Resilient Cloud-based Replication with Low Latency

Michael Eischer and Tobias Distler
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Erlangen, Germany

Abstract

Existing approaches to tolerate Byzantine faults in geo-replicated environments require systems to execute complex agreement protocols over wide-area links and consequently are often associated with high response times. In this paper we address this problem with Spider, a resilient replication architecture for geo-distributed systems that leverages the availability characteristics of today's public-cloud infrastructures to minimize complexity and reduce latency. Spider models a system as a collection of loosely coupled replica groups whose members are hosted in different cloud-provided fault domains (i.e., availability zones) of the same geographic region. This structural organization makes it possible to achieve low response times by placing replica groups in close proximity to clients while still enabling the replicas of a group to interact over short-distance links. To handle the inter-group communication necessary for strong consistency Spider uses a reliable group-to-group message channel with first-in-first-out semantics and built-in flow control that significantly simplifies system design.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Keywords: Byzantine fault tolerance, geo-replication

ACM Reference Format:
Michael Eischer and Tobias Distler. 2020. Resilient Cloud-based Replication with Low Latency. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425689

1 Introduction

Byzantine fault-tolerant (BFT) protocols enable a system to withstand arbitrary faults and consequently have been used to increase the resilience of a wide spectrum of critical applications such as key-value stores [22, 33, 42, 43], SCADA systems [9, 10, 41], firewalls [15, 26], coordination services [11, 18, 20, 24, 30], and permissioned blockchains [28, 47]. To provide their high degree of fault tolerance, BFT protocols replicate the state of an application across a set of servers and rely on a leader-based consensus algorithm to keep these replicas consistent. This task requires several subprotocols (e.g., for leader election, checkpointing, state transfer) and multiple phases of message exchange between replicas [17].

Unfortunately, this complexity makes it inherently difficult to achieve low latency in use cases in which the clients of an application are scattered across various geographic locations. For example, placing replicas in close proximity to each other may reduce the latency of strongly consistent requests whose execution must be coordinated by the consensus protocol between replicas. However, with replicas being located farther apart from clients this strategy also increases the response times of requests such as weakly consistent reads that do not need to be agreed on and only involve direct interaction between clients and replicas. In contrast, co-locating replicas with clients has the inverse effect of speeding up client–replica communication but adding a significant performance overhead to the agreement protocol.

Existing approaches for BFT wide-area replication aim at minimizing this overhead by (1) applying weighted-voting schemes to reduce the quorum sizes needed to complete consensus [12, 46], (2) rotating the leader role among replicas to shorten the path necessary to insert a request into the agreement protocol [36, 50, 51], or (3) relying on a two-level system design that deploys an entire BFT replica cluster at each client site in order to be able to use crash-tolerant replication between sites [4, 6]. In all these cases, BFT systems still need to run complex consensus-based replication protocols over wide-area links which not only results in response-time overhead but also makes it difficult to dynamically introduce new replica sites, for example, to serve clients at new locations.

In this paper we address these problems with Spider, a cloud-based BFT system architecture for geo-replicated services that models a system as a collection of loosely coupled replica groups that are deployed in different regions. Separating agreement from execution [52], one of the groups ("*agreement group*") establishes an order on all requests with strong consistency demands while all other groups ("*execution groups*") are responsible for communicating with clients and processing requests. In contrast to existing approaches, Spider does not require complex wide-area protocols but instead handles tasks such as consensus, leader election, and

Middleware '20, December 7–11, 2020, Delft, Netherlands

checkpointing within a group and over short-distance links. To make this possible while still offering resilience against replica failures, Spider leverages the design of today's cloud infrastructures [2, 27, 38] and places the replicas of a group in different availability zones of the same region; availability zones are hosted by data centers at distinct sites and specifically engineered to represent different fault domains.

In particular we make four contributions in this paper: (1) We present the Spider architecture and discuss how it achieves low latency for weakly consistent reads by placing execution groups close to clients, while at the same time minimizing agreement response times for strongly consistent reads and writes. (2) We show how to design Spider in a modular way so that execution groups do not depend on internals of the agreement group (e.g., a specific consensus protocol). As an additional benefit, the modularity also makes it straightforward to add/remove execution groups at runtime. (3) We introduce a wide-area BFT flow-control mechanism that exploits the special characteristics of Spider to minimize complexity. Our approach is based on a simple message-channel abstraction that handles the inter-regional communication between two replica groups and prevents one group from overwhelming the other. (4) We evaluate Spider in comparison to the state of the art in BFT wide-area replication.

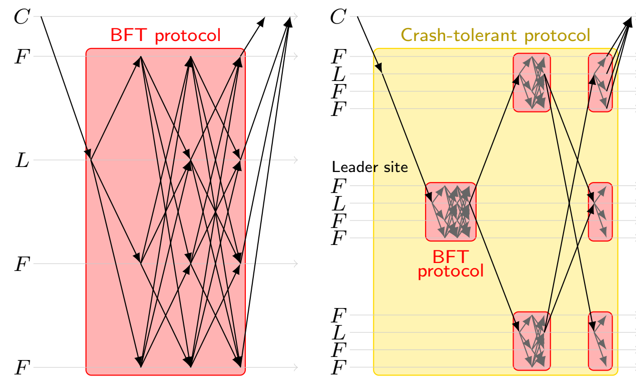
2 Background and Problem Statement

In this section, we present background on existing approaches and common requirements of BFT wide-area replication.

2.1 System Model

Our work focuses on stateful applications with strong reliability requirements whose clients are scattered across different geographic locations. To access the application a client submits a request to the server side. We assume that both clients and servers can be subject to Byzantine faults. As a consequence, nodes (i.e., clients and servers) do not trust each other and do not make irreversible decisions based on the input provided by another node alone. For example, to tolerate up to f faulty servers, a client only accepts a result after it has obtained at least $f+1$ matching replies from different servers.

Besides service availability and correctness in the presence of failures, low latency is a primary concern in our target systems. Achieving this goal while keeping the states of servers consistent is inherently difficult in use cases in which clients are geographically dispersed. The problem is further complicated by the fact that we assume that the locations from which clients access the application may change over time, typically as a result of the global day/night cycle. To continuously provide low latency under such conditions, a system must offer some kind of reconfiguration mechanism enabling an adaptation to varying workloads. One possibility to achieve this, for example, is to dynamically include additional servers that are located closer to newly started clients.



(a) PBFT [17] **(b)** Steward [6]

Figure 1. System architectures for BFT geo-replication connecting a client (C) with leader (L) and follower (F) replicas.

2.2 Existing Approaches

In the following, we elaborate on the problems associated with Byzantine fault tolerance in geo-distributed systems and discuss existing approaches to solve them.

BFT in Wide-Area Environments. The straightforward approach to offer resilience against arbitrary failures is to rely on a BFT replication protocol, for example PBFT [17]. As illustrated in Figure 1a, PBFT requires at least $3f+1$ replicas to tolerate f failures. To keep the application state consistent across replicas, PBFT ensures that replicas run an agreement protocol to decide in which order to process client requests. For this purpose, PBFT elects one of the replicas as leader (marked L in Figure 1a) while all other replicas assume the roles of followers (F). Having received a new request, the leader is responsible for initiating the agreement process, which then involves multiple message exchanges between replicas. To deal with scenarios where a faulty leader does not behave according to specification, for example by ignoring a request, PBFT provides a mechanism that enables followers to depose the leader and appoint a new one. Once the agreement process is complete, all non-faulty replicas execute the request and send the result to the client, thereby enabling the client to validate the result by comparison.

Using BFT protocols such as PBFT to build resilient systems is effective but has several disadvantages in the context of geo-replication: (1) With replicas being distributed across different geographic sites, the entire BFT protocol needs to be executed over wide-area links, which often results in high response times. Note that this is not only true with regard to the task of agreeing on requests during normal operation, but for example also for electing a new leader as part of fault handling. (2) Due to the fact that all requests must flow through the leader, the geographic location of the leader, and in particular its position relative to the majority of followers, usually has a significant influence on latency [23, 46]. Consequently, a leader switch may decisively change a system's performance characteristics, requiring clients to deal with the associated latency volatility. (3) Consisting of only $3f+1$ replicas, for traditional BFT systems it is inherently difficult to select suitable replica locations in cases where a large and varying number of clients are scattered across the globe. Ideally, replicas would be placed both in close distance to each other (to speed up agreement) as well as in close distance to clients (to minimize the transmission time of requests and results). For systems with just a few replicas but many clients meeting this requirement is essentially impossible.

Weighted Voting. By assigning different weights on the votes replica have within the consensus protocol [12, 46] it is feasible to introduce additional replicas while keeping response times low or even reducing them in a geo-replicated setting. Unfortunately, this comes at the cost of an increased number of messages exchanged between replicas, which can be prohibitively expensive in public-cloud settings as providers typically charge extra for wide-area traffic.

Leader Rotation. Different authors have proposed to improve performance by rotating the leader role among replicas, following the idea of enabling each client to submit requests to its nearest replica [36, 50, 51]. Results from an extensive experimental evaluation by Sousa et al. [46], however, showed that in practice this approach does not provide significant benefits compared with appointing a fixed leader at a well-connected site. Besides, leader rotation still requires the execution of a complex protocol over wide-area links.

Hierarchical System Architecture. To increase the scalability of BFT systems in wide-area settings, Amir et al. presented a hierarchical architecture as part of their Steward system [6]. As shown in Figure 1b, instead of hosting a single replica, each site in Steward comprises a cluster of replicas that run a site-local BFT agreement protocol. A key benefit of this approach is the fact that, although individual replicas still may be subject to Byzantine faults, an entire cluster can be assumed to only fail by crashing. This property at the local level enables Steward to rely on a crash-tolerant agreement protocol at the global level (i.e., between sites), which compared with traditional BFT systems requires fewer phases and fewer message transmissions over wide-area links.

The efficiency enhancements made possible by its architecture enable Steward to improve performance, however, they come at the cost of an increased overall complexity that stems from the need to maintain replication protocols at two levels: within each site as well as between sites. Designing and implementing such protocols in isolation already is a non-trivial task, additionally guaranteeing a correct interplay between them is even more challenging. To ensure liveness Steward, for example, requires timeouts at different levels to be carefully coordinated [6]. Amir et al. addressed these problems in a subsequent work [4], which in this paper we refer to as CFT-WAR. In contrast to Steward, in CFT-WAR each step of the wide-area protocol (e.g., Paxos [31]) is handled by a full-fledged multi-phase consensus protocol at each site (e.g., PBFT). As a main advantage, this approach disentangles the protocols used for wide-area and site-internal replication. On the downside, it introduces additional overhead that in general prevents CFT-WAR from achieving response times as low as Steward's when providing the same degree of fault tolerance [4]. Furthermore, due to performing agreement at two levels CFT-WAR still needs to run multiple subprotocols for tasks such as leader election, one at each level. A set of additional subprotocols would be required to support the dynamic addition/removal of individual replicas or entire sites in a hierarchical system architecture, thereby further increasing complexity. To our knowledge, the ability to adjust to varying workload conditions was not a design goal of Steward and CFT-WAR, which is why the systems do not offer mechanisms for changing their composition at runtime.

2.3 Problem Statement

Our analysis in Section 2.2 shows that applying existing approaches to provide BFT in a cloud-based geo-replicated environment is possible, for example with regard to safety, but cumbersome due to the associated high complexity and the lack of effective means to react to changing workloads. This observation led us to ask whether these problems can be circumvented by a BFT system architecture that is specifically tailored to the characteristics of today's cloud infrastructures. In particular, we aim for a resilient system architecture that has three properties: efficiency, modularity, and adaptability.

Efficiency. To minimize response times during both normal-case operation as well as fault handling, a system architecture in the ideal case does not require the execution of complex protocols over wide-area links. Instead, tasks involving multiple phases of message exchange between replicas, such as the agreement on requests, should be handled by replicas that are located in comparably close distance to each other.

Modularity. Supporting a variety of cloud use cases with different requirements is difficult if the protocols responsible for the agreement and execution of requests are hard-wired into the BFT system architecture. To address this issue, we join other authors [4] in aiming for an architecture that, for example, can be integrated with different consensus protocols depending on the specific demands of an application.

Adaptability. One major strength of public clouds is to quickly provide resources on demand and at various geographic locations all over the globe. A BFT system architecture should be able to leverage this feature for hosting replicas in the proximity of clients to reduce the latency with which clients access the replicated service. Specifically, if new clients are started at other sites, there should be a lightweight mechanism for dynamically adding new replicas. The same applies to means for removing replicas that are no longer of benefit as the clients in their vicinity have been shut down.

3 Spider

This section presents the cloud-based BFT system architecture Spider. In particular, we focus on how the architecture achieves low latency by performing consensus only over short-distance links, how Spider achieves modularity by relying on a novel message-channel abstraction, and how it can be dynamically reconfigured to adapt to workload changes.

3.1 Architecture

Targeting use cases in wide-area environments, Spider's system architecture is distributed across multiple geographic sites. For this purpose, Spider leverages the common organizational structure of state-of-the-art cloud infrastructures such as Amazon EC2 [2], Microsoft Azure [38], or Google Compute Engine [27] by grouping sites into *regions*, as shown in Figure 2. The sites within a region typically are several tens of kilometers apart from each other and represent separate fault domains, commonly referred to as *availability zones*. In addition to constructing the data centers at distinct geographic locations, cloud providers also ensure that data centers in different availability zones are equipped with dedicated power supply systems and network links to minimize the probability of dependent failures. For the Spider system architecture, availability zones play an important role as they allow us to place replicas in separate fault domains and still enable them to interact over short-distance links with comparably low latency.

Replica Groups. Relying on this setting, Spider is composed of multiple loosely coupled replica groups, each being distributed across different availability zones of a specific region. One of the replica groups in the system, the *agreement group*, is responsible for establishing a global total order on incoming requests. The size of this group depends on the protocol it uses for consensus. Running PBFT [17], for example, the agreement group consists of $3f_a + 1$ replicas and is able to tolerate f_a Byzantine faults. All other replica groups in the system, the *execution groups*, host the application logic, process the ordered requests, and handle the communication with clients. Each of these groups comprises $2f_e + 1$ replicas and tolerates at most f_e Byzantine faults. The level of fault tolerance provided by the agreement group and the executions groups may be selected independently. Supporting multiple execution groups enables Spider to scale throughput by adding/removing groups and to minimize latency by placing groups in the vicinity of clients.

Execution-Replica Registry. Spider contains an execution-replica registry to provide clients with information on the locations and addresses of active replicas. The registry is a BFT service that is hosted and maintained by the agreement group. Its contents are updated by agreement replicas whenever the composition of the system changes (see Section 3.6).

Efficient BFT Replication. In contrast to existing approaches (see Section 2.2), Spider does not run a full-fledged and

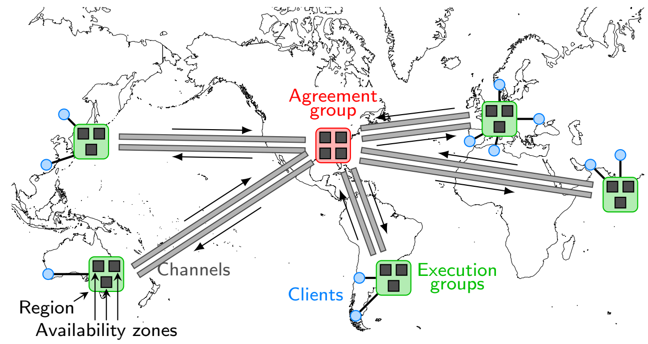


Figure 2. Spider system architecture

complex replication protocol over long-distance links. Instead, all non-trivial tasks (e.g., reaching consensus on requests) are carried out within a replica group using low-latency intra-region connections. Following this design principle, Spider handles requests by forwarding them along a chain of stages represented by different replica groups. Specifically, clients submit their requests to their nearest execution group, which in turn forwards the request to the agreement group for ordering. Once this step is complete, the agreement group instructs all execution groups to process the ordered request. This ensures that execution-group states remain consistent without requiring the execution groups to reach consensus themselves. Having processed the request, the replicas of the execution group the client is connected to return the result. As each execution group comprises $2f_e + 1$ replicas, clients are able to verify the correctness of a result solely based on the replies they receive from their local execution group.

With all communication-intensive steps being performed over intra-region links, inter-region links in Spider are only responsible for forwarding the outputs of one stage to the replica group(s) constituting the next stage. In particular, this approach has the following benefits: (1) It greatly simplifies the interaction of replicas over long-distance connections. (2) It enables a modular design that allows different deployments to rely on different agreement protocols without the need to modify the implementation of execution replicas. (3) As we show in Section 3.2, it allows Spider to use the same abstraction, a reliable message channel, for all inter-region links, thereby facilitating system implementation.

Practical Considerations. As of this writing, all major public clouds offer several regions with at least three availability zones (Amazon EC2: 20, Microsoft Azure: 10, Google Compute Engine: 24) and therefore support the world-wide deployment of Spider execution groups which tolerate one faulty replica. In addition, Amazon (Virginia, Oregon, Tokyo) and Google (Iowa) also already operate regions with four or more availability zones, which consequently are candidates for hosting Spider's agreement group. With public cloud infrastructures still being expanded, new regions and availability zones are added every year, increasing the deployment options for Spider. Besides, to further improve the resilience of Spider, agreement and execution replicas may be distributed across different clouds, thereby reducing the dependence on a single provider [1, 13]. As there are several regions hosting data centers and availability zones of multiple providers (e.g., Europe, North America, South America, India, Asia, and Australia), this approach also makes it possible to deploy larger agreement and execution groups that tolerate $f_a > 1$ and $f_e > 1$ replica failures, respectively.

Representing distinct fault and upgrade domains, availability zones are designed to enable uninterrupted execution of services that are replicated within the same region. Despite the efforts undertaken by providers, in the past there have been rare incidents where problems in one availability zone caused temporary availability issues in other zones belonging to the same region [3]. In Spider, if more than f_a agreement replicas are unresponsive, the agreement group temporarily cannot order new requests until the replicas become available again. However, as we detail in Section 3.3, in such cases Spider is still able to process weakly consistent read requests as these operations are handled within a client's local execution group. On the other hand, if more than f_e replicas of the same execution group become unavailable, affected clients can temporarily switch to a different execution group and continue to use the service.

3.2 Inter-Regional Message Channels

To support a modular design, we use an abstraction to handle all interaction between replica groups in Spider: the *inter-regional message channel (IRMC)*. Specifically, IRMCs are responsible for forwarding messages from a group of sender replicas in one region to a group of receiver replicas in another region. Conceptually, IRMCs can be viewed as an extension of BLinks [4], however, unlike BLinks, IRMCs (1) do not require messages to be totally ordered at the channel level and (2) comprise built-in flow control. To forward information, an IRMC internally can be divided into multiple subchannels providing first-in-first-out semantics. Each subchannel has a configurable maximum capacity (i.e., an upper bound on the number of messages that can be concurrently in transmission) and relies on a window-based flow-control mechanism to prevent senders from overwhelming receivers. Below, we discuss the specifics of IRMCs at a conceptual level. For possible implementations please refer to Section 4.

Overview. Figure 3 presents an example IRMC that comprises two subchannels and connects four senders to three receivers. Subchannels of the same IRMC are independent of each other and can be regarded as distributed queues with limited capacity that distinguish messages based on unique position indices. Both senders and receivers run dedicated endpoints which together form the IRMC and enable the replicas to access it. When a replica sends a message,

```
Sender Endpoint Interface

void send(SUBCHANNEL sc, POSITION p, MESSAGE m);
void move_window(SUBCHANNEL sc, POSITION p);

Receiver Endpoint Interface

MESSAGE receive(SUBCHANNEL sc, POSITION p);
void move_window(SUBCHANNEL sc, POSITION p);
```

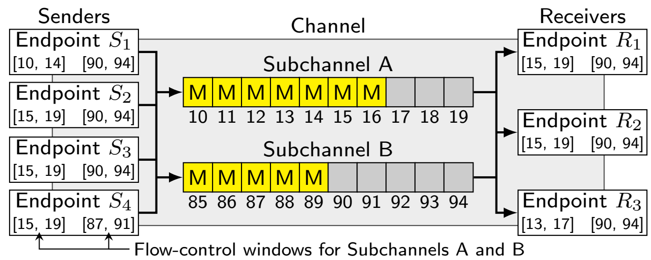


Figure 3. Conceptual view of an example IRMC with two independent subchannels that both have a maximum capacity of ten messages (M). Senders (S_*) and receivers (R_*) access the subchannels via their local endpoints; each endpoint manages its own subchannel-specific flow-control windows.

it provides its local endpoint with the information which subchannel and position to use for the message (`send()`). Similarly, to receive a message a replica queries its local endpoint for the message corresponding to a specific subchannel and position (`receive()`). In addition, IRMC endpoints offer a method to shift the flow-control window of a subchannel (`move_window()`), as further discussed below.

Send Semantics. IRMCs are not designed to exchange arbitrary messages between replicas but instead provide specific send semantics enabling Spider to safely forward the decision of a replica group to another. In particular, tolerating at most f_s senders with Byzantine faults, the IRMC only forwards a message after at least $f_s + 1$ different senders transmitted a message with identical content using the same subchannel and position. Consequently, in order for a message to pass the channel at least one correct sender must have vouched for the validity of the message's content and requested its transmission. In contrast, messages solely submitted by the up to f_s faulty senders have no possibility of getting through and being delivered to receivers.

Authentication. IRMCs protect all channel-internal communication with digital signatures to enable the recipient of a message to verify the integrity and the origin of the message. If an endpoint is unable to validate the authenticity of a received message, it immediately discards the message.

Flow Control. With the capacities of subchannels being limited, IRMC endpoints apply a flow-control mechanism to coordinate senders and receivers. For this purpose, for each subchannel an endpoint manages a separate window that restricts which messages a sender/receiver is able to transmit/obtain at a given time. If a subchannel's window at a sender endpoint is full, the sender cannot insert additional messages into this subchannel until the endpoint moves the window forward. In the normal case, this action is triggered by receivers calling `move_window()` and requesting the start of the window to be shifted to a higher position. Whenever a sender endpoint learns that the window position has changed at one of the receiver endpoints, the sender endpoint sets its own window start to the $f_r + 1$ highest position requested by any receiver where f_r denotes the number of receivers with Byzantine faults to tolerate. This ensures that correct sender endpoints only move their windows, and thus discard messages at lower positions, after receiving the information that at least one correct receiver has permitted such a step.

Besides receiver-driven window shifts, our channels also allow senders to request an increase of the starting position of a subchannel's window. If senders opt to do so, it may become impossible for a receiver endpoint to provide the message at the position the endpoint's local replica requested. The same scenario can occur if a receiver endpoint is slow or falls behind (e.g., due to a network problem) while $f_r + 1$ other receivers have already requested the window to be moved forward. In such cases, the affected receiver endpoint aborts the `receive()` call with an exception and thereby enables its local replica to handle the situation. As discussed in Section 3.4, replicas react to such an exception by obtaining the missed information from other replicas.

Use in Spider. IRMCs are an essential building block of Spider's modular architecture as they enable us to design a geo-replicated BFT system as a composition of loosely coupled replica groups that interact using the same channel abstraction. In particular, Spider relies on two different IRMC instances to perform all inter-group communication over long-distance links: the *request channel* and the *commit channel*.

The request channel allows an execution group to forward newly received requests to the agreement group; that is, this channel is an IRMC that connects $2f_e + 1$ senders (i.e., execution replicas) to $3f_a + 1$ receivers (i.e., agreement replicas). To transmit the requests, the request channel comprises multiple subchannels, one for each client. In contrast, the commit channel only consists of a single subchannel and is used by the agreement group to inform an execution group about the totally ordered sequence of agreed requests. The commit channel consequently is responsible for forwarding the decisions of $3f_a + 1$ senders to $2f_e + 1$ receivers. In summary, the agreement group maintains a pair of IRMCs (i.e., one request channel and one commit channel) to each execution group.

3.3 Request Handling

Spider differentiates between requests that potentially modify application state ("writes") and those that do not ("reads"). This distinction enables the system to handle requests of each category as efficiently as possible. While writes need to be applied to all execution groups to keep their states consistent,

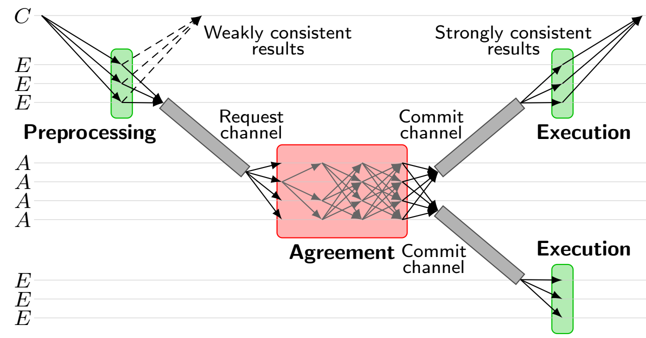


Figure 4. Overview of Spider's replication protocol

it is sufficient for reads to only process them at the execution group a client is connected to. Figure 4 gives an overview of how requests flow through Spider. Below, we provide details on the system's replication protocols for writes and reads. In this context, it is important to note that all messages exchanged between clients and replicas must be authenticated, for example using HMACs [49]. For messages sent through IRMCs, the authentication is handled by the channels.

In the following, we describe Spider's handling of write and read requests . The proof of correctness and liveness is deferred to the extended version of the paper [25].

Writes. Spider's protocol for writes is presented in Figure 5. To perform a write operation w, a client c creates a corresponding message $\langle \textsc{Write}, w, c, t_c \rangle$ using a unique client-local counter value t_c and sends the message to all replicas of an execution group. In general, a client for this purpose may select any execution group in the system, however, in an effort to minimize latency, Spider clients typically choose the group closest to their own site.

When an execution replica receives the client's request, it first checks whether the message is correctly authenticated and whether the client has permission to access the system. If any of these checks fail the replica discards the message. Otherwise, the replica of execution group e wraps the entire request r in a message $\langle \textsc{Request}, r, e \rangle$ and submits the message to the agreement group via its request channel. More precisely, unless the execution replica has already forwarded the request (Lines 5–6) it moves the window of the client's subchannel to position t_c and inserts the write request at that position (L. 7–8). Once at least $f_e + 1$ members of the execution group (i.e., at least one correct execution replica) have validated and forwarded the request, the request channel permits agreement replicas to retrieve the message (L. 30). This allows the agreement group to initiate the consensus process for the message (L. 33), which is then performed entirely within the group's region. Having learned that the request is committed and has been assigned the agreement-sequence number s (L. 35), an agreement replica creates a confirmation $\langle \textsc{Execute}, r, s \rangle$. As write operations need to be processed by all execution groups, the agreement replica sends this message through all commit channels at position s (L. 40).

Once $f_a + 1$ agreement replicas (among them at least one correct replica) have sent an Execute message with the same content and sequence number, a commit channel enables its receivers to obtain the message (L. 10). Having done so, an execution replica processes the included request by applying the corresponding write to its local state (L. 14). Each replica of execution group e also returns a reply $\langle \text{Result}, u_c, t_c \rangle$ with the operation's result u_c to the client that submitted the request with counter value t_c (L. 16). The client accepts a result after it has received $f_e + 1$ replies with matching result and counter value from different execution replicas.

As we detail in Section 3.4, when processing writes replicas in Spider also create periodic checkpoints (L. 17–21 and 41–51) to assist other replicas that might have fallen behind.

Reads. For reads, Spider offers two different operations providing weakly consistent and strongly consistent results, respectively. To perform a weakly consistent read, a client sends a read request to all members of an execution group, which for a valid request immediately responds with a result, as illustrated by the dashed lines in Figure 4. As for writes, a client verifies the result based on $f_e + 1$ matching replies. Weakly consistent reads achieve low latency as they only involve communication between the client and its execution group. Due to these reads being processed without further coordination with writes, in the presence of concurrent writes to the same state parts they may return stale values or fewer than $f_e + 1$ matching results, similar to the optimized reads in existing BFT protocols [17, 46]. Spider clients react to stalled reads by retrying the operation or performing a strongly consistent read, which is guaranteed to produce a stable result.

Strongly consistent reads in Spider for the most part have the same control and data flow as writes, with one important exception. With reads not modifying the application state, it is sufficient to process them at the client's execution group. Consequently, after a read request completed the consensus process, agreement replicas only forward it to the execution group that needs to handle the request. The Executes to all other groups instead contain a placeholder including only the client request counter value for the same sequence number, thereby minimizing network and execution overhead.

3.4 Checkpointing

As discussed in Section 3.2, an IRMC may garbage-collect messages before they have been delivered to all correct receivers. In the normal case in which all receivers advance at similar speed, this property usually does not take effect, resulting in each receiver to obtain every message. To address exceptional cases in which a correct receiver misses messages (e.g. due to a network problem), Spider provides means to bring the affected receiver up to date via a checkpoint. The specific contents of a checkpoint vary depending on the

```
Execution Replica of Execution Group e

1  s_n := 0                                          // Current sequence number
2  t[c] := 0  //Vector with counter value of latest forwarded client request
3  u[c] := ∅                                        // Reply cache ⟨Reply, u_c, t_c⟩

4  on receive(r = ⟨Write, w, c, t_c⟩ from c):
5    if t_c ≤ t[c]: return send result u[c] to c
6    t[c] := t_c                                   // Remember forwarded request
7    request-IRMC.move_window(c, t_c)
8    request-IRMC.send(c, t_c, ⟨Request, r, e⟩)

9  main loop:
10   m := commit-IRMC.receive(0, s_n + 1)
11   if m = ⟨TooOld, s'⟩: fetch checkpoint for s'
12   else:
13     m = ⟨Execute, ⟨Request, ⟨Write, w, c, t_c⟩, e'⟩, s_n + 1⟩
14     u_c := Application execute m
15     s_n := s_n + 1
16     send ⟨Result, u_c, t_c⟩ to c if e' = e and store in u[c]
17     if s_n ≡ 0 mod k_e:
18       create checkpoint for s_n with u and Application

19 on stable checkpoint(SeqNr s, u', Application'):
20   commit-IRMC.move_window(0, s + 1)
21   if s ≥ s_n: apply checkpoint to s_n, u and Application

Agreement Replica

22 s_n := 0                                          // Current sequence number
23 t[c] := 0 // Counter values of latest agreed request; used by consensus
24 t+[c] := 0                       // Counter values for next expected request
25 AG-WIN ≥ k_a                                     // Size of agreement window
26 win := [1,AG-WIN]      // Range with [lower, upper] bound (inclusive)
27 hist := last |commit-IRMC window| Execute messages

28 for each client c and execution group e in parallel:
29   while true:
30     m := request-IRMC.receive(c, t+[c]) from group e
31     if m = ⟨TooOld, t_c⟩: t+[c] := t_c
32     else: // m = ⟨Request, ⟨Write, w, c, t_c⟩, e⟩
33       Consensus order request m
34       t+[c] := t+[c] + 1

35 on Consensus ordered(SeqNr s,
       r = ⟨Request, ⟨Write, w, c, t_c⟩, e⟩):
36   sleep until upper limit of win > s
37   s_n := s
38   t[c] := t_c
39   t+[c] := max(t_c + 1, t+[c])
40   commit-IRMC.send(0, s, ⟨Execute, r, s⟩) for each
       execution group e and add Execute to hist
41   if s_n ≡ 0 mod k_a:
42     create checkpoint for s_n with t, hist

43 on stable checkpoint(SeqNr s, t', hist'):
44   commit-IRMC.move_window(0, s − |hist'| + 1)
45   Consensus collect garbage before s + 1
46   if s > s_n:
47     h_missing := {⟨Execute, r, s'⟩ ∈ hist' | s' ∈ [s_n + 1, s]}
48     apply checkpoint to s_n, t and hist
49     for each execution group e:
50       send h_missing via commit-IRMC of group e
51   win := [s+1, s+AG-WIN]
```

Figure 5. Spider protocol for writes (pseudo code)

receiver-replica group (see below). Checkpoints are periodically created after a group has agreed on / processed the message for a sequence number s that satisfies $s \equiv 0 \ mod\ k$. The checkpoint interval k of a replica group is configurable and for the execution to sustain liveness must be smaller than the maximum capacity of the group's input IRMC. The agreement-checkpoint interval k_a may be selected independently from the interval for execution checkpoints k_e.

Agreement Checkpoints. Having completed the consensus process for a request for which a checkpoint is due, an agreement replica creates an agreement snapshot and includes (1) a vector t that for each client contains the counter value t_c of the client's latest agreed request and (2) the last EXECUTE messages corresponding to the commit subchannel capacity (L. 41–42 in Figure 5). In a next step, the agreement replica computes a hash h over the snapshot and sends a message $\langle$CHECKPOINT, $h, s\rangle$ protected with a digital signature to all members of its group. Having obtained f_a+1 correctly signed and matching checkpoint messages for the same sequence number, a replica has proof that its snapshot is correct. At this point, the replica can move forward its separate window used to ensure the periodic creation of a new checkpoint (L. 36 and 51) and also instruct the consensus protocol to garbage collect preceding consensus instances (L. 45).

Agreement replicas require periodic checkpoints to continue ordering new requests and thus there is at least one correct agreement replica that possesses both a corresponding valid checkpoint as well as proof of the checkpoint's correctness in the form of $f_a + 1$ matching checkpoint messages. As a consequence, if a correct agreement replica falls behind and queries its group members for the latest checkpoint, the replica will eventually be able to acquire this checkpoint, verify it, and apply it in order to catch up by skipping consensus instances. In such case, the checkpoint enables the replica to learn (1) the request-subchannel positions at which to query the IRMC for the next client requests and (2) the EXECUTES of the skipped consensus instances (L. 48–50).

Execution Checkpoints. Execution-group checkpointing follows the same basic work flow as in the agreement group. An execution snapshot comprises a copy of the application state and the latest reply to each client, similar to the checkpoints in Omada [24]. This information enables a trailing execution replica to consistently update its local state without needing to process all agreed requests. When an execution checkpoint for a sequence number s becomes stable at an execution replica, the replica moves the flow-control window of its incoming commit channel to $s+1$ (L. 19–21). This ensures that agreed requests are only discarded after at least one correct execution replica has collected a stable checkpoint. Note that there is no need for checkpoints to contain requests. A client moves its request subchannel's window forward by issuing a new request, thereby confirming that the old request can be garbage-collected from the IRMC. This also allows execution replicas to skip forward to the current request (L. 31).

3.5 Global Flow Control

With the flow-control mechanism of an IRMC only operating at the communication level between two replica groups, SPIDER takes additional measures to coordinate the message flow at the point where the endpoints of multiple IRMCs meet: the agreement group. Specifically, there are two types of messages (i.e., new requests received through request channels and EXECUTES sent through commit channels) that have individual characteristics and are handled in different ways: (1) With regard to incoming requests, agreement replicas represent the receiver side of request channels and therefore directly manage the positions of the channels' flow-control windows. As described in Section 3.4, to be able to quickly retrieve new requests an agreement replica updates the counter value of each client's latest request each time an agreement checkpoint becomes stable. (2) With regard to outgoing EXECUTES, in contrast, agreement replicas represent the sender side of commit channels and therefore depend on the respective execution group at the other end of each channel to move the flow-control window forward. To prevent a single execution group from delaying overall progress, agreement replicas in SPIDER do not wait until they are able to submit a newly produced EXECUTE to every outgoing commit channel. Instead, having completed inserting an EXECUTE for a sequence number s into $n_e - z$ commit channels an agreement replica is allowed to continue; n_e is the total number of execution groups in the system and z a configurable value ($0 \leq z < n_e$). To inform the execution groups of trailing commit channels, once such a request is garbage-collected a replica updates the channels' window positions to sequence number $s + 1$. If an affected execution replica subsequently tries to receive EXECUTES for sequence numbers of s or lower, the commit channel responds with an exception (see Section 3.2). In reaction, the execution replica starts to seek a stable execution checkpoint, querying members of both its own group and others, in order to compensate for the missed messages.

3.6 Adaptability

SPIDER's modular architecture makes it possible to dynamically change the number of execution groups in the system and thereby adjust to varying workloads. With the consensus protocol being limited to the agreement group, in contrast to traditional BFT systems such a reconfiguration in SPIDER does not require complex mechanisms or subprotocols.

Adding an Execution Group. To add a new execution group e to the system, a privileged admin client first starts the replicas of the group and then submits an $\langle$ADDGROUP, $e, \mathcal{E}\rangle$ message; $\mathcal{E}$ is a set containing the identity and address of each group member. As soon as the agreement process for this message is complete, agreement replicas establish an IRMC pair (i.e., a request channel and a commit channel) to the new execution group, update the execution-replica registry to reflect the changes, and start the reception of requests and the forwarding of Executes. Trying to obtain an Execute for sequence number 0, the new replicas will be notified by their commit channels that they have fallen behind and consequently use the mechanism of Section 3.5 to fetch an execution checkpoint from another group.

Removing an Execution Group. To remove an existing execution group e from the system, the administrator client submits a $\langle$RemoveGroup$, e\rangle$ message that, once agreed on, causes the agreement replicas to update the execution-replica registry and close their IRMCs to the affected group.

3.7 Handling Faulty Clients and Replicas

Besides enabling Spider's modular architecture, IRMCs also play a crucial role when it comes to limiting the impact faulty clients and replicas can have on the system. In this context, especially one IRMC property is of major importance: the fact that a channel only delivers a message after $f+1$ senders submitted it and the channel therefore has proof that at least one correct sender vouches for the message's validity (see Section 3.2). If, for example, a faulty client either sends conflicting requests to an execution group or the same request to fewer than $f_e + 1$ execution replicas, the request channel of the affected execution group prevents the message's delivery to the agreement group. Note that in such case the effects of the faulty client are strictly limited to the subchannel of this client, which will not deliver a request if fewer than $f_e + 1$ execution replicas insert the same message. As execution replicas use a dedicated request subchannel for each client, the subchannels of correct clients remain unaffected.

If faulty execution replicas collaborate with a faulty client, different agreement replicas may receive different values for this client's requests. For example, a faulty client might submit a different request $R_1, R_2, \ldots, R_{f_e+1}$ to each of the $f_e + 1$ correct execution replicas of one group and provide all requests to the f_e faulty execution replicas of that group. Depending on which of the request versions the faulty execution replicas transmit to which agreement replica, in such a situation it is possible that some agreement replicas obtain an $f_e + 1$ quorum for request R_1 while others receive f_e+1 matching messages for request R_2 and so on. Again, the effects are limited to the faulty client's subchannel, requests of correct clients can proceed as usual. This scenario is not specific to Spider, but in a similar way can also occur in traditional BFT systems [17, 24, 46, 51, 52], in which clients directly submit their possibly conflicting requests to the replicas performing the agreement. Consequently, all BFT protocols that tolerate faulty clients already comprise mechanisms to handle this scenario. This is usually combined with only executing client requests with a counter value which is higher than the highest value processed so far for that client, which ensures that old or duplicate requests are skipped.

Besides tolerating faulty clients, agreement protocols in general also provide means that allow correct follower replicas to elect a new leader if the current leader is faulty and, for example, fails to start the consensus process for a new client request within a given timeout [17, 24, 46, 51, 52]. To be able to monitor the leader, follower replicas must obtain information about incoming requests. In Spider, this is ensured by the fact that request channels only garbage-collect a request from a correct client if the latter has successfully obtained a valid reply. A request for which this is not the case will be uploaded to all correct members of the client's execution group and through this group's request channel eventually reach all correct follower agreement replica, thereby enabling followers to hold the leader accountable.

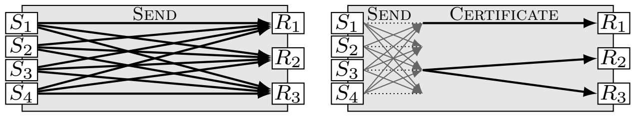
In addition, faulty agreement replicas cannot forward manipulated messages via the commit channel. As the consensus process ensures that all correct agreement replicas deliver the same total order of requests, eventually $f_a + 1$ correct agreement replicas will send matching messages enabling the execution groups to receive the correctly ordered requests. In contrast, the delivery of faulty requests sent by the faulty agreement replicas is prevented by the IRMC.

4 IRMC Implementations

In this section, we present two different variants to implement inter-regional message channels, focusing on simplicity (IRMC-RC) and efficiency (IRMC-SC), respectively. Additional variants are possible, as discussed in Section 6.

IRMC with Receiver-side Collection (IRMC-RC). The receiver endpoint of an IRMC only delivers a message m for a specific subchannel sc and position p if at least f_s+1 senders previously instructed the channel to transmit a message with identical content for the same subchannel position (see Section 3.2). As illustrated in Figure 6a, the IRMC-RC solves this problem by each sender endpoint S_x directly forwarding a $\langle$Send$, m, sc, p\rangle_{S_x,\mathcal{X}}$ message and thereby enabling each receiver endpoint to individually collect f_s+1 matching messages. To allow receivers to verify the origin and integrity of a Send, a sender signs messages with its private key $\mathcal{X}$. When a receiver requests a subchannel's flow-control window to be shifted, its receiver endpoint R_y submits a signed $\langle$Move$, sc, p\rangle_{R_y,\mathcal{Y}}$ message to all sender endpoints. For each receiver and subchannel, a sender endpoint stores the Move message with the highest position p and sets the subchannel's window start to the $f_r + 1$ highest position requested by any receiver (see Section 3.2). To request a shift of a subchannel's flow-control window, sender endpoints also send Move messages which the receivers process analogously.

IRMC with Sender-side Collection (IRMC-SC). IRMC-SCs minimize the number of messages transferred across wide-area links by applying the concept of *collectors* [28]. That is,



(a) IRMC-RC **(b)** IRMC-SC

Figure 6. Overview of two possible IRMC implementations.

sender endpoints in IRMC-SCs do not submit their SENDS to the receiver side but, as indicated in Figure 6b, instead exchange signed hashes of them within the sender group. Each sender endpoint serves as a collector, which means that the endpoint assembles a vector $\vec{v}$ of $f_s + 1$ correct signatures from different senders for the same SEND message content *sm*. Having obtained this vector, a collector S_x sends it in a signed $\langle$CERTIFICATE, $sm, \vec{v}\rangle_{S_x,\mathcal{X}}$ message to one or more receiver endpoints. On reception, a receiver verifies the validity of the CERTIFICATE by checking both the signatures of the message and the $f_s + 1$ signatures contained in the vector $\vec{v}$. If all of these signatures are correct and match the SEND message content *sm*, the endpoint has proof that *sm* is valid as it was sent by at least one correct replica and delivers the associated message to its receiver on request.

IRMC-SC receiver endpoints individually select the sender endpoint serving as their current collector and announce these decisions attached to their MOVES. As a protection against faulty collectors, all sender endpoints periodically transmit $\langle$PROGRESS, $\vec{p}\rangle_{S_x,\mathcal{X}}$ messages directly to receiver endpoints in which they include a vector $\vec{p}$ with the highest position of each subchannel for which they have a CERTIFICATE. If at least $f_s + 1$ sender endpoints claim to have reached a certain position but a receiver's collector fails to provide a corresponding and valid CERTIFICATE within a configurable amount of time, the endpoint switches to a different collector.

5 Evaluation

In this section, we experimentally evaluate SPIDER in comparison to existing approaches for BFT wide-area replication.

Environment. To compare different techniques, we implemented a Java-based prototype that can be configured to reflect three different system architectures (cf. Section 2.2): (1) **BFT** represents the traditional approach of distributing a single set of replicas across different geographic locations. It relies on PBFT [17] as agreement protocol and uses HMAC-SHA-256 as MACs to authenticate the messages exchanged between replicas. (2) **HFT** employs a hierarchical system architecture running the two-level Steward protocol [6] to coordinate multiple sites that each host a dedicated cluster of replicas. Steward requires threshold cryptography for which HFT uses the scheme proposed by Shoup [45] based on 1024-bit RSA signatures. (3) **SPIDER** represents our system architecture proposed in this paper. In this evaluation, SPIDER's agreement group runs PBFT for consensus and its IRMCs protect their messages with 1024-bit RSA signatures.

To conduct our experiments in an actual wide-area environment, we start virtual machines (t3.small, 2 VCPUs, 2 GB RAM, Ubuntu 18.04.4 LTS, OpenJDK 11) in 4 Amazon EC2 regions across the globe (Virginia, Oregon, Ireland, and Tokyo). In each of these regions, we deploy 50 clients that issue 100 writes/reads per second (200 bytes) to a key-value store provided by our systems under test; client messages carry 1024-bit RSA signatures. Given this client setting, our architectures demand the following replica placement for $f = 1$: For BFT, 1 replica is hosted in each of the 4 regions. HFT expects a cluster of 4 replicas in each region, which is used as contact cluster for local clients. For SPIDER, we deploy 1 execution group (3 replicas) per region, distributed across different availability zones. In addition, we start SPIDER's 4 agreement replicas in separate Virginia availability zones.

Writes. In our first experiment, we examine the latency of writes issued by clients at different sites. Based on the results presented in Figure 7, we make three important observations: (1) In all evaluated architectures the response times to a major degree depend on a client's geographic location. For BFT and HFT, clients in Virginia for example benefit from the fact that their local replica (cluster) experiences comparably short round-trip times when communicating with its counterparts in Oregon and Ireland. In particular, this results in low latency when the Virginia replica (cluster) acts as leader of the wide-area consensus protocol and is able to reach a quorum together with these two other

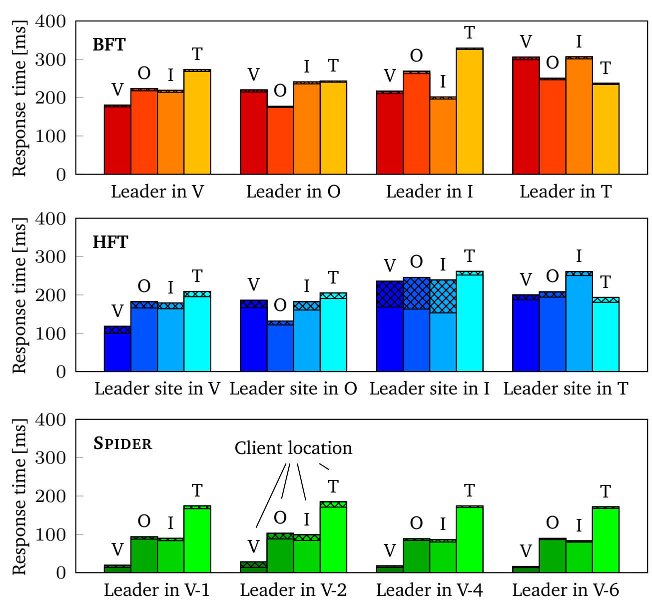
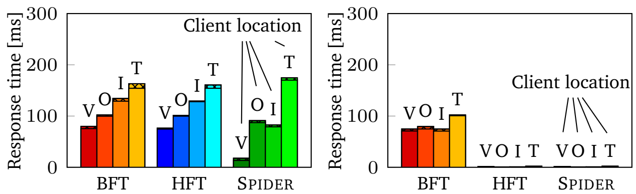


Figure 7. 50th (□) and 90th (▨) percentiles of write latencies for different client and leader locations including Virginia (V), Oregon (O), Ireland (I), and Tokyo (T).



(a) Strongly consistent reads **(b)** Weakly consistent reads

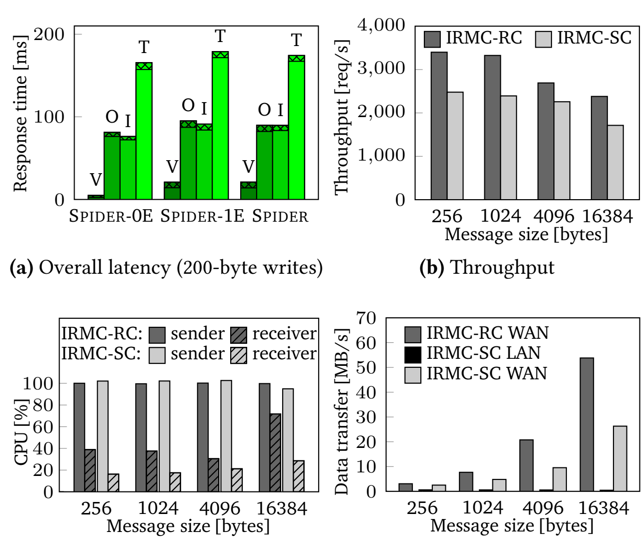
Figure 8. 50th (☐) and 90th (⊠) percentiles of read latencies.

sites. In Spider, clients in Virginia also observe low write latency, but for a different reason. Here, the fact that the agreement group resides in the same region as the clients' local execution group allows clients in Virginia to achieve response times of as low as 13 milliseconds. (2) For each client location, Spider provides significantly lower latency than BFT (up to 95 %) and HFT (up to 94 %). This is a direct consequence of the fact that in contrast to the other two system architectures Spider does not execute a full-fledged replication protocol over wide-area links. Instead, a write request only has to wait for two wide-area hops: from a client's local execution group to the agreement group and back. The distribution of the ordered write request to other execution groups is handled by the agreement group and thus does not require execution groups to explicitly wait for each other. That is, when an execution replica in Spider receives an Execute for a write from the agreement group, the replica can immediately process the operation and return a reply to the client. (3) The response times of BFT and HFT vary considerably depending on the position of the current leader of the wide-area consensus protocol. HFT clients in Ireland, for example, experience a 53 % higher latency when the leader is positioned in Tokyo compared to when the leader role is assigned to Virginia. In contrast, the specific location of the agreement-group leader in Spider only has a negligible effect on overall response times due to all agreement replicas residing in the same region, resulting in stable response times even across leader changes.

Reads. In our second experiment, we compare the evaluated architectures regarding the performance of their individual (fast-)paths for read operations with different consistency guarantees. As the results in Figure 8 show, response times of strongly consistent reads in Spider display a similar pattern as writes due to following the same path through the system. For clients in Tokyo, this leads to slightly higher response times compared with BFT and HFT, which in this case benefit from directly querying replicas without intermediaries in between. For all other client locations, Spider's approach, which only requires waiting for one wide-area round trip from a client's execution group to the agreement group and back, enables lower latency than provided by BFT and HFT. With regard to weakly consistent reads, both HFT and Spider achieve response times of 2 milliseconds or less, as these operations can be entirely handled by replicas in a client's vicinity and therefore do not require wide-area communication as in BFT.

Modularity Impact. In our third experiment, we quantify the impact of our decision to design Spider as a modular architecture that separates agreement from execution and consists of loosely coupled replica groups connected via IRMCs. We create two variants of Spider where (1) the agreement group also executes requests and is the only group in the system (Spider-0E) and (2) there is only one execution group that is co-located with the agreement group in Virginia (Spider-1E). While, Spider-0E allows us to study Spider without IRMC and externalized execution, based on Spider-1E we can assess the influence of an IRMC without wide-area delays. Our results show that when clients access Spider-0E and Spider-1E from different sites, response times are dominated by the wide-area communication between clients and replicas. Thus, the modularization overhead is small and adds less than 14 milliseconds (see Figure 9a).

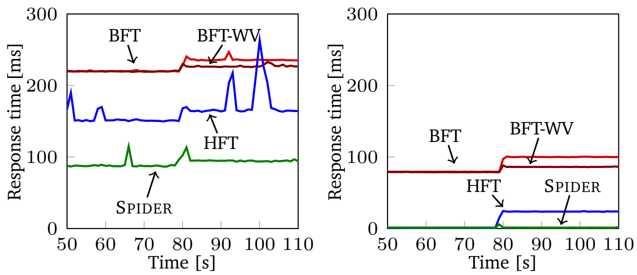
IRMC Implementations. In our fourth experiment, we evaluate the two IRMC variants presented in Section 4 by establishing a channel of each type between Virginia and Tokyo and submitting messages of different sizes. The comparison of results in Figures 9b–9d confirm the two implementations to have individual characteristics. Without the need to verify signatures for Certificate messages, IRMC-RC sender endpoints require less CPU resources per message and therefore enable IRMC-RCs to achieve a higher maximum throughput. On the other hand, forwarding only one wide-area message per receiver endpoint IRMC-SCs significantly reduce the amount of data transferred over long-distance links, thereby saving costs in public-cloud environments.



(a) Overall latency (200-byte writes) **(b)** Throughput

(c) CPU usage **(d)** Network usage

Figure 9. Performance and resource usage of IRMCs.



(a) Writes **(b)** Weakly consistent reads

Figure 10. Impact of a new client site on overall latency.

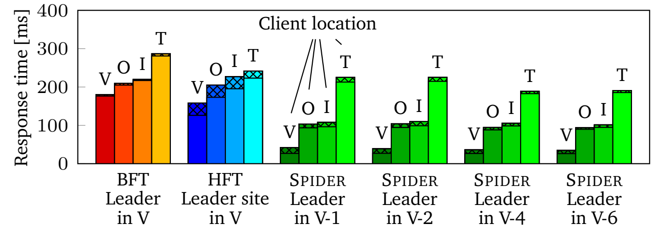


Figure 11. 50th (□) and 90th (▩) percentiles of write latencies for different client sites when tolerating $f = 2$ faults.

Adaptability. In our fifth experiment, we evaluate the write and read performance new clients experience when they join the system at an additional location. For this purpose, we start with our usual setting and after 80 seconds launch 50 clients in the EC2 region Sao Paulo. Once running, the new clients in BFT and HFT issue their requests to existing replicas, while for Spider they contact an additional execution group also set up in Sao Paulo. Involving more client sites than replica sites in BFT and HFT, the setting in this experiment represents a typical use-case scenario for weighted-voting approaches (see Section 2.2). We therefore repeat the experiment with a fourth system (BFT-WV) that extends BFT with weighted voting and comprises a replica at each of the five client locations. As required by weighted voting, two of the five replicas are assigned higher weights in the consensus protocol. Specifically, these are the replicas in Virginia and Oregon because this weight distribution achieves the best performance in our evaluation scenario. Figure 10 presents the results of this experiment showing the average response times observed across all active client sites. To save space, we omit the results for strongly consistent reads as they show a similar picture as writes. For each system, we evaluate different leader locations, but for clarity Figure 10 only reports the results of the configurations achieving the lowest response times for each system.

Figure 10a shows that the overall write latency increases for all evaluated architectures once the clients in Sao Paulo join the system. This is a consequence of the fact that due to its geographic location EC2's Sao Paulo region has comparably high transmission times to other cloud regions. Clients in Sao Paulo therefore observe response times between about 124 milliseconds (Spider) and about 298 milliseconds (BFT), which alone causes the measurable jumps in the overall write latency averages; the response times for clients in other regions remain unaffected. Interestingly, BFT and BFT-WV achieve similar write performance throughout the experiment and thereby confirm that weighted voting does not automatically improve response times. This is only true when the additional replica is located at a site that is better connected than the existing ones and therefore enables the wide-area consensus protocol to reach faster quorums. In the setting evaluated here, BFT's typical consensus quorum is based on the votes of the replicas in Virginia, Oregon, and Ireland and therefore already provides better performance than any combination that includes the replica in Sao Paulo.

As shown in Figure 10b, of the evaluated architectures Spider is the only one that allows the new clients in Sao Paulo to perform weakly consistent reads with low latency. While all other systems require the clients in Sao Paulo to read from at least one remote replica and consequently experience overall read-latency increases of up to 23 milliseconds, Spider makes it possible to introduce an execution group in the new region to efficiently handle the reads of local clients.

Tolerating Two Faults. In our final experiment, we examine write latencies for settings that are configured to tolerate $f = 2$ faults in each agreement and execution group. We place the additional replicas into nearby EC2 regions (Ohio, California, London, Seoul) to make use of further fault domains. The results in Figure 11 show that due to increased communication latency within groups both HFT and Spider see a moderate increase of response times by up to 46 milliseconds compared with the $f = 1$ setting, with Spider still providing significantly lower latency than BFT and HFT.

6 Related Work

Adaptive BFT Replication. Spider is not the first work to argue that it is crucial to enable BFT systems to dynamically adapt to changing conditions. Abstract [7] makes it possible to substitute the consensus protocol of a BFT system at runtime, for example, switching to a more robust algorithm once a replica failure has been suspected or detected. CheapBFT [30] and ReBFT [20] follow a similar idea by comprising two different agreement protocols (one for the normal case and one for fault handling) of which only one is active at a time. In contrast, the reconfiguration mechanism developed by Carvalho et al. [16] for BFT-SMaRt [14] temporarily runs two consensus algorithms in parallel to achieve a more efficient switch. As a result of Spider's modularity, integrating support for the dynamic substitution of the agreement protocol is feasible and the use of customized protocols designed for high performance [11, 37] or strong resilience [5, 8] would not require modifications to execution groups.

Other works allow BFT systems to dynamically change specific protocol properties at runtime. Depending on the current workload, de Sá et al. [19], for example, vary the parameters deciding how many requests are batched together and ordered within a single consensus instance. Berger et al. [12] rely on a weighted voting scheme [46] and by changing weights adjust the individual impact a replica has on the outcome of the agreement process. While adapting the batch size can be a measure to improve the performance of Spider's agreement group, the use of a weighted voting scheme in general is only effective if (1) a system contains more than the minimum number of agreement replicas and (2) agreement replicas are located in different geographic regions; both of these points do not apply to Spider.

Communication Between Replica Groups. Amir et al. proposed BLinks [4] as a means to send the totally ordered outputs of one replicated state machine to another replicated state machine that uses them as inputs. Unfortunately, the requirement of a channel-wide total order prevents Spider from relying on BLinks as execution replicas do not necessarily have to use the same order when submitting new requests to the agreement group via their request channels. IRMCs, on the other hand, do not have this restriction and furthermore comprise a built-in flow-control mechanism that represents the basis of Spider's global flow control. However, transmitting only a single message between one dedicated sender and one dedicated receiver, BLinks may be used as a template for an IRMC implementation that involves even fewer wide-area messages than IRMC-SC.

Partitioned Agreement Groups. GeoBFT [29] makes use of replica groups located in different regions, which each run a full agreement protocol. In each protocol round every group orders a request yielding a request certificate, which is shared with all other groups. Afterwards the requests are merged into a single total order and are executed. This requires all groups to distribute a certificate in every round, even if it just contains a placeholder request, and thus all groups must work at the same time to make progress. In Spider this requirement only applies to the agreement group whereas a limited number of slow execution groups can be skipped. Sharing a request ordering certificate in GeoBFT works by having the leader replica forward it to $f + 1$ replicas of each group, which then further forward the certificate within their group. This request distribution scheme represents a middle ground between BLinks and IRMC-SCs. Unlike IRMCs it is coupled with the agreement protocol and has to remotely trigger a view-change to replace a leader replica which does not complete the request distribution in a timely manner.

Efficient Client Communication. In most BFT systems, clients need to receive replies from different replicas in order to prove a result correct [17], which in geo-replicated settings can significantly increase the number of messages exchanged over wide-area links. SBFT [28] addresses this problem by adding a protocol phase that aggregates request acknowledgements of multiple replicas into a single message to the client. In Troxy [32], a client also has to wait for a single reply only, because the reply voter is hosted inside a trusted domain at the server side and forwards its decisions to the client through a secure channel. In Spider, clients are typically located in the same region as an execution group allowing for communication over short-distance links. For scenarios in which this is not the case, it would be possible to extend Spider to use one of the approaches discussed above.

Leader Selection in Geo-replicated Systems. Multiple authors have underlined the impact that the leader-replica location has on response times, independent of the fault model, and presented solutions to select the leader in a way that minimizes overall latency [23, 34, 46]. Other agreement-based systems do not need to determine a fixed leader as they continuously rotate the leader role among replicas [35, 36, 39, 50, 51]. As our experiments show, with agreement replicas residing in different availability zones of the same cloud region, the specific location of the consensus leader in Spider only has a negligible effect on response times. Consequently, Spider achieves low and stable latency without requiring means to dynamically select or rotate the leader.

Crash-tolerant Wide-Area Replication. Several works addressed the efficiency of geo-replication in systems that unlike Spider solely tolerate crashes, not Byzantine failures. In Pileus [48], for example, writes are only handled by a subset of replicas that first order and execute them, and then bring all other replicas up to date by transferring state changes. P-Store [44] improves efficiency in wide-area environments by performing partial replication, thereby freeing a site from the need to receive and process all updates. Clock-RSM [21] establishes a total order on requests by exploiting the timestamps of physical clocks and without requiring a dedicated leader replica. EPaxos [40] in contrast does not rely on a total request order, but only orders those requests that interfere with each other due to accessing the same state parts.

7 Conclusion

The cloud-based Spider system architecture models a BFT system as a collection of loosely coupled replica groups that can be flexibly distributed in geo-replicated environments. In contrast to existing approaches, Spider does not require the execution of complex multi-phase protocols over wide-area links, but instead performs essential tasks such as consensus, leader election, and checkpointing across replicas residing in the same region. Our experiments show that this approach enables Spider to achieve low and stable response times.

Acknowledgments

This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 (“REFIT”).

References

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st Symposium on Cloud Computing (SoCC '10)*. 229–240.

[2] Amazon EC2. 2020. Regions and Availability Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[3] Amazon Web Services. 2011. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/.

[4] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2007. Customizable Fault Tolerance for Wide-Area Replication. In *Proceedings of the 26th International Symposium on Reliable Distributed Systems (SRDS '07)*. 65–82.

[5] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2010. Prime: Byzantine Replication Under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2010), 564–577.

[6] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (2010), 80–93.

[7] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems* 32, 4 (2015), 12:1–12:45.

[8] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS '13)*. 297–306.

[9] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 328–335.

[10] Amy Babay, Thomas Tantillo, Trevor Aron, Marco Platania, and Yair Amir. 2018. Network-attack-resilient Intrusion-tolerant SCADA for the Power Grid. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 255–266.

[11] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2015. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proceedings of the 16th Middleware Conference (Middleware '15)*. 173–184.

[12] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. 2019. Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*.

[13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage (TOS)* 9, 4 (2013), 12:1–12:33.

[14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*. 355–362.

[15] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2008. The CRUTIAL Way of Critical Infrastructure Protection. *IEEE Security & Privacy* 6, 6 (2008), 44–51.

[16] Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. 2018. Dynamic Adaptation of Byzantine Consensus Protocols. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. 411–418.

[17] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. 173–186.

[18] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. UpRight Cluster Services. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*. 277–290.

[19] Alírio Santos de Sá, Allan Edgard Silva Freitas, and Raimundo José de Araújo Macêdo. 2013. Adaptive Request Batching for Byzantine Replication. *SIGOPS Operating System Review* 47, 1 (2013), 35–42.

[20] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. 2016. Resource-efficient Byzantine Fault Tolerance. *IEEE Trans. Comput.* 65, 9 (2016), 2807–2819.

[21] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2014. Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *Proceedings of the 44th International Conference on Dependable Systems Networks (DSN '14)*. 343–354.

[22] Michael Eischer, Markus Büttner, and Tobias Distler. 2019. Deterministic Fuzzy Checkpoints. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*.

[23] Michael Eischer and Tobias Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18)*. 140–145.

[24] Michael Eischer and Tobias Distler. 2019. Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers. *Computing* 101, 2 (2019), 97–118.

[25] Michael Eischer and Tobias Distler. 2020. Resilient Cloud-based Replication with Low Latency (Extended Version). arXiv:2009.10043 [cs.DC]

[26] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2016. SieveQ: A Layered BFT Protection System for Critical Services. *IEEE Transactions on Dependable and Secure Computing* 15, 3 (2016), 511–525.

[27] Google Compute Engine. 2020. Regions and Zones. https://cloud.google.com/compute/docs/regions-zones/.

[28] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 568–580.

[29] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[30] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*. 295–308.

[31] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[32] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. 2018. Troxy: Transparent Access to Byzantine Fault-Tolerant Systems. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 59–70.

[33] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*. 77–88.

[34] Shengyun Liu and Marko Vukolić. 2017. Leader Set Selection for Low-Latency Geo-Replicated State Machine. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1933–1946.

[35] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*. 369–384.

[36] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2009. Towards Low Latency State Machine Replication for Uncivil Wide-Area Networks. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep '09)*.

[37] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[38] Microsoft Azure. 2020. Azure Regions. https://azure.microsoft.com/en-us/global-infrastructure/regions/.

[39] Zarko Milosevic, Martin Biely, and André Schiper. 2013. Bounded Delay in Byzantine-Tolerant State Machine Replication. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems (SRDS '13)*. 61–70.

[40] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 358–372.

[41] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. 2018. On the Challenges of Building a BFT SCADA. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 163–170.

[42] Ricardo Padilha, Enrique Fynn, Robert Soulé, and Fernando Pedone. 2016. Callinicos: Robust Transactional Storage for Distributed Data Structures. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC '16)*. 223–235.

[43] Ricardo Padilha and Fernando Pedone. 2013. Augustus: Scalable and Robust Storage for Cloud Applications. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)*. 99–112.

[44] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. 2010. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proceedings of the 29th International Symposium on Reliable Distributed Systems (SRDS '10)*. 214–224.

[45] Victor Shoup. 2000. Practical Threshold Signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT '00)*. 207–220.

[46] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems (SRDS '15)*. 146–155.

[47] Joao Sousa, Alysson Bessani, and Marko Vukolić. 2018. A Byzantine Fault-tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 51–58.

[48] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Agu ilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 309–324.

[49] Gene Tsudik. 1992. Message Authentication with One-Way Hash Functions. *ACM SIGCOMM Computer Communication Review* 22, 5 (1992), 29–38.

[50] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS '09)*. 135–144.

[51] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 12th Symposium on High-Assurance Systems Engineering (HASE '10)*. 10–19.

[52] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*. 253–267.