Brief Announcement: Revisiting Consensus Protocols through Wait-free Parallelization

Suyash Gupta
Exploratory Systems Lab, Department of Computer Science, University of California, Davis, USA
sugupta@ucdavis.edu

Jelle Hellings
Exploratory Systems Lab, Department of Computer Science, University of California, Davis, USA
jhellings@ucdavis.edu

Mohammad Sadoghi
Exploratory Systems Lab, Department of Computer Science, University of California, Davis, USA
msadoghi@ucdavis.edu

Abstract

In this brief announcement, we propose a *protocol-agnostic* approach to improve the design of primary-backup consensus protocols. At the core of our approach is a novel wait-free design of running several instances of the underlying consensus protocol *in parallel*. To yield a high-performance parallelized design, we present coordination-free techniques to order operations across parallel instances, deal with instance failures, and assign clients to specific instances. Consequently, the design we present is able to reduce the load on individual instances and primaries, while also reducing the adverse effects of any malicious replicas. Our design is fine-tuned such that the instances coordinated by non-faulty replicas are *wait-free*: they can continuously make consensus decisions, independent of the behavior of any other instances.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, primary-backup, high-performance, wait-free parallelization

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.43

Category Brief Announcement

Related Version The technical report on which this brief announcement is based is available at https://arxiv.org/abs/1908.01458.

1 Introduction

At the core of any blockchain application is a *consensus protocol* that facilitates replicating data across a group of servers (or replicas), some of which can fail or act malicious [5]. Commonly, these protocols use the *primary-backup model* pioneered in the Practical Byzantine Fault Tolerant consensus protocol [2]. In these BFT-style protocols, a single replica is designated as *the primary* and is responsible for coordinating the consensus decisions, while all the other replicas perform the *backup role*. The primary-backup model simplifies the development of consensus protocols substantially: when a primary is non-malicious, then even the simplest broadcast replication protocols suffice. However, the only complication in these consensus protocols is in the way they deal with malicious primaries: malicious behavior must be either detected (after which the primary can be replaced) or prevented altogether. This simplicity of the primary-backup model negatively affects its performance in three ways [3, 1]:

1. *Primary load.* The primary not only has to perform the primary tasks, but also the backup role (as it is itself a replica). Consequently, the primary receives a higher load than other replicas, and this load at the primary can become a *bottleneck in the overall system*

throughput. This is especially the case in fine-tuned high-performance consensus protocols that employ complex cryptographic primitive, for example, to reduce communication overheads or to improve resilience.

2. *Primary replacement.* A primary-backup consensus protocol works only when the primary behaves in accordance with the protocol. If the primary acts malicious or is faulty, then it will be replaced. However, detection of such behaviors requires setting timers. Further, replacing a faulty primary usually takes a while. During this time the system is unable to handle requests, which negatively affects its overall *throughput.*
3. *Malicious behavior.* Primary-backup consensus protocols are only capable of detecting catastrophic failures that prevent new consensus decisions altogether, but they fail to detect or deal with primaries that *affect the performance of the system in other ways*, for example, a malicious primary could reduce or throttle the throughput of the system.

To the best of our knowledge, no approach is yet able to address all these limitations of primary-backup consensus protocols. In this brief announcement, we address these limitations in a *protocol-agnostic* manner by exploiting parallelization[1]. In our *paradigm*, we run several instances of the underlying consensus protocol *in parallel* and balance the system load among these parallel instances. This parallelism helps to reduce the *load per primary* and mitigates the negative impacts of a single primary on the *throughput of the system.*

2 Parallelizing consensus

Consider a system with $\mathbf{n}$ replicas, of which $\mathbf{f}$ are faulty. At the core of our parallelization paradigm is the coordination of $\mathbf{m}$, $\mathbf{f} < \mathbf{m} \leq \mathbf{n}$, instances of an off-the-shelf primary-backup consensus protocol running *in parallel.* This implies that a single round of our paradigm coordinates *multiple parallel consensus rounds*, each of which is initiated and managed by a *distinct* primary $\mathcal{P}_i$ for the instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$. Each consensus decision succeeds whenever $\mathcal{P}_i$ is non-faulty. This approach to parallelization raises several important challenges:

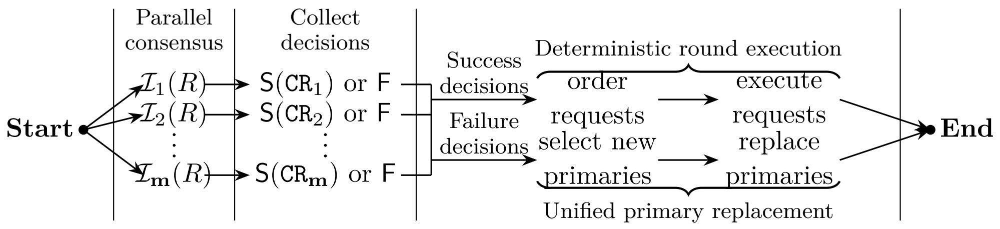
1. For optimal throughput, we need to ensure that each instance is making a *distinct consensus decision*, that is, each instance is processing a distinct client request.
2. Every non-faulty replica should execute all the accepted client requests in *the same order.*
3. When several instances fail in a round and want to *transfer control to new primaries*, then all the non-faulty replicas need to do so in the same manner.

In our design, we address each of these challenges. Figure 1 sketches a high-level overview of a *parallelized consensus round* at replica R. Our paradigm also identifies various ways in which malicious replicas can prevent it from efficently operation. Next, we highlight the design decisions taken to addresses these possible attacks.

Deterministic round execution. The correctness of the underlying consensus protocol, used by instances $\mathcal{I}_1, \ldots, \mathcal{I}_{\mathbf{m}}$, can guarantee that each non-faulty replica derives the same set of client requests in round ρ. Hence, our paradigm behaves correctly whenever all non-faulty replicas can determine the same order of execution of these client requests. To avoid that malicious replicas can influence, predict, or reliably exploit the order of execution to their advantage, we permute the order of execution based on the set of client requests accepted in round ρ (some of which are proposed by non-faulty replicas).

Dealing with primary failure. In primary-backup consensus protocols, the primary can fail or act malicious. Our paradigm supports two ways to deal with such failures. The

[1] A complementary approach to increase parallelism is the partial replication through sharding [4]



Parallel consensus
Collect decisions
Start
$\mathcal{I}_1(R)$
$\mathcal{I}_2(R)$
$\mathcal{I}_\mathbf{m}(R)$
$\mathsf{S}(\mathtt{CR}_1)$ or F
$\mathsf{S}(\mathtt{CR}_2)$ or F
$\mathsf{S}(\mathtt{CR}_\mathbf{m})$ or F
Success decisions
Failure decisions
Deterministic round execution
order requests
execute requests
select new primaries
replace primaries
Unified primary replacement
End

Figure 1 A high-level overview of a replica R. The replica coordinates a single consensus round among $\mathbf{m}$ instances of some *consensus protocol*. Each instance yields a consensus decision. The success decisions $\mathsf{S}(\cdot)$ yield a set of client requests, which are executed in a deterministic order. The failure decisions F are collected and can be used to replace primaries in a unified manner.

straightforward approach is to suspend instances with failed primaries and to try recovering these failed primaries after some delay. To avoid recovering too often, this delay is doubled after each failure. We also support replacement of failed primaries by other replicas. To do so, we provide a *unified primary replacement* protocol that only requires coordination among the instances with failed primaries.

Consistent handling of client requests. To ensure that subsequent client requests are always processed in order, we assign clients to instances in a round-robin manner. We do this by requiring each instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$, to only deal with client requests of a client c, if $i = c \bmod \mathbf{m}$. We notice that a client c can be assigned to an instance with a faulty primary that might ignore its request. To deal with this situation, we allow clients to periodically issue instance-change requests, to reassign them to other instances. To assure balanced load among the instances, a non-faulty instance only accepts an instance-change request, if it has not been yet assigned $\lceil \mathbf{c}/(\mathbf{n}-\mathbf{f}) \rceil$ clients, where $\mathbf{c}$ refers to the total number of clients.

Wait-free parallelization. To ensure the correctness of our parallelization paradigm, we do not require any non-faulty instance to wait for other instances. In our paradigm, the *execution of client requests* in round ρ has no influence on the consensus decisions of future rounds. Second, the instances arriving at successful consensus decisions do not require any coordination. The only required coordination is between instances with failed primaries. Hence, instances that arrived at successful consensus decisions in the current round are free to make consensus decisions for the future rounds, while the *execution* of the client requests of previous rounds occurs in the background. Furthermore, we have coordination-free ways to detect and sanction malicious behavior of primaries (e.g., throttling performance). Combined, these approaches guarantee that instances with non-faulty primaries are *wait-free*: they are always able to operate at maximum throughput and will always see their client requests executed within bounded time, this independent of any malicious behavior in other instances.

References

1 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant byzantine fault tolerance. In *ICDCS*, 2013.
2 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.
3 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.
4 Jelle Hellings and Mohammad Sadoghi. Brief announcement: the fault-tolerant cluster-sending problem. In *DISC*, 2019.
5 Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *ICDE*, 2019.