking digraph—$p_i$ updates $\mathbf{g_i}[p_*]$ after adding $F_{new}$ to the set $F_{old}$ of already known failure notifications; see Table 2.1 for digraph notations.

```
92  def UpdateTrackingDigraph(g_i[p_*], F_old, F_new):
93      F ← F_old
94      foreach (p_j, p_k) ∈ F_new do
95          F ← F ∪ {(p_j, p_k)}
96          if p_j ∉ V(g_i[p_*]) then continue
97          if p_j^+(g_i[p_*]) = ∅ then        {maybe p_j sent m_* further before failing}
98              Q ← {(p_j, p) : p ∈ p_j^+(G_R) \ {p_k}}                    {FIFO queue}
99              foreach (p_p, p) ∈ Q do
                    /* recursively expand g_i[p_*]                                  */
100                 Q ← Q \ {(p_p, p)}
101                 if p ∉ V(g_i[p_*]) then
102                     V(g_i[p_*]) ← V(g_i[p_*]) ∪ {p}
103                     if ∃(p, *) ∈ F then Q ← Q ∪ {(p, p_s) : p_s ∈ p^+(G_R)} \ F
104                 E(g_i[p_*]) ← E(g_i[p_*]) ∪ {(p_p, p)}
105         else if p_k ∈ p_j^+(g_i[p_*]) then
                    /* p_k has not received m_* from p_j                            */
106             E(g_i[p_*]) ← E(g_i[p_*]) \ {(p_j, p_k)}
107             foreach p ∈ V(g_i[p_*]) s.t. ∄π_{p_*,p} in g_i[p_*] do
108                 V(g_i[p_*]) ← V(g_i[p_*]) \ {p}                {prune: no input}
109         if ∀p ∈ V(g_i[p_*]), (p, *) ∈ F then
110             V(g_i[p_*]) ← ∅                              {prune: no dissemination}
```

are stored in $M_i^{next}$; unreliable messages sent from $[\widehat{e},\widehat{r}+1]$ are dropped in favor of reliable messages sent from $[[\widehat{e}+1,\widehat{r}+1]]$ (line 23 and 32).

#### 6.3.2.7 Updating the tracking digraphs

The tracking digraphs are needed only for reliable rounds (i.e., for early termination). Therefore, the tracking digraphs are updated either when a reliable round starts or during a reliable round when receiving a valid failure notification (see Algorithms 6.3–6.5). Updating the tracking digraphs after receiving a failure notification follows the procedure described in the AllConcur chapter (see Section 6.3.1 for details). Algorithm 6.6 shows the `UpdateTrackingDigraph` procedure used by $p_i$ to update a tracking digraph $\mathbf{g_i}[p_*]$ after adding a set of new failure notifications to the set of already known failure notifications.

### 6.3.3 Widening the scope

Sections 6.3.1 and 6.3.2 describe the design of AllConcur+ under the assumption of both $\mathscr{P}$ and no more than $f$ failures during the whole deployment. Here, we discuss the implications of dropping the two assumptions.

#### 6.3.3.1 Eventual accuracy

Without any doubt, using $\Diamond\mathscr{P}$ instead of $\mathscr{P}$ increases AllConcur+'s applicability, i.e., not having to guarantee accuracy, allows for AllConcur+ to be deployed over asynchronous networks, such as the Internet. When using $\Diamond\mathscr{P}$, severs can (temporarily) suspect each other to have failed, without actually failing (see Section 2.2.4). In unreliable rounds, whether

a failure notification is accurate or not does not affect the outcome, i.e., when a server receives a failure notification, it rolls back to the latest A-delivered round and switches to the reliable mode. Replacing $\mathscr{P}$ with $\Diamond\mathscr{P}$ has an impact on reliable rounds, in particular, on the early termination mechanism, i.e., a false suspicion of failure may cause AllConcur+ to not guarantee Proposition 5.2.1, which is necessary for early termination (see Section 5.2.1). Therefore, as in AllConcur, once an AllConcur+ server sends a failure notification targeting one of its predecessors, it ignores any subsequent messages (except failure notifications) received from that predecessor. Moreover, for the set agreement property to hold, servers employ the forward-backwards mechanism (described in Section 5.2.1) in order to decide whether it is safe to A-deliver messages. Finally, the servers for which it is not safe to A-deliver messages and hence, cannot make progress, are eventually removed from the system, i.e., process controlled crash. In Section 6.5, we evaluate the effect of the forward-backwards mechanism on AllConcur+ (i.e., see AllConcur-w/EA).

Using $\Diamond\mathscr{P}$ entails that failure notifications no longer result necessarily in their targets being eventually removed. Let $fn_{j,k}$ be an inaccurate failure notification (i.e., its target $p_j$ is non-faulty). Despite $p_k$ disseminating $fn_{j,k}$ throughout the system, due to $G_R$'s edge-connectivity, which is at least as large as its vertex-connectivity $\kappa(G_R) > f$, $p_j$'s messages are received by all non-faulty servers. Therefore, $p_j$ is never removed (since, in AllConcur+, as in AllConcur, a server is removed if and only if, at the end of a round, no message is A-delivered for that server). Not removing $p_j$ implies that, as long as $p_k$ is also non-faulty, the notification $fn_{j,k}$ remains valid, which results in AllConcur+ running in the reliable mode, even though no failures have occurred.

In order to enable the redundancy-free agreement of the unreliable mode, AllConcur+ must eventually invalidate inaccurate failure notifications. To this end, every failure notification is tagged with a unique sequence number[4] $s$, i.e., $fn_{j,k,s}$. Then, once $p_k$ realizes that $fn_{j,k,s}$ is inaccurate (i.e., it no longer suspects $p_j$ to have failed), it piggybacks on the next message it A-broadcast a $revoke_{j,k,s}$ control message, revoking thus $fn_{j,k,s}$. When a server A-delivers a $revoke_{j,k,s}$ control message, it considers $fn_{j,k,s}$ to be invalid and consequently, removes it from its set of received failure notifications. Since every non-faulty server A-delivers the same set of messages (see Theorem 6.4.6 in Section 6.4.1), this mechanism provides a consistent view of the system Note that, although inaccurate failure notifications that remain valid have less of an impact on AllConcur, by sending *revoke* control messages, the servers can avoid retransmitting these notifications at the beginning of every round (see Section 5.3).

#### 6.3.3.2 Updating the reliable digraph

In general, it is not necessary to update $G_R$, as long as the number of failed servers does not exceed $f$. In AllConcur+, $f$ provides the reliability of the system. Once servers fail, this reliability drops. To keep the same level of reliability, $G_R$ must be periodically updated. Yet, in AllConcur+, failure notifications are immediately handled, which requires $G_R$ to remain unchanged. Thus, we introduce the concept of an *eon*—a sequence of epochs in which $G_R$ remains unchanged. To connect two subsequent eons $\varepsilon_1$ and $\varepsilon_2$, we extend $\varepsilon_1$ with a *transitional* reliable round. This transitional round is similar to the transitional configuration in Raft [130]. In comparison to a normal reliable round, in a transitional round, a server executes two additional operations before A-broadcasting its message: (1) set up $G_R^{\varepsilon_2}$, the digraph for eon $\varepsilon_2$; and (2) start sending heartbeat messages also on $G_R^{\varepsilon_2}$. Note that all the other operations are executed on $G_R^{\varepsilon_1}$.

[4]For instance, a straightforward method to ensure the uniqueness of the sequence number, is for each server to keep a separate counter for each one of the predecessors it monitors.

The transitional round acts as a delimiter between the two digraphs, $G_R^{\varepsilon_1}$ and $G_R^{\varepsilon_2}$, and provides the following guarantee: no non-faulty server can start eon $\varepsilon_2$, before all non-faulty servers start the transitional round. Thus, when a server starts using $G_R^{\varepsilon_2}$ for both R-broadcasting and detecting failures, all other non-faulty servers already set up $G_R^{\varepsilon_2}$ and started sending heartbeat messages on it. Note that failure notifications are eon specific—failure notifications from $\varepsilon_1$ are dropped in $\varepsilon_2$, while failure notifications from $\varepsilon_2$, received in $\varepsilon_1$, are postponed.

## 6.4 Informal proof of correctness

As described in Section 6.3, AllConcur+ solves non-uniform atomic broadcast. Consequently, to prove AllConcur+'s correctness, we show that the four properties of non-uniform atomic broadcast—integrity, validity, agreement, and total order—are guaranteed (see Theorems 6.4.1, 6.4.3, 6.4.7, and 6.4.8). Afterwards, we discuss the modifications required for these properties to apply also to faulty servers—uniform atomic broadcast (§ 6.4.2).

### 6.4.1 Non-uniform atomic broadcast

**Theorem 6.4.1** (Integrity)**.** *For any message m, every non-faulty server A-delivers m at most once, and only if m was previously A-broadcast by sender*$(m)$.

*Proof.* Integrity is guaranteed by construction; the reason is twofold. First, when rolling back, servers rerun rounds not yet A-delivered; thus, every round can be A-delivered at most once. Second, when a round is A-delivered, all the messages in the set $M$ (which are A-delivered in a deterministic order) were previously A-broadcast. □

To prove validity, we introduce the following lemma that proves AllConcur+ makes progress:

**Lemma 6.4.2.** *Let $p_i$ be a non-faulty server that starts epoch e and round r. Then, eventually, $p_i$ makes progress and moves to another state.*

*Proof.* We identify two cases: $[[e,r]]$ and $[e,r]$. If $p_i$ starts $[[e,r]]$, it eventually either completes it (due to early termination) and moves to the subsequent round (after either $\mathrm{T}_{\mathrm{R}\triangleright}$ or $\mathrm{T}_{\mathrm{RR}}$), or it skips it and moves to $[[e,r+1]]$ (after $\mathrm{T}_{\mathrm{Sk}}$). If $p_i$ starts $[e,r]$, either it eventually completes it, after receiving messages from all servers (e.g., if no failures occur), and moves to $[e,r+1]$ (after $\mathrm{T}_{\mathrm{UU}}$) or it eventually receives a failure notifications and rolls back to the latest A-delivered round (after either $\mathrm{T}_{\mathrm{UR}}$ or $\mathrm{T}_{\triangleright\mathrm{R}}$). □

**Theorem 6.4.3** (Validity)**.** *If a non-faulty server A-broadcasts m, then it eventually A-delivers m.*

*Proof.* Let $p_i$ be a non-faulty server that A-broadcast $m$ (for the first time) in round $r$ and epoch $e$. If $[[e,r]]$, then due to early termination, $p_i$ eventually A-delivers $r$. If $p_i$ A-broadcast $m$ in $[e,r]$, then $[e,r]$ is either completed or interrupted by a failure notification (cf. Lemma 6.4.2). If $p_i$ completes $[e,r]$, then $p_i$ either completes $[e,r+1]$ or, due to a fail transition, reruns $r$ reliably in $[[e+1,r]]$ (cf. Lemma 6.4.2). In both cases, $p_i$ eventually A-delivers $r$ (§ 6.2.3).

If $p_i$ does not complete $[e,r]$ (due to a failure notification), we identify two cases depending on whether $[e,r]$ is the first in a sequence of unreliable rounds. If $[e,r]_{\triangleright}$, then $p_i$ reruns $r$ reliably in $[[e+1,r]]$ (after a $\mathrm{T}_{\triangleright\mathrm{R}}$ transition); hence, $p_i$ eventually A-delivers $r$. Otherwise, $p_i$ moves to $[[e+1,r-1]]$, which will eventually be followed by one of $\mathrm{T}_{\mathrm{R}\triangleright}$, $\mathrm{T}_{\mathrm{RR}}$, or $\mathrm{T}_{\mathrm{Sk}}$. $\mathrm{T}_{\mathrm{R}\triangleright}$ leads to $[e+1,r]_{\triangleright}$ and, by following one of the above cases, eventually to the A-delivery of $r$. Both $\mathrm{T}_{\mathrm{RR}}$ and $\mathrm{T}_{\mathrm{Sk}}$ lead to a reliable rerun of $r$ and thus, to its eventual A-delivery. □

To prove both agreement and total order, we first prove *set agreement*—all non-faulty servers agree on the same set of messages for all A-delivered rounds. To prove set agreement, we introduce the following lemmas:

**Lemma 6.4.4.** *Let $p_i$ be a non-faulty server that A-delivers $[e,r]$ after completing $[e,r+1]$. Then, any other non-faulty server $p_j$ eventually A-delivers $[e,r]$.*

*Proof.* If $p_i$ A-delivers $[e,r]$ after completing $[e,r+1]$, $p_j$ must have started $[e,r+1]$ (cf. Proposition 6.2.1), and hence, completed $[e,r]$. As a result, $p_j$ either receives no failure notifications, which means it eventually completes $[e,r+1]$ and A-delivers $[e,r]$, or receives a failure notification and moves to $[\![e+1,r]\!]$ (after $\mathrm{T_{UR}}$). Yet, this failure notification will eventually trigger on $p_i$ a $\mathrm{T_{UR}}$ transition from $[e,r+2]$ to $[\![e+1,r+1]\!]$, which eventually will trigger on $p_j$ a $\mathrm{T_{Sk}}$ transition that leads to the A-delivery of $[e,r]$ (see Figure 6.1). □

**Lemma 6.4.5.** *If a non-faulty server A-delivers round r in epoch e (i.e., either $[\![e,r]\!]$ or $[e,r]$), then, any non-faulty server eventually A-delivers round r in epoch e.*

*Proof.* If $p_i$ A-delivers $[\![e,r]\!]$ (when completing it), then every non-faulty server eventually A-delivers $[\![e,r]\!]$. The reason is twofold: (1) since $p_i$ completes $[\![e,r]\!]$, every non-faulty server must start $[\![e,r]\!]$ (cf. Proposition 6.2.1); and (2) due to early termination, every non-faulty server eventually also completes and A-delivers $[\![e,r]\!]$ (see Section 5.4).

Otherwise, $p_i$ A-delivers $[e,r]$ either once it completes the subsequent unreliable round or after a skip transition from $[\![e+1,r]\!]$ to $[\![e+1,r+1]\!]$ (§ 6.2.3). On the one hand, if $p_i$ A-delivers $[e,r]$ after completing $[e,r+1]$, then any other non-faulty server eventually A-delivers $[e,r]$ (cf. Lemma 6.4.4). On the other hand, if $p_i$ A-delivers $[e,r]$ after a skip transition, then at least one non-faulty server A-delivered $[e,r]$ after completing $[e,r+1]$ (see Figure 6.1). Thus, any other non-faulty server eventually A-delivers $[e,r]$ (cf. Lemma 6.4.4). □

**Theorem 6.4.6** (Set agreement)**.** *If two non-faulty servers A-deliver round r, then both A-deliver the same set of messages.*

*Proof.* Let $p_i$ and $p_j$ be two non-faulty servers that A-deliver round $r$. Clearly, both servers A-deliver $r$ in the same epoch $e$ (cf. Lemma 6.4.5). Thus, we distinguish between $[\![e,r]\!]$ and $[e,r]$. If $[\![e,r]\!]$, both $p_i$ and $p_j$ A-deliver the same set of messages due to the set agreement property of AllConcur (see Section 5.4). If $[e,r]$, both $p_i$ and $p_j$ completed $[e,r]$, i.e., both received messages from all servers; thus, both A-deliver the same set of messages. □

**Theorem 6.4.7** (Agreement)**.** *If a non-faulty server A-delivers m, then all non-faulty servers eventually A-deliver m.*

*Proof.* We prove by contradiction. Let $p_i$ be a non-faulty server that A-delivers $m$ in round $r$ and epoch $e$. We assume there is a non-faulty server $p_j$ that never A-delivers $m$. According to Lemma 6.4.5, $p_j$ eventually A-delivers round $r$ in epoch $e$. Yet, this means $p_j$ A-delivers (in round $r$) the same set of messages as $p_i$ (cf. Theorem 6.4.6), which contradicts the initial assumption. □

**Theorem 6.4.8** (Total order)**.** *If two non-faulty servers $p_i$ and $p_j$ A-deliver messages $m_1$ and $m_2$, then $p_i$ A-delivers $m_1$ before $m_2$, if and only if $p_j$ A-delivers $m_1$ before $m_2$.*

*Proof.* From construction, in AllConcur+, every server A-delivers rounds in order (i.e., $r$ before $r+1$). Also, the messages of a round are A-delivered in a deterministic order. Moreover, according to both Lemma 6.4.5 and Theorem 6.4.6, $p_i$ A-delivers $m_1$ and $m_2$ in the same states as $p_j$. Thus, $p_i$ and $p_j$ A-deliver $m_1$ and $m_2$ in the same order. □

### 6.4.2 Uniformity in AllConcur+

Because of message stability, AllConcur solves *uniform* atomic broadcast (see Section 5.4). However, as described in Section 6.3, AllConcur+ solves non-uniform atomic broadcast—agreement and total order apply only to non-faulty servers. For AllConcur+ to guarantee uniform properties, the concept of message stability can be extended to rounds, i.e., *round stability*—before a server A-delivers a round $r$ in epoch $e$, it must make sure that all non-faulty servers will eventually deliver $r$ in epoch $e$. This is clearly the case if $[\![e,r]\!]$ (due to AllConcur's early termination mechanism). Yet, round stability is not guaranteed when $[e,r]$. Let $p_i$ be a server that A-delivers $[e,r]$ after completing $[e,r+1]$ and then it fails. All the other non-faulty servers receive a failure notification while in $[e,r+1]$ and consequently, rollback to $[\![e+1,r]\!]$. Therefore, all non-faulty servers eventually A-deliver round $r$ in epoch $e+1$, breaking round stability.

To ensure round stability in AllConcur+, $p_i$ A-delivers $[e,r]$ once it receives messages from at least $f$ servers in $[e,r+2]$[5] Thus, at least one non-faulty server A-delivers $[e,r]$ after completing $[e,r+1]$, which guarantees all other non-faulty servers A-deliver $[e,r]$ (cf. Lemma 6.4.4). Note that delaying the A-delivery of unreliable rounds is the cost of providing uniform properties.

To prove both uniform agreement and uniform total order, we adapt AllConcur+'s correctness proof from Section 6.4 to the modification that guarantees round stability.

**Lemma 6.4.9.** *Let $p_i$ be a server that A-delivers $[e,r]$ after completing $[e,r+1]$. Then, any other non-faulty server $p_j$ eventually A-delivers $[e,r]$.*

*Proof.* Due to round stability, at least one server that A-delivered $[e,r]$ after completing $[e,r+1]$ is non-faulty; let $p$ be such a server. Moreover, since $p_i$ A-delivers $[e,r]$ after completing $[e,r+1]$, $p_j$ must have started $[e,r+1]$ (cf. Proposition 6.2.1), and hence, completed $[e,r]$. As a result, $p_j$ either receives no failure notifications, which means it eventually completes $[e,r+1]$ and A-delivers $[e,r]$ (since $n > 2f$), or receives a failure notification and moves to $[\![e+1,r]\!]$ (after $\mathrm{T_{UR}}$). Yet, this failure notification will eventually trigger on $p_i$ a $\mathrm{T_{UR}}$ transition from $[e,r+2]$ to $[\![e+1,r+1]\!]$, which eventually will trigger on $p_j$ a $\mathrm{T_{Sk}}$ transition that leads to the A-delivery of $[e,r]$ (see Figure 6.1). □

**Lemma 6.4.10.** *If a server A-delivers round r in epoch e (i.e., either $[\![e,r]\!]$ or $[e,r]$), then, any non-faulty server eventually A-delivers round r in epoch e.*

*Proof.* If $p_i$ A-delivers $[\![e,r]\!]$ (when completing it), then every non-faulty server eventually A-delivers $[\![e,r]\!]$. The reason is twofold: (1) since $p_i$ completes $[\![e,r]\!]$, every non-faulty server must start $[\![e,r]\!]$ (cf. Proposition 6.2.1); and (2) due to early termination, every non-faulty server eventually also completes and A-delivers $[\![e,r]\!]$.

Otherwise, $p_i$ A-delivers $[e,r]$ either once it completes the subsequent unreliable round or after a skip transition from $[\![e+1,r]\!]$ to $[\![e+1,r+1]\!]$ (§ 6.2.3). On the one hand, if $p_i$ A-delivers $[e,r]$ after completing $[e,r+1]$, then any other non-faulty server eventually A-delivers $[e,r]$ (cf. Lemma 6.4.9). On the other hand, if $p_i$ A-delivers $[e,r]$ after a skip transition, then at least one non-faulty server A-delivered $[e,r]$ after completing $[e,r+1]$. Thus, any other non-faulty server eventually A-delivers $[e,r]$ (cf. Lemma 6.4.9). □

**Corollary 6.4.10.1.** *If two servers A-deliver round r, then both A-deliver r in the same epoch e.*

**Theorem 6.4.11** (Uniform set agreement)**.** *If two servers A-deliver round r, then both A-deliver the same set of messages in round r.*

[5] Clearly, $n > 2f$ in order to avoid livelocks while waiting for messages from at least $f$ servers.

*Proof.* Let $p_i$ and $p_j$ be two servers that A-deliver round $r$. Clearly, both servers A-deliver $r$ in the same epoch $e$ (cf. Corollary 6.4.10.1). Thus, we distinguish between $[[e,r]]$ and $[e,r]$. If $[[e,r]]$, both $p_i$ and $p_j$ A-deliver the same set of messages due to the set agreement property of early termination. If $[e,r]$, both $p_i$ and $p_j$ completed $[e,r]$, i.e., both received messages from all servers; thus, both A-deliver the same set of messages. □

**Theorem 6.4.12** (Uniform agreement)**.** *If a server A-delivers m, then all non-faulty servers eventually A-deliver m.*

*Proof.* We prove by contradiction. Let $p_i$ be a server that A-delivers $m$ in round $r$ and epoch $e$. We assume there is a non-faulty server $p_j$ that never A-delivers $m$. According to Lemma 6.4.10, $p_j$ eventually A-delivers round $r$ in epoch $e$. Yet, this means $p_j$ A-delivers (in round $r$) the same set of messages as $p_i$ (cf. Theorem 6.4.11), which contradicts the initial assumption. □

**Theorem 6.4.13** (Uniform total order)**.** *If two servers $p_i$ and $p_j$ A-deliver messages $m_1$ and $m_2$, then $p_i$ A-delivers $m_1$ before $m_2$, if and only if $p_j$ A-delivers $m_1$ before $m_2$.*

*Proof.* From construction, in AllConcur+, every server A-delivers rounds in order (i.e., $r$ before $r+1$). Also, the messages of a round are A-delivered in a deterministic order. Moreover, according to both Lemma 6.4.10 and Theorem 6.4.11, $p_i$ A-delivers $m_1$ and $m_2$ in the same states as $p_j$. Thus, $p_i$ and $p_j$ A-deliver $m_1$ and $m_2$ in the same order. □

## 6.5 Evaluation

To evaluate AllConcur+, we consider a distributed ledger, a representative application for large-scale atomic broadcast. In a distributed ledger, servers receive transactions as input, and their goal is to agree on a common subset of ordered transactions to be added to the ledger. Moreover, we consider two deployments. First, all the servers are located inside a single datacenter. For example, a group of airplane tickets retailers, located in the same area, want to offer the same flights, without divulging their clients queries. Second, the servers are distributed throughout five data centers across Europe[6]. For example, a group of airplane tickets retailers with clients in multiple countries, distributed the ledger across multiple datacenters in order to provide faster local queries. Henceforth, for brevity, we refer to the single datacenter deployment as SDC and to the multiple datacenters deployment as MDC. For both deployments, we base our evaluation on OMNeT++, a discrete-event simulator [156]. To realistically simulate the communication network, we use the INET framework of OMNeT++.

For every datacenter, we consider a fat-tree network topology [102] that provides support for multi-path routing [6]. The topology consists of three layers of $k$-port switches, resulting in $k$ *pods* connected among each other through $k^2/4$ core switches; every pod is directly connected to $k/2$ subnets of $k/2$ hosts each. In the case of MDC, one port from every core switch is used to stream traffic between datacenters; thus, the number of pods is reduced to $k-1$. Hosts are connected to switches via 1 GigE 10m cables (i.e., $0.05\mu$s delay), while switches are connected to each other via 1 GigE 100m cables (i.e., $0.5\mu$s delay). Datacenters are interconnected via fiber optic (i.e., $5\mu$s delay per km) with an available bandwidth of 10 Gbps (i.e., 10% of the typical datacenter interconnect bandwidth). The length of the fiber optic is estimated as $1.1\times$ the geographical distance between the datacenters, resulting in latencies between 2.5ms and 8.9ms. To reduce the likelihood of correlated failures, we deploy one server per subnet (i.e., $n = k^2/2$ for SDC and $n = 5(k-1)k/2$ for MDC). The servers communicate via TCP.

[6]Dublin, London, Paris, Frankfurt, Stockholm

| $n$ | 8 | 18 | 30 | 32 | 72 | 75 | 128 | 140 | 225 | 242 | 450 | 455 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\kappa$ | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 7 | 8 | 8 |

TABLE 6.2: The vertex-connectivity of the $G_S(n,d)$ digraph for the values of $n$ used throughout the evaluation.

We evaluate AllConcur+ against AllConcur, AllConcur-w/EA, AllGather, LCR [68] and Libpaxos [147]. AllConcur-w/EA is an implementation of AllConcur with support for $\Diamond\mathscr{P}$ (i.e., the forward-backward mechanism described in Section 5.2.1). We deploy both AllConcur and AllConcur-w/EA with a reliability of 6-nines, estimated conservatively over a period of 24 hours with a server $MTTF \approx 2$ years [67]. The servers are connected through a $G_S(n,d)$ digraph [150]; AllConcur+ uses the same digraph in the reliable mode. Table 6.2 shows the vertex-connectivity of $G_S(n,d)$ for the values of $n$ used throughout the evaluation. AllGather is a round-based non-fault-tolerant distributed agreement algorithm where every server uses a binomial tree to A-broadcast its messages; AllConcur+ uses the same mechanism in the unreliable mode. LCR is an atomic broadcast algorithm that is based on a ring topology and uses vector clocks for message ordering. Libpaxos is an open-source implementation of Paxos [92]. To adapt it for distributed agreement, we deploy it over $n$ servers with one proposer, five acceptors (sufficient for a reliability of 6-nines) and $n$ learners.

In our experiments, to simplify our evaluation, we omit from where transactions originate (i.e., from where they are initially received by the servers) and how they are interpreted by the ledger. Every server A-broadcast a message consisting of a batch of transactions; once a server A-delivers a message, it adds all enclosed transactions to (its copy of) the ledger. Each transaction has a size of 250 bytes, sufficient to hold a payload and cryptographic signatures (e.g., a typical size for Bitcoin [124] transactions). Every server can have one outstanding message at a time: Before A-broadcasting another message, it waits either for the round to complete or for the message to be A-delivered. Using this benchmark, we measure both latency and throughput. Latency is defined as the time between a server A-broadcasting and A-delivering a message, while throughput, as the number of transactions A-delivered per server per second.

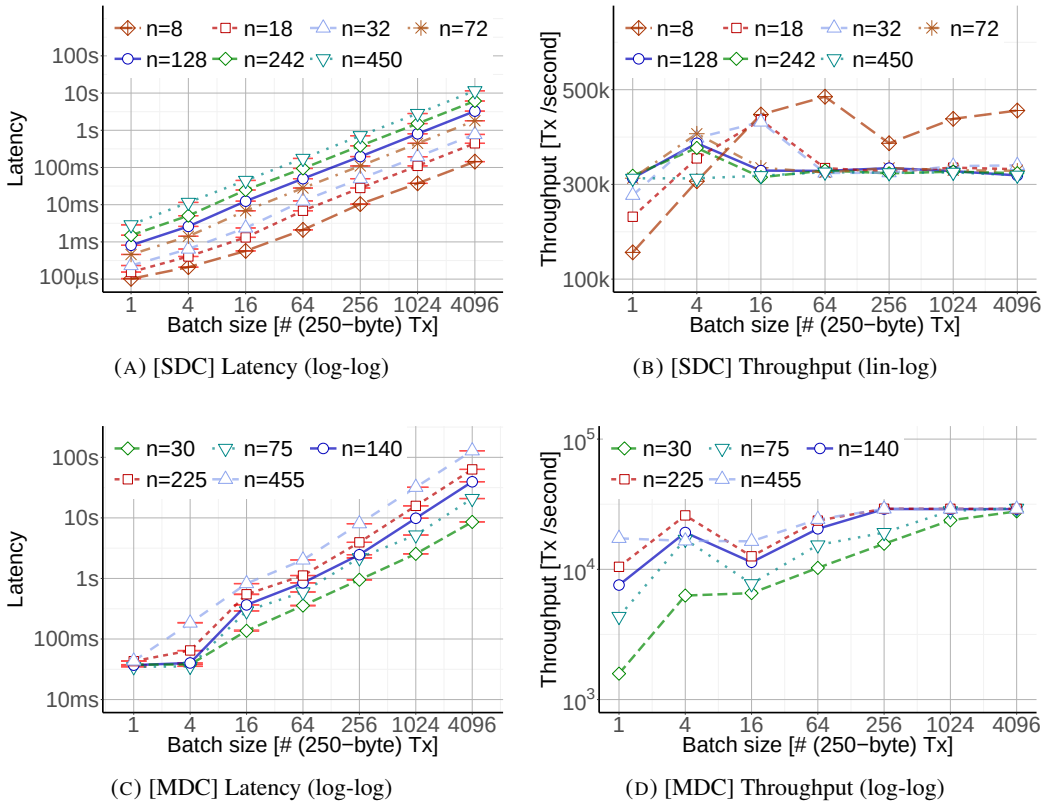
We first evaluate AllConcur+ in non-faulty scenarios (§ 6.5.1). Then, we evaluate the impact of different failure scenarios on AllConcur+'s performance (§ 6.5.2).

### 6.5.1 Non-failure scenarios

We evaluate AllConcur+'s performance in scenarios with no failures for both SDC and MDC. We measure the performance at each server during a common measuring window between $t_1$ and $t_2$; we define $t_1$ and $t_2$ as the time when every server A-delivered at least $10 \times n$ and $110 \times n$ messages, respectively. If servers A-deliver the same amount of messages from every server, as is the case of AllConcur+, then every server A-delivers 100 own messages during a window. Figures 6.5a, 6.5c, 6.6a, and 6.6c report the median latency with a 95% nonparametric confidence interval, while Figures 6.5b, 6.5d, 6.6b, and 6.6d report the average throughput.

#### 6.5.1.1 Message size

We evaluate the effect batching transactions has on AllConcur+'s performance, starting from one transaction per message (i.e., no batching) to 4,096 transactions per message (i.e., $\approx$ 1MB messages). Figure 6.5 plots, for different deployments in both SDC and MDC, the




(A) [SDC] Latency (log-log)

(B) [SDC] Throughput (lin-log)

(C) [MDC] Latency (log-log)

(D) [MDC] Throughput (log-log)

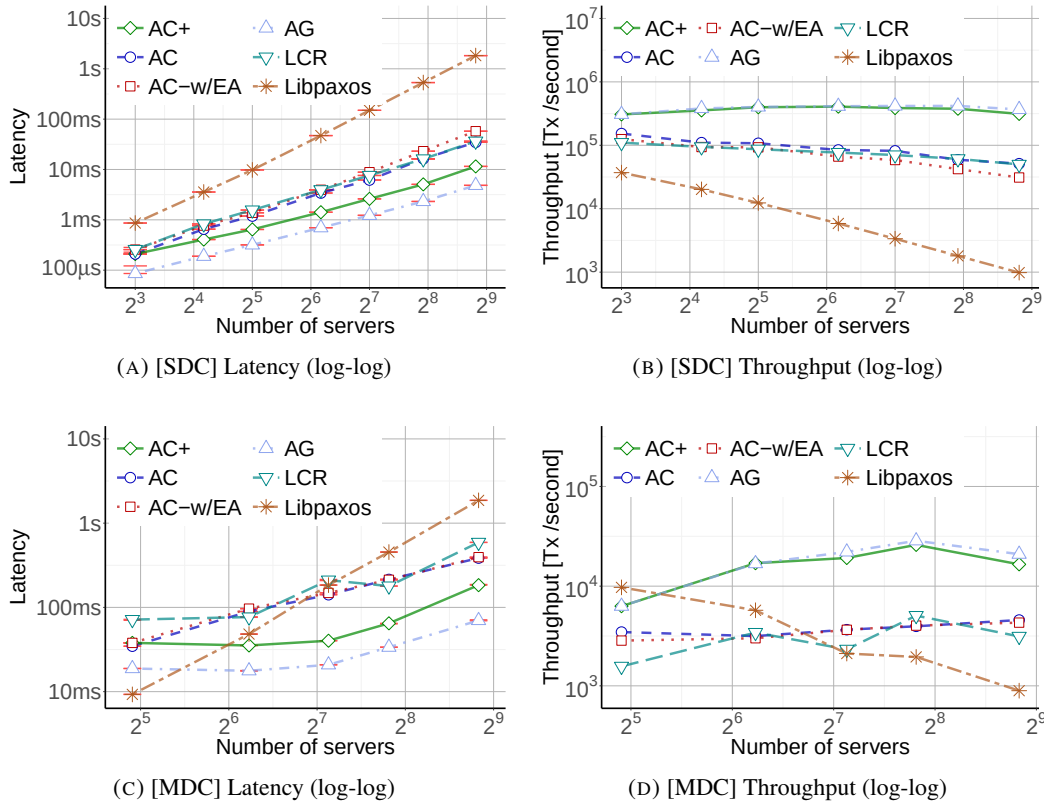
FIGURE 6.5: The effect of batching on AllConcur+'s performance in a non-failure scenario for both a single datacenter [SDC] (a), (b) and multiple datacenters [MDC] (c), (d). The measurements are done using OMNeT++. The latency is reported as the median with a 95% nonparametric confidence interval; the throughput is reported as the average over the measuring window.

latency and the throughput as a function of batching size. As expected, the latency is sensitive to increases in message size. Without batching multiple transactions into a single message, the latency is minimized. Yet, no batching entails usually a low throughput, since the system's available bandwidth is only saturated for large system sizes (e.g., $\approx 450$ servers). Indeed, increasing the message size leads to higher throughput: By batching transactions, AllConcur+'s throughput exceeds 320,000 transactions per second for all SDC deployments, and 27,000 transactions per second for all MDC deployments (see Figures 6.5b and 6.5d). This increase in throughput comes though at the cost of higher latency.

Moreover, increasing the batch size may lead to higher latency due to the TCP protocol. For example, in an MDC deployment of 30 servers with a batch size of 16, the 65,535-byte TCP Receive Window causes servers to wait for TCP packets to be acknowledged before being able to send further all the messages of a round. Since acknowledgements across datacenters are slow, this results in a sharp increase in latency and thus, a drop in throughput (see Figures 6.5c and 6.5d). To reduce the impact of TCP, for the remainder of the evaluation, we fix the batch size to four (i.e., 1kB messages).

#### 6.5.1.2 Comparison to other algorithms

We evaluate AllConcur+'s performance against AllConcur, AllConcur-w/EA, AllGather, LCR and Libpaxos, while scaling up to 455 servers. Figure 6.6 plots, for both SDC and MDC, the latency and the throughput as a function of the number of servers.




(A) [SDC] Latency (log-log)

(B) [SDC] Throughput (log-log)

(C) [MDC] Latency (log-log)

(D) [MDC] Throughput (log-log)

FIGURE 6.6: AllConcur+'s performance evaluated against AllConcur, AllConcur-w/EA, AllGather, LCR and Libpaxos, in a non-failure scenario with a batch size of four, (i.e., 1kB messages) for both a single datacenter [SDC] (a), (b) and multiple datacenters [MDC] (c), (d). The measurements are done using OMNeT++. The latency is reported as the median with a 95% nonparametric confidence interval; the throughput is reported as the average over the measuring window.

**AllConcur+ vs. AllGather.** For both deployments, AllConcur+'s latency is around 2–2.6× higher than AllGather's. The roughly[7] two-fold increase in latency is as expected: When no failures occur, AllConcur+ A-delivers a message after completing two rounds. At the same time, AllConcur+ achieves between 79% and 100% of the throughput of AllGather, while providing also fault tolerance. The reason behind this high throughput is that, similar to AllGather, AllConcur+ A-delivers messages at the end of every round (except for the first round).

**AllConcur+ vs. AllConcur.** Due to the redundancy-free overlay network, AllConcur+ performs less work and introduces less messages in the network and, as a result, outperforms both AllConcur and AllConcur-w/EA. When comparing to AllConcur, it is up to 3.2× faster for SDC and up to 3.5× faster for MDC; also, it achieves up to 6.4× higher throughput for SDC and up to 6.5× higher throughput for MDC. When comparing to AllConcur-w/EA, it is up to 5× faster for SDC and up to 3.7× faster for MDC; also, it achieves up to 10× higher throughput for SDC and up to 6.5× higher throughput for MDC.

**AllConcur+ vs. LCR.** AllConcur+ outperforms LCR in both latency and throughput. The reason behind it is twofold: first, the dissemination latency of the ring topology adopted

[7]The overhead is due to providing fault tolerance (e.g., larger message headers, more TCP connections).

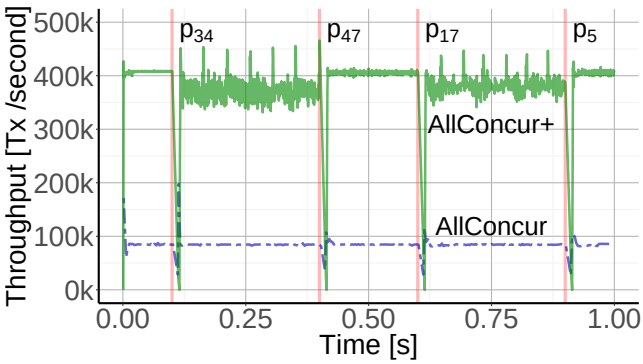



FIGURE 6.7: AllConcur+'s performance evaluated against AllConcur during a failure scenario for a single datacenter deployment of 72 servers, each A-broadcasting 1kB messages. The red vertical bars denote the time of failures. The FD has $\Delta_{hb} = 1ms$ and $\Delta_{to} = 10ms$. The measurements are done using OMNeT++. The reported throughput is sampled for server $p_0$ at the completion of every round.

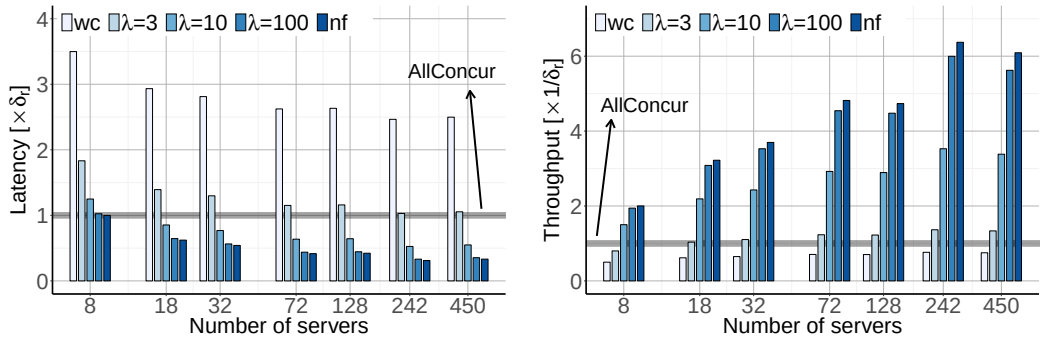
by LCR; and second, the message overhead necessary for using vector clocks. Thus, for SDC, AllConcur+ is up to 3.2× faster and achieves up to 6.3× higher throughput. Although LCR is designed for local area networks [68], for completion, we evaluate its performance also for MDC: AllConcur+ is up to 5.2× faster and achieves up to 8.3× higher throughput. When deploying LCR, we order the servers in the ring with minimal communication between different datacenters.

**AllConcur+ vs. Libpaxos.** Paxos is designed for data replication [144] and not intended to scale to hundreds of instances. Accordingly, the performance of Libpaxos drops sharply with increasing $n$. As a result, AllConcur+ is up to 158× faster for SDC and up to 10× faster for MDC; also, it achieves up to 318× higher throughput for SDC and up to 18× higher throughput for MDC.

### 6.5.2 Failure scenarios

To evaluate AllConcur+'s performance when failures do occur, we first consider the following failure scenario: In an SDC deployment of 72 servers, each A-broadcasting 1kB messages, four failures occur during an interval of one second. To detect failures, servers rely on a heartbeat-based FD with a heartbeat period $\Delta_{hb} = 1ms$ and a timeout period $\Delta_{to} = 10ms$. Figure 6.7 plots the throughput of server $p_0$ as a function of time for both AllConcur and AllConcur+. The throughput is sampled at the completion of every round, i.e., the number of transactions A-delivered divided by the time needed to complete the round.

The four failures (indicated by red vertical bars) have more impact on AllConcur+'s throughput than AllConcur's. In AllConcur, a server's failure leads to a longer round, i.e., $\approx \Delta_{to}$ instead of $\approx$ 3.3ms when no failures occur (see Figure 6.6a). For example, when $p_{34}$ fails, $p_0$ must track its message, which entails waiting for each of $p_{34}$'s successors to detect its failure and send a notification. Once the round completes, AllConcur's throughput returns to $\approx 85,000$ transactions per second. In AllConcur+, every failure triggers a switch to the reliable mode. For example, once $p_0$ receives a notification of $p_{34}$'s failure, it rolls back to the latest A-delivered round and reliably reruns the subsequent round. Once $p_0$ completes the reliable round, it switches back to the unreliable mode, where it requires two unreliable rounds to first A-deliver a round. In Figure 6.7, we see AllConcur+'s throughput drop for a short interval of time after every failure (i.e., $\approx$ 16ms). However, since AllConcur+'s throughput




(A) Expected latency (B) Expected throughput

FIGURE 6.8: AllConcur+'s estimated expected latency and throughput (expressed as ratios to AllConcur's values) for the non-failure scenario (nf), the worst-case scenario (wc), and scenarios with sequences of $\lambda$ unreliable rounds (ordered by decreasing frequency of failures). We assume an SDC deployment with batch size four.

is significantly higher in general, the short intervals of low throughput after failures have only a minor impact and on average, AllConcur+ achieves $\approx 4.6\times$ higher throughput than AllConcur.

Furthermore, we analyze the robustness of AllConcur+'s performance with regard to more frequent failures. The analysis is based on a model with two parameters: $\delta_u$, the expected duration of an unreliable round; and $\delta_r$, the expected duration of a reliable round. Clearly, $\delta_u < \delta_r$. Since every server has one outstanding message at a time, we use data from Figure 6.6 to estimate both parameters: $\delta_u$ is half the latency of AllConcur+ and $\delta_r$ is the latency of AllConcur. We assume an SDC deployment with batch size four. We first focus on both the expected latency and throughput over a sequence of multiple rounds. Then, for latency sensitive applications, we look into the worst-case latency of a single round. Throughout the analysis, we consider only no-fail and fail transitions, since skip transitions entail both lower latency and higher throughput, i.e., a skip transition always triggers the A-delivery of messages.

**Expected performance.** We estimate AllConcur+'s expected performance for several scenarios with different expected failure frequencies. If failures occur so frequently that no unreliable rounds can be started (i.e., a single sequence of reliable rounds), AllConcur+ is equivalent to AllConcur—a round is A-delivered once it is completed. Thus, the estimate of the expected latency is $\delta_r$ and of the expected throughput is $1/\delta_r$. For comparison, Figure 6.8 plots the latency as a factor of $\delta_r$ and the throughput as a factor of $1/\delta_r$. If no failures occur, the algorithm performs a sequence of unreliable rounds; for this scenario (denoted by nf), we use the actual measurements, reported in Figure 6.6a.

The worst-case scenario (denoted by wc), requires all messages to be A-delivered during reliable rounds and, in addition, in between any two subsequent reliable rounds there must be exactly two unreliable rounds (the second one not being completed), i.e., the longest sequence possible without A-delivering an unreliable round. Such a scenario consists of the repetition of the following sequence:

$$\begin{aligned} &\ldots \to [e, r-1] \to [e, r] \xrightarrow{\text{fail}} [[e+1, r-1]] \to \\ &\quad [e+1, r] \to [e+1, r+1] \xrightarrow{\text{fail}} [[e+2, r]] \to \ldots \end{aligned} \tag{6.1}$$

Thus, the expected latency is $3\delta_u + 2\delta_r$, e.g., $r$'s messages are A-broadcast in $[e, r]$ but A-delivered once $[[e+1, r]]$ completes. This scenario provides also the worst-case expected

throughput—only reliable rounds are A-delivered, hence $1/(2\delta_u + \delta_r)$. In the worst case, AllConcur+ has up to $3.5\times$ higher expected latency and up to $2\times$ lower expected throughput than AllConcur.

The low performance of the worst-case scenario is due to the lack of sufficiently long sequences of unreliable rounds (i.e., at least three, two completed and one begun), thus never enabling the minimal-work distributed agreement of AllConcur+. Let $\lambda \geq 3$ be the length of each sequence of unreliable rounds, i.e., the expected frequency of failures is between $1/(\delta_r + \lambda\delta_u)$ and $1/(\delta_r + (\lambda-1)\delta_u)$. Then, the expected latency is $2\delta_u + \frac{\delta_u + 2\delta_r}{\lambda}$, i.e., the first $\lambda - 2$ unreliable rounds are A-delivered after $2\delta_u$, the $(\lambda-1)$-th round after $2\delta_u + \delta_r$ and the final round after $3\delta_u + \delta_r$. Also, the expected throughput is $\frac{1-1/\lambda}{\delta_u + \delta_r/\lambda}$, i.e., during a period of $\lambda\delta_u + \delta_r$, $\lambda - 2$ unreliable rounds and one reliable round are A-delivered. For $\lambda = 10$, AllConcur+ has up to $1.9\times$ lower expected latency and up to $3.5\times$ higher expected throughput than AllConcur.

**Worst-case latency for a single round.** Let $r$ be a round and $e$ be the epoch in which $r$'s messages were A-broadcast for the first time. If $[\![e,r]\!]$, then $r$ is A-delivered once $[\![e,r]\!]$ completes (as in AllConcur); thus, the estimated latency is $\delta_r$. Yet, if $[e,r]$, the A-delivery of $r$ must be delayed (§ 6.2.3). If no failures occur, $[e,r]$ will be followed by two unreliable rounds; thus, the estimated latency is $2\delta_u$. In case of a single failure, the worst-case estimated latency is $3\delta_u + \delta_r$ and it corresponds to the sequence in (6.1), with the exception that $[e+1, r+1]$ completes i.e., $[e,r]$ is succeeded by a fail transition to $[\![e+1, r-1]\!]$, a reliable rerun of the preceding not A-delivered unreliable round $r-1$; then, due to no other failures, round $r$ is rerun unreliably in epoch $e+1$ and A-delivered once $[e+1, r+1]$ completes. If multiple failures can occur, $[e+1, r+1]$ may be interrupted by a fail transition to $[\![e+2, r]\!]$; thus, in this case, the worst-case estimated latency is $3\delta_u + 2\delta_r$.

For latency sensitive applications, we can reduce the worst-case latency by always rerunning rounds reliably (after a rollback). For the sequence in (6.1), $[\![e+1, r-1]\!]$ would then be followed by $[\![e+2, r]\!]$. In this case, the worst-case estimated latency is $\delta_u + 2\delta_r$. Note that if $\delta_r > 2\delta_u$, then in the case of a single failure, rerunning reliably is more expensive, i.e., $\delta_u + 2\delta_r$ as compared with $3\delta_u + \delta_r$. Another optimization is to rerun all (not A-delivered) unreliable rounds in one single reliable round. Let $\overline{\delta_r}$ denote the expected time of such a round; then, the worst-case estimated latency is $2\delta_u + \overline{\delta_r}$. Note that if a skip transition is triggered while two rounds $r-1$ and $r$ are rerun in one reliable round, only round $r-1$ must be A-delivered.

# Chapter 7

# Conclusion

State-machine replication enables the implementation of fault-tolerant distributed services that guarantee strong consistency. To provide strong consistency it is necessary to establish total order of the requests, which requires the execution of a distributed agreement algorithm, such as consensus or atomic broadcast. Consequently, guaranteeing strong consistency adds a considerable performance overhead to the distributed service. The goal of this dissertation is to come up with efficient algorithms that reduce this overhead. As a result, we designed, implemented, and evaluated three novel algorithms that enable the implementation of high-performance state-machine replication.

All three algorithms are designed with enhancing performance as the main objective. However, they target different use cases. The first algorithm, DARE, is designed for use cases, where replication is employed as a means of providing availability and reliability. Such use cases require only a handful of replicas, rendering a leader-based approach suitable. The remaining two algorithms, AllConcur and AllConcur+, are designed for use cases, where replication is a requirement of the application and scaling out across hundreds of servers is not uncommon (e.g., a distributed ledger servicing users world-wide). At these scales, leader-based approaches are no longer suitable. Therefore, we designed both AllConcur and AllConcur+ to be leaderless.

## 7.1 DARE

DARE is a novel leader-based algorithm that uses RDMA features in novel, atypical ways, in order to push the limits of high-performance state-machine replication. Thirty years of extensive research have yielded robust and well understood leader-based algorithms that enable state-machine replication, such as Paxos [92] and Raft [130]. However, these algorithms rely on message-passing for communication. In general, adapting a message-passing algorithm to the one-sided communication model enabled by RDMA raises several challenges.

First, in RDMA, servers can remotely access the memory of other servers, but not their stable storage. This means that a server updating a remote data item, which is also stored on disk, leads to inconsistencies. DARE eliminates the need of stable storage by using raw replication—the persistent state of every server is replicated in the memory of multiple servers. In addition, DARE's in-memory approach improves the overall performance of state-machine replication by eliminating high-latency accesses to on-disk data.

Second, the one-sided RMDA operations bypass the remote CPUs, which means that the targets are unaware of write accesses to their local memories. This means that a target, which must take some action as a result of a remote access, must periodically poll its memory. However, polling too often, consumes too much CPU time, while not polling often enough, may delay that action. DARE removes memory polling from the critical path of data replication—although the servers poll the logs for new entries in order to apply them to the state machine, the leader can proceed with replicating the log without waiting for them. In other words, DARE is a wait-free, direct-access algorithm.

Finally, since servers can directly access remote memory, it is possible for both read-write and write-write races to occur. This is unlike message-passing, where every server has exclusive access to its own memory. In DARE, servers access remotely two data structures: the circular log and the control data arrays. To avoid read-write races in the circular log, the leader has access only to the log region between the commit and head pointers, while the local server (owning the log) has access only to the opposite region. Moreover, write-write races are avoided by guaranteeing one leader at a time. To enforce this, DARE uses QP disconnects—only the current leader has write access to the logs. In the case of the control data arrays, write-write races are eliminated directly, since any given array entry only has a single writer. Read-write races are avoided either by having monotonic array entries [80] (e.g., the heartbeat array) or by granting exclusive access to the owner of each entry (e.g., the private data array).

The design of DARE makes several research contributions. First, to the best of our knowledge, DARE is the first state-machine replication algorithm to be designed entirely with one-sided RDMA semantics, allowing it to exploit the full potential of RDMA-capable networks. As a result, at the time of its publication, our DARE implementation over high-performance InfiniBand Verbs improved the performance over state of the art state-machine replication algorithms by more than an order of magnitude. Second, in DARE, the leader is solely responsible for replicating the log; the other servers are mostly idle. This is unlike message-passing algorithms, where non-leader servers need to receive message, access the local memory, and send replies to the leader. DARE's sole-leader approach has two main benefits: (1) it simplifies the state-machine replication algorithms, since instinctively, data replication is a one-sided operation; and (2) non-leader servers are available for other tasks, such as the recovery of another server. Finally, the in-memory RDMA-based design of DARE leads to a different view of a failing system than message-passing. In message-passing approaches, a failure of the CPU results in the entire server being inaccessible. In DARE, servers with a faulty CPU, but with both NIC and memory working, are still remotely accessible during log replication.

In summary, the design of DARE addresses fundamental challenges regarding adapting a message-passing state-machine replication algorithm to RDMA semantics. These challenges are general and applicable to the design of other RDMA-based algorithms. Moreover, DARE has already served as a basis for the design of more recent algorithms that use RDMA to enhance the performance of state-machine replication, such as APUS [161]. In general, using one-sided RDMA operations, instead of message-passing, enables fast coordination services, such as configuration management.

## 7.2 AllConcur / AllConcur+

Both AllConcur and AllConcur+ are leaderless concurrent atomic broadcast algorithms that are designed with the purpose of scaling out state-machine replication across hundreds of servers, while achieving high performance. Most common practical approaches to implement state-machine replication are leader-based, i.e., they rely on algorithms such as Paxos, Zab [84], and Raft. The reasons are twofold. First, leader-based algorithms are easier to understand—there is a central component that serializes multiple streams of data. Second, the so far most common use cases of state-machine replication require only a handful of servers, making leader-based approaches suitable. However, leader-based algorithms entail heavy workloads on the leader and thus, do not perform well at large scales.

Building a leaderless atomic broadcast algorithm that achieves high-performance at large scales raises several challenges. First, having no leader raises the obvious question of how to establish total order. Excluding approaches that require the work per A-broadcast message

to be linear in the number of participating servers, we remain with two classes of algorithms [47]: deterministic merge and destinations agreement. AllConcur and AllConcur+ are a mixture of both classes—the execution is split into rounds, messages are timestamped with the round number, and in every round, the servers agree on a set of messages to A-deliver.

Second, since, in each round, every server must collect a set of messages, another challenge is to decide in what conditions can a server stop collecting messages for a certain round. The deterministic merge approach entails the set should contain a message from every server (except maybe for faulty servers). This gives us a straightforward condition to stop collecting messages—receiving messages from all the servers. However, failures may lead to messages being lost, which means that this condition may never be fulfilled. In such scenarios, the following question arises: How long should a server wait for a message before it can safely decide that it will never arrive? Both AllConcur and AllConcur+ address this question by using an early termination mechanism, which allows servers to track the whereabout of messages and consequently, identify lost messages. Moreover, early termination entails that servers must not wait for the worst case and thus, it reduces the expected number of communication steps significantly.

Finally, a leaderless approach is not sufficient for achieving high performance. For instance, we already excluded causal history algorithms, which, although leaderless, require the size of each message to be linear in the number of servers and therefore, are unsuitable for large-scale deployments. Similarly unsuitable are the atomic broadcast algorithms that adopt an all-to-all pattern for server communication, since the work for sending a message is again linear. Both AllConcur and AllConcur+ model the communication between servers by an overlay network described by a sparse digraph, i.e., each server sends messages only to its successors, which are considerably less than the total number of servers. In addition, this communication model enables a more efficient mechanism for failure detection, since every server monitors only its predecessors instead of all the servers in the system.

To reliably disseminate messages, the sparse digraph used by AllConcur for communication must also be resilient (i.e., its vertex-connectivity exceeds the maximum number of failure tolerated). This resiliency comes at the cost of redundancy, which introduces a lower bound on the work per A-broadcast message. In general, this redundancy is needed since the frequency of failures in distributed systems makes non-fault-tolerant services unfeasible. However, intervals with no failures are common enough to motivate a less conservative approach. Therefore, AllConcur+ uses a redundancy-free overlay network during intervals with no failures and, when failures do occur, it automatically recovers by switching to the resilient overlay network. Nonetheless, designing this dual digraph approach raises another series of challenges.

First, the dual digraph approach entails two modes, which means also two round types—AllConcur+ distinguishes between reliable and unreliable rounds. As in AllConcur, the completion of a reliable round leads to the messages received (during that round) being A-delivered. Yet, this is not the case with unreliable rounds. When failures occur, AllConcur+ must rollback to the latest A-delivered round and, as a result, may discard an already completed unreliable round. To guarantee that an A-delivered round can never be discarded, AllConcur+ requires that an unreliable round is A-delivered once the subsequent unreliable round completes as well. It is important though to notice that this extra round of delay is a small price to pay in order to reduce the cost of providing fault-tolerance: This delay does not affect the throughput of atomic broadcast; moreover, for the common case where one reliable round takes longer than two subsequent unreliable rounds, AllConcur+ achieves lower latency than AllConcur.

Second, having two types of rounds complicates the design of an atomic broadcast algorithm. In general, the understandability of a distributed system is improved by modeling its execution as an ordered sequence of states. For instance, in AllConcur, servers progress

through an ordered sequence of rounds. However, due to the rollbacks caused by failures, the sequence of rounds in AllConcur+ is not ordered. To simplify its design, AllConcur+ introduces epochs, which, in a nutshell, consist each of a reliable round followed by a sequence of zero or more unreliable rounds. This enables us to model the execution of AllConcur+ also as an ordered sequence of states, where the states are obtained by pairing rounds together with their corresponding epochs.

Finally, designing a distributed system, where the participating servers are asynchronously transitioning through a series of states, is not a straightforward endeavor. For instance, at any given time, any two AllConcur+ servers can be in different rounds or in different epochs; moreover, a subset of the servers may entirely skip a state other servers went through. To facilitate the design of AllConcur+, we determine four concurrency properties, one asserting the uniqueness of a state and the other three specifying what messages a server can receive.

AllConcur and AllConcur+ make several research contributions. First, the digraph-based communication model adopted by both AllConcur and AllConcur+ enables a trade-off between reliability and performance: The fault tolerance is given by the vertex-connectivity of the (reliable) digraph used by servers to communicate and thus, it can be adapted to system-specific requirements; moreover, it can be dynamically adjusted during execution. Second, the digraph-based communication model also enables a trade-off between low latency and high throughput—using digraphs with low diameter (e.g., a complete graph), reduces the latency of communication, while using digraphs with small degree (e.g., a circular graph) increase throughput. For disseminating messages reliably, our AllConcur and AllConcur+ implementations use a digraph that has a quasiminimal diameter [150] and is optimally connected, i.e., for the provided fault tolerance, the degree is minimal. Moreover, during intervals with no failures, AllConcur+ servers use binomial trees to A-broadcast their messages, i.e., both diameter and degree are bounded by $\log n$, with $n$ the number of servers. Third, the early termination mechanism enables both algorithms to avoid assuming always the worst case and therefore, the expected number of communication steps are reduced significantly. Finally, the dual-digraph approach of AllConcur+ further enhances the performance of atomic broadcast. By leveraging redundancy-free execution during intervals with no failures, AllConcur+ achieves significantly higher throughput and lower latency than both AllConcur and other state of the art atomic broadcast algorithms.

In general, by enabling the implementation of large-scale high-performance state-machine replication, both AllConcur and AllConcur+ make it possible to scale out modern applications, such as distributed ledgers, to hundreds of servers, while maintaining a high quality of service.

## 7.3 Future directions

We conclude this dissertation by suggesting some possible directions for future research:

**Implementation of an open-source state-machine replication library.** To demonstrate the enhancement of state-machine replication performance enabled by our designs, we have implemented all three algorithms. We provide an open-source reference implementation of DARE over high-performance InfiniBand Verbs. Furthermore, we implemented both AllConcur and AllConcur+. AllConcur's implementation supports standard sockets-based TCP as well as high-performance InfiniBand Verbs communications, while AllConcur+ is implemented within the OMNeT++ discrete-event simulator. Although these implementations are extensive, including details such as membership changes or recovery after failures, and can

serve as a basis for software developers, they are merely prototypes meant to asses the mechanisms presented in this dissertation and therefore, not intended for deployments in production systems. The implementation and maintenance of an open-source state-machine replication library that brings together DARE, AllConcur, and AllConcur+ under a well-defined API would facilitate the evaluation of these algorithms in real-world distributed systems.

**Optimizations.** The three algorithms we present are designed with one main objective in mind—to push the limits of high-performance state-machine replication. We achieve this through various mechanisms, such as using RDMA techniques in atypical ways, adopting a communication model based on a sparse digraph, and employing a dual-digraph approach. Nonetheless, the potential for further enhancing the performance of state-machine replication is not exhausted. As already demonstrated by APUS, optimizing state-machine replication to be scalable on multiple concurrent client connections improves the overall performance. DARE's implementation can be adapted (through pipelining) to concurrently service multiple clients. Moreover, we believe that some typical state-machine replication optimizations, such as distributing the load by allowing multiple servers to answer requests and answering not-interfering requests out-of-order, could be added to DARE's design with moderate effort. In the case of both AllConcur and AllConcur+, a thought-provoking optimization is to find an overlay network that can be mapped on the physical network with minimizing the communication latency as main objective; for instance, such an optimization would attempt to reduce the amount of data exchanges across datacenters.

**Byzantine fault tolerance.** In this dissertation, we assume a fail-stop failure model, which means that, once a server fails, it no longer affects the operation of the other servers. Many distributed systems can be modeled this way. First, servers that recover after failing can be added back to the system through a membership change mechanism. Second, even if servers are falsely suspected of having failed, the fail-stop model could be enforced through process controlled crashes. However, the fail-stop model cannot account for arbitrary failures, such as software errors or malicious behaviors. Therefore, designing high-performance state-machine replication algorithms that can tolerate Byzantine failures may be a worthwhile endeavor. This is especially true for large-scale deployments, where such algorithms could facilitate, for instance, the implementation of efficient Byzantine fault tolerant (BFT) distributed ledgers. Today's BFT distributed ledgers are implemented either through BFT replication, which typically has limited scalability, or through proof-of-work based consensus, which typically achieves limited performance [160]. Yet, adapting algorithms, such as AllConcur, to Byzantine fault tolerance entails several challenges, such as devising an early termination mechanism that can tolerate Byzantine failures.

# Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, Boston, MA, USA, 2002. ACM.

[2] M. K. Aguilera. *Stumbling over Consensus Research: Misunderstandings and Issues*, pages 59–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] M. K. Aguilera and S. Toueg. A simple bivalency proof that t-resilient consensus requires t+1 rounds. *Information Processing Letters*, 71(3):155 – 158, 1999.

[4] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[7] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, Santa Barbara, CA, USA, 1995. ACM.

[8] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem Single-ring Ordering and Membership Protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, November 1995.

[9] E. Anceaume. A Lightweight Solution to Uniform Atomic Broadcast for Asynchronous Systems. In *Proc. 27th International Symposium on Fault-Tolerant Computing*, FTCS '97, Seattle, WA, USA, 1997. IEEE Computer Society.

[10] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proc. 13th EuroSys Conference*, EuroSys '18, Porto, Portugal, 2018. ACM.

[11] T. Angskun, G. Bosilca, and J. Dongarra. Binomial Graph: A Scalable and Fault-tolerant Logical Network Topology. In *Proc. 5th International Conference on Parallel and Distributed Processing and Applications*, ISPA'07, Niagara Falls, ON, Canada, 2007. Springer-Verlag.

[12] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., USA, 2004.

[13] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. Conference on Innovative Data system Research (CIDR)*, 2011.

[14] Z. Bar-Joseph, I. Keidar, and N. A. Lynch. Early-Delivery Dynamic Atomic Broadcast. In *Proc. 16th International Symposium on Distributed Computing*, DISC '02, Toulouse, France, 2002. Springer-Verlag.

[15] J. Behrens, S. Jha, K. Birman, and E. Tremel. RDMC: A Reliable RDMA Multicast for Large Objects. In *Proc. 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2018, Luxembourg City, Luxembourg, June 2018.

[16] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003®. In *Proc. ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, Portland, OR, USA, 2004. ACM.

[17] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling Large-scale, High-speed, Peer-to-peer Games. In *Proc. ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, Seattle, WA, USA, 2008. ACM.

[18] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *Proc. 3rd Conference on Networked Systems Design & Implementation*, NSDI'06, San Jose, CA, USA, 2006. USENIX Association.

[19] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[20] K. P. Birman. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer Publishing Company, Incorporated, 2012.

[21] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[22] Blizzard Entertainment. WoW PvP battlegrounds, 2008. `http://www.worldofwarcraft.com/pvp/battlegrounds`.

[23] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, March 2003.

[24] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, Boston, MA, USA, 2011. USENIX Association.

[25] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The Primary-backup Approach*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[26] D. Buntinas. Scalable Distributed Consensus to Support MPI Fault Tolerance. In *Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS'12, Shanghai, China, 2012.

[27] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, Seattle, WA, USA, 2006.

[28] C. P. Burton. Replicating the Manchester Baby: motives, methods, and messages from the past. *IEEE Annals of the History of Computing*, 27(3):44–60, July 2005.

[29] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Cascais, Portugal, 2011. ACM.

[30] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, 1(2):74–85, 1985.

[31] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, New Orleans, LA, USA, 1999.

[32] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, Portland, OR, USA, 2007. ACM.

[33] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, Philadelphia, PA, USA, 1996. ACM.

[34] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, March 1996.

[35] J. M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[36] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[37] G. V. Chockler, N. Huleihel, and D. Dolev. An Adaptive Totally Ordered Multicast Protocol That Tolerates Partitions. In *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, Puerto Vallarta, Mexico, 1998. ACM.

[38] F. R. K. Chung and M. R. Garey. Diameter bounds for altered graphs. *Journal of Graph Theory*, 8(4):511–534, December 1984.

[39] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10, Indianapolis, IN, USA, 2010. ACM.

[40] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[41] A. Correia, J. Pereira, L. Rodrigues, N. Carvalho, and R. Oliveira. *Practical Database Replication*, pages 253–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[42] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA+ Proofs. In *Proc. 18th International Symposium On Formal Methods*, FM'12, Paris, France, 2012. Springer.

[43] F. Cristian. Reaching agreement on processor-group membrship in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, December 1991.

[44] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[45] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[46] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, December 2007.

[47] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[48] A. H. Dekker and B. D. Colbert. Network Robustness and Graph Topology. In *Proc. 27th Australasian Conference on Computer Science - Volume 26*, ACSC '04, Dunedin, New Zealand, 2004. Australian Computer Society, Inc.

[49] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, FTCS '23, Toulouse, France, June 1993.

[50] D. Dolev and C. Lenzen. Early-deciding Consensus is Expensive. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, Montréal, QC, Canada, 2013. ACM.

[51] D. Dolev, R. Reischuk, and H. R. Strong. Early Stopping in Byzantine Agreement. *J. ACM*, 37(4):720–741, October 1990.

[52] J. Domke, T. Hoefler, and S. Matsuoka. Fail-in-place Network Design: Interaction Between Topology, Routing Algorithm and Failures. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 597–608, New Orleans, LA, USA, 2014. IEEE Press.

[53] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proc. 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Seattle, WA, USA, 2014.

[54] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. 25th Symposium on Operating Systems Principles*, SOSP '15, Monterey, CA, USA, 2015. ACM.

[55] D. Z. Du and F. K. Hwang. Generalized De Bruijn Digraphs. *Netw.*, 18(1):27–38, March 1988.

[56] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, April 1988.

[57] R. Ekwall, A. Schiper, and P. Urban. Token-based Atomic Broadcast Using Unreliable Failure Detectors. In *Proc. 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, Florianpolis, Brazil, 2004. IEEE Computer Society.

[58] P. D. Ezhilchelvan, R. A. Macedo, and S. K. Shrivastava. Newtop: A Fault-tolerant Group Communication Protocol. In *Proc. 15th International Conference on Distributed Computing Systems*, ICDCS '95, Vancouver, BC, Canada, 1995. IEEE Computer Society.

[59] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

[60] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5, August 2004.

[61] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Vancouver, BC, Canada, 2010.

[62] T. Friedman and R. Van Renesse. Packing Messages As a Tool for Boosting the Performance of Total Ordering Protocols. In *Proc. 6th IEEE International Symposium on High Performance Distributed Computing*, HPDC '97, Portland, OR, USA, 1997. IEEE Computer Society.

[63] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, pages 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[64] E. Gafni and L. Lamport. Disk Paxos. *Distrib. Comput.*, 16(1):1–20, February 2003.

[65] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, Denver, CO, USA, 2013. ACM.

[66] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, August 2011.

[67] Global Scientific Information and Computing Center. Failure History of TSUBAME2.0 and TSUBAME2.5, 2014. `http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm`.

[68] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[69] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.

[70] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and Tolerating Heterogeneous Failures in Large Parallel Systems. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 45:1–45:11, Seattle, WA, USA, 2011. ACM.

[71] D. Hensgen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, February 1988.

[72] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[73] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, Austin, TX, USA, 2015. ACM.

[74] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[75] T. Hoefler and D. Moor. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Supercomput. Front. Innov.: Int. J.*, 1(2):58–75, July 2014.

[76] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. 2010 USENIX Annual Technical Conference*, ATC'10, Boston, MA, USA, 2010. USENIX Association.

[77] InfiniBand Trade Association. *InfiniBand Architecture Specification: Volume 2, Release 1.3.1*. 2016.

[78] G. Intelligence. Number of Mobile Subscribers Worldwide Hits 5 Billion, 2017. `https://www.gsma.com/newsroom/press-release/number-mobile-subscribers-worldwide-hits-5-billion/`.

[79] M. Isard. Autopilot: Automatic Data Center Management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, April 2007.

[80] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, S. Zink, K. P. Birman, and R. Van Renesse. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst. (accepted)*, 2019.

[81] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Proc. Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. ACM.

[82] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *Proc. 2012 12th IEEE/ACM International Symposium on Cluster, Cloud*

*and Grid Computing*, CCGRID '12, Ottawa, ON, Canada, 2012. IEEE Computer Society.

[83] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proc. 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Taipei, Taiwan, 2011.

[84] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance Broadcast for Primary-backup Systems. In *Proc. 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, Hong Kong, China, 2011. IEEE Computer Society.

[85] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. 11th International Conference on Distributed Computing Systems*, May 1991.

[86] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proc. 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, Chicago, IL, USA, 2014. ACM.

[87] I. Keidar and D. Dolev. *Totally Ordered Broadcast in the Face of Network Partitions*, pages 51–75. Springer US, Boston, MA, 2000.

[88] J. Kirsch and Y. Amir. Paxos for System Builders: An Overview. In *Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, Yorktown Heights, NY, USA, 2008. ACM.

[89] M. S. Krishnamoorthy and B. Krishnamurthy. Fault Diameter of Interconnection Networks. *Comput. Math. Appl.*, 13(5-6):577–582, April 1987.

[90] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95 – 114, May 1978.

[91] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[92] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[93] L. Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[94] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[95] L. Lamport. Generalized Consensus and Paxos. Technical report, March 2005.

[96] L. Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, October 2006.

[97] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, April 1982.

[98] L. Lamport and M. Massa. Cheap Paxos. In *Proc. 2004 International Conference on Dependable Systems and Networks*, DSN '04, Florence, Italy, 2004. IEEE Computer Society.

[99] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[100] B. W. Lampson. How to Build a Highly Available System Using Consensus. In *Proc. 10th International Workshop on Distributed Algorithms*, WDAG '96, Berlin, Heidelberg, 1996. Springer-Verlag.

[101] B. W. Lampson. The ABCD's of Paxos. In *Proc. Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, Newport, RI, USA, 2001. ACM.

[102] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, Oct 1985.

[103] J. M. Lewis, P. Trinh, and D. Kirsh. A Corpus Analysis of Strategy Video Game Play in Starcraft: Brood War. In *Proc. 33rd Annual Conference of the Cognitive Science Society*, Austin, TX, USA, 2011. Cognitive Science Society.

[104] C. L. Li, T. S. McCormick, and D. Simich-Levi. The Complexity of Finding Two Disjoint Paths with Min-max Objective Function. *Discrete Appl. Math.*, 26(1):105–115, January 1990.

[105] B. Liskov and J. Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[106] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART Way to Migrate Replicated Stateful Services. *SIGOPS Oper. Syst. Rev.*, 40(4):103–115, April 2006.

[107] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):271–285, July 1990.

[108] D. Malkhi, M. K. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in byzantine environments. In *Proc. 20th IEEE Symposium on Reliable Distributed Systems*, SRDS '01, New Orleans, LA, USA, Oct 2001.

[109] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, San Diego, CA, USA, 2008. USENIX Association.

[110] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN, Chicago, IL, USA, June 2010.

[111] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: High-Throughput Atomic Broadcast. *The Computer Journal*, 60(6):866–882, June 2017.

[112] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *Proc. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, Atlanta, GA, USA, 2014. IEEE Computer Society.

[113] D. Mazieres. Paxos Made Practical, 2007. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`.

[114] Mesosphere. DC/OS, 2017. `https://docs.mesosphere.com/overview/`.

[115] F. J. Meyer and D. K. Pradhan. Flip-Trees: Fault-Tolerant Graphs with Wide Containers. *IEEE Trans. Comput.*, 37(4):472–478, April 1988.

[116] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT Protocols. In *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, Vienna, Austria, 2016. ACM.

[117] C. Mitchell, Y. Geng, and J. Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proc. 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, San Jose, CA, USA, 2013.

[118] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, New Orleans, LA, USA, 2010.

[119] I. Moraru, D. G. Andersen, and M. Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, Farminton, PA, USA, 2013. ACM.

[120] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous Fault-tolerant Total Ordering Algorithms. *SIAM J. Comput.*, 22(4):727–750, August 1993.

[121] A. Mostefaoui and M. Raynal. Low Cost Consensus-based Atomic Broadcast. In *Proc. 2000 Pacific Rim International Symposium on Dependable Computing*, PRDC '00, Los Angeles, CA, USA, 2000. IEEE Computer Society.

[122] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(01):95–107, 2001.

[123] MPI Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015.

[124] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `http://bitcoin.org/bitcoin.pdf`.

[125] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *Proc. 8th International Conference on Distributed*, June 1988.

[126] T. P. Ng. Ordered broadcasts for large applications. In *Proc. 10th Symposium on Reliable Distributed Systems*, SRDS '91, Pisa, Italy, Sep 1991.

[127] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs. In *Proc. 6th Conference on Computer Systems*, EuroSys '11, Salzburg, Austria, 2011. ACM.

[128] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, Lombard, IL, USA, 2013. USENIX Association.

[129] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. 7th Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, Toronto, ON, Canada, 1988. ACM.

[130] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proc. 2014 USENIX Annual Technical Conference*, ATC'14, Philadelphia, PA, USA, June 2014. USENIX Association.

[131] F. Pedone and A. Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, January 2003.

[132] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *Proc. 4th European Dependable Computing Conference on Dependable Computing*, EDCC-4, Toulouse, France, 2002. Springer-Verlag.

[133] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 1989.

[134] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. 5th USENIX Conference on File and Storage Technologies*, FAST '07, San Jose, CA, USA, 2007.

[135] J. S. Plank, A. L. Buchsbaum, and B. T. Vander Zanden. Minimum Density RAID-6 Codes. *Trans. Storage*, 6(4):16:1–16:22, June 2011.

[136] M. Poke and C. W. Glass. A Dual Digraph Approach for Leaderless Atomic Broadcast (Extended Version). *CoRR*, abs/1708.08309, 2017.

[137] M. Poke and C. W. Glass. Formal Specification and Safety Proof of a Leaderless Concurrent Atomic Broadcast Algorithm. *CoRR*, abs/1708.04863, 2017.

[138] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, Portland, OR, USA, 2015. ACM.

[139] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks (Extended Version). `http://htor.inf.ethz.ch/sec/dare-tr.pdf`, 2015.

[140] M. Poke, T. Hoefler, and C. W. Glass. AllConcur: Leaderless Concurrent Atomic Broadcast (Extended Version). *CoRR*, abs/1608.05866, 2016.

[141] M. Poke, T. Hoefler, and C. W. Glass. AllConcur: Leaderless Concurrent Atomic Broadcast. In *Proc. 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, Washington, DC, USA, 2017. ACM.

[142] L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. In *Proc. 20th International Conference on Distributed Computing Systems*, ICDCS '00, Taipei, Taiwan, 2000. IEEE Computer Society.

[143] L. Rodrigues and P. Verissimo. xAMp: a multi-primitive group communications service. In *Proc. 11th Symposium on Reliable Distributed Systems*, SRDS '92, Houston, TX, USA, 1992.

[144] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[145] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. 5th USENIX Conference on File and Storage Technologies*, FAST '07, San Jose, CA, USA, 2007.

[146] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proc. Workshop on Managed Many-Core Systems*, Boston, MA, USA, June 2008. Association for Computing Machinery, Inc.

[147] D. Sciascia. Libpaxos3, 2013. `http://libpaxos.sourceforge.net/paxos_projects.php`.

[148] Securities and Exchange Commission. Release No. 34-70129; File No. SR-NASDAQ-2013-099, 2013. `https://www.sec.gov/rules/sro/nasdaq/2013/34-70129.pdf`.

[149] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. v. Steen. Replication for Web Hosting Systems. *ACM Comput. Surv.*, 36(3):291–334, September 2004.

[150] T. Soneoka, M. Imase, and Y. Manabe. Design of a d-connected digraph with a minimum number of edges and a quasiminimal diameter II. *Discrete Appl. Math.*, 64(3):267–279, February 1996.

[151] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[152] P. Unterbrunner, G. Alonso, and D. Kossmann. High Availability, Elasticity, and Strong Consistency for Massively Parallel Scans over Relational Data. *The VLDB Journal*, 23(4):627–652, August 2014.

[153] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[154] 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*, 12(4):472–484, July 2015.

[155] M. van Steen and G. Pierre. *Replicating for Performance: Case Studies*, pages 73–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[156] A. Varga and R. Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proc. 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, Marseille, France, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[157] W. Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[158] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. *SIGOPS Oper. Syst. Rev.*, 29(5):40–53, December 1995.

[159] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, April 1992.

[160] M. Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Proc. IFIP WG 11.4 Workshop on Open Research Problems in Network Security*, iNetSec'15, Zurich, Switzerland, 2016. Springer International Publishing.

[161] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, Santa Clara, CA, USA, 2017. ACM.