On Choosing Server- or Client-Side Solutions for BFT

MARCO PLATANIA, DANIEL OBENSHAIN, THOMAS TANTILLO, and YAIR AMIR,
Computer Science Department, Johns Hopkins University
NEERAJ SURI, Computer Science Department, TU Darmstadt

Byzantine Fault Tolerant (BFT) protocols have the ability to work correctly even when up to a threshold f of system servers are compromised. This makes them appealing for the construction of critical systems connected to the Internet, which are constantly a target for cyber attacks.

BFT protocols differ based on the kind of application, deployment settings, performance, access control mechanisms, number of servers in the system, and protocol implementation. The large number of protocols present in the literature and their differences make it difficult for a system builder to choose the solution that best satisfies the requirements of the system that he wants to build. In particular, the main difference among BFT protocols lies in their system models: server-side versus client-side. In the server-side model each client relies on the system to consistently order and replicate updates, while in the client-side model each client actively participates in the protocol.

In this article, we classify BFT protocols as server-side or client-side. We analyze the trade-offs between the two models, describe systems that use these models and the trade-offs they choose, highlight the research gaps, and provide guidelines to system builders in order to choose the solution that best satisfies their needs.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Fault-Tolerance

General Terms: Byzantine Fault Tolerance, Algorithms, Performance

Additional Key Words and Phrases: BFT state machine replication, BFT quorums, performance, deployment strategies, trade-offs

ACM Reference Format:
Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. 2016. On choosing server- or client-side solutions for BFT. ACM Comput. Surv. 48, 4, Article 61 (March 2016), 30 pages.
DOI: http://dx.doi.org/10.1145/2886780

1. INTRODUCTION

Byzantine Fault Tolerant (BFT) protocols guarantee the correct execution of operations even when up to a threshold f of servers show arbitrary (i.e., Byzantine) behavior. This makes BFT protocols an appealing building block for the construction of intrusion-tolerant systems. Nowadays many distributed systems, such as clouds and critical infrastructures, are connected to the Internet, and thus are a possible target for cyber attacks. As such, the ability to withstand faults provoked by an attack is paramount.

This work was supported in part by DARPA grant N660001-1-2-4014. Its contents are solely the responsibility of the authors and do not represent the official view of DARPA or the Department of Defense.
This work was also supported in part by TUD EC-SPRIDE and Loewe CASED projects.
Authors' addresses: M. Platania, D. Obenshain, T. Tantillo, and Y. Amir, Computer Science Department, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA; emails: {platania, dano, tantillo, yairamir}@cs.jhu.edu; N. Suri, Computer Science Department, TU Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Germany; email: suri@cs.tu-darmstadt.de.

DOI: http://dx.doi.org/10.1145/2886780

The problem of agreeing on a binary value in a Byzantine environment [Pease et al. 1980] was later generalized to the Byzantine Generals problem [Lamport et al. 1982] (see Section 2). In the last 20 years, a large number of BFT protocols have been presented. PBFT [Castro and Liskov 1999] was the first to demonstrate that BFT protocols can have good performance and be practical. After that, many BFT protocols were proposed with the goal of optimizing normal-case performance. They either use speculative execution [Kotla et al. 2007], rely on a replier set known in advance [Serafini et al. 2010; Abd-El-Malek et al. 2005], or achieve optimal latency by integrating a consensus primitive with a state machine replication protocol [Sousa and Bessani 2012]. Recently, some protocols have been proposed that focus on how to guarantee performance while under attack. Prime [Amir et al. 2008] and BFT-Mencius [Milosevic et al. 2013] execute client operations within a bounded delay when the network is synchronous enough, while Aardvark [Clement et al. 2009b] guarantees that under attack the system reaches a throughput that is within a constant factor of the throughput that the system reaches under normal operations.

The BFT protocols mentioned previously differ from each other based on the specific applications and their requirements, such as performance, scalability over geodistributed systems, message authentication, number of servers in the system, and protocol implementation. As an example, PBFT can execute thousands of operations per second in the absence of attacks when deployed on a local cluster, while Q/U [Abd-El-Malek et al. 2005] scales better than PBFT in terms of performance on wide area networks, due to loose synchronization among servers. On the other hand, Q/U requires $5f + 1$ servers in the system, while PBFT requires only $3f + 1$ servers, which is the minimum number of servers required to solve consensus in a Byzantine environment [Pease et al. 1980].

However, the main difference among these protocols lies in their system models. We divide current BFT protocols into two categories: server side and client side.[1] This distinction takes into account the role that clients play during the execution of the BFT protocol.

In the server-side model the clients do not actively participate in the protocol execution. A client only sends a request to one or more servers and waits to receive at least $f + 1$ identical replies from different servers before completing the request. The BFT protocol is completely run by the servers. Typically, one of these plays the role of the leader and initiates a protocol round when it receives a client request directly from that client or from some other server in the system. Then, servers exchange messages to reach an agreement on the order in which client operations should be executed.

In contrast to the server-side model, in the client-side model the clients actively participate in the protocol. The role played by the clients depends on the kind of application. Clients may act either as *proposers* [Malkhi and Reiter 1998c; Abd-El-Malek et al. 2005], *repairers* [Kotla et al. 2007; Serafini et al. 2010; Abd-El-Malek et al. 2005], or both [Abd-El-Malek et al. 2005]. When the clients act as *proposers*, they initiate a round of the protocol by proposing a sequence number for the next operation they want to execute. This model is typically adopted by applications in which the same resource is updated by a single client or a few clients. When the clients act as *repairers*, they drive system reconfiguration in case of conflicting updates or server failures.

The motivation for our work is that including clients in the computation extends the boundaries of the system, which has an impact on many factors, such as

—Security: the impact of malicious client and how the system should be protected;
—Protocol configuration: the protocol implementation and its parameters (e.g., number of servers) and how these change when clients are part of the system; and

[1]We consider the terms *server side* and *client side* to convey a loose, conceptual pedagogical classification rather than a formal definition.

—Performance: how the performance of the system improves when part of the job is shifted to the client side.

In this article, we analyze many BFT protocols present in the literature and classify each as server side or client side. For both approaches we present baseline protocols, describe possible attacks on them, and show how different BFT protocols cope with malicious attacks or propose strategies that improve the performance of the system. We also provide a description of the trade-offs between server-side and client-side models and how existing applications chose those trade-offs. Our ultimate goal is to provide guidelines to system builders to help them choose the best model (server side or client side) based on the requirements of the system that they want to build. Hence, we provide an analysis from a system-oriented point of view, which is orthogonal to previous work [Merideth and Reiter 2010; Dantas et al. 2007] that categorizes BFT protocols as *quorum-based* or *state machine replication* approaches.

Note that some BFT protocols described in the literature [Guerraoui et al. 2010; Cowling et al. 2006] include aspects of both models. They follow the client-side approach during normal-case executions to improve the performance of the system and switch to the server-side model to resolve conflicts. Based on the definitions of server-side and client-side models (see Section 3), in this article we label them as client side because clients actively participate in the protocols by acting as *proposers* or *repairers*.

The remainder of the article is organized as follows. Section 2 introduces the BFT problem. Section 3 defines server-side and client-side systems, and the attack model we consider throughout the article. Sections 4 and 5 describe the reference server-side and client-side protocols, respectively. Section 6 presents two *hybrid* protocols and motivates why we label them as client side. Section 7 describes applications that use the server-side or client-side models, highlighting the differences between these models. Section 8 presents the trade-offs between the two models and how different applications made choices. Section 9 discusses the main aspects that a system builder should consider for building a practical BFT application, based on the lessons we learn in this work. Finally, Section 10 concludes the article.

2. THE BFT PROBLEM

The BFT problem was introduced by Pease et al. [1980]. The article discusses how to reach agreement in a system with n processors, where at most f of them can be faulty. Each correct processor i communicates a private value v_i to all other correct processors, while faulty processors may lie.

The goal of the article is to devise an algorithm that guarantees *interactive consistency* [Pease et al. 1980], that is, each correct processor has to compute a vector of values, one for each processor, such that

(1) correct processors compute the same vector; and
(2) the value v_i corresponding to a correct processor i is the private value of that processor.

Because of point (1), every correct processor must compute exactly the same value for each faulty processor. These values can be arbitrary. The article demonstrates that a solution to this problem requires $n \geq 3f + 1$ processors, and shows impossibility results for $n \leq 3f$.

The BFT problem [Pease et al. 1980] has been generalized later to the Byzantine Generals problem [Lamport et al. 1982]: a group of Byzantine Generals, some of whom may be traitors, surround an enemy city and must agree on a plan of action. This problem is used to abstract a reliable computer system that must be able to cope with the failure of one or more components.

3. DEFINITIONS

In this section, we categorize *server-side* and *client-side* systems. We consider the terms *server side* and *client side* to convey a loose, conceptual pedagogical classification rather than a formal definition. Moreover, we present the attack model that we will use in the next sections to show vulnerabilities of BFT protocols.

3.1. Server-Side and Client-Side Systems

—**Server-side system:** We say a system is *server side* if clients only send requests to one or more servers and wait for replies from multiple servers.
—**Client-side system:** We say a system is *client side* if clients may act either as *proposers*, *repairers*, or both. A *proposer* is an entity that initiates an ordering round by proposing a sequence number for the operation that it wants to execute. A *repairer* is an entity that drives system reconfiguration when the normal case execution of the BFT protocol fails.

The main difference between these two approaches is that in server-side systems the clients do not actively participate in the protocol; they submit requests and wait for the servers to complete the execution. On the contrary, in client-side systems the clients actively participate in the protocol execution by playing one or more roles (i.e., *proposer, repairer*). We will describe in the next sections how this paradigm change affects security, protocol configuration, and system performance.

3.2. Attack Model

We consider a powerful adversary that can compromise clients and servers, delay the sending and receipt of messages of compromised nodes (clients or servers), but cannot delay messages exchanged by correct nodes. We assume that the adversary is computationally bounded and cannot subvert the cryptographic assumptions that the BFT protocols described in the next sections make. We group malicious attacks that the adversary can launch into two different categories: *performance attacks* and *correctness attacks*.

—**Performance attack:** We define *performance attack* to mean a Denial of Service (DoS) attack that compromised clients or servers may launch to slow down system progress.
—**Correctness attack:** We define *correctness attack* to mean an attack that compromised clients or servers may launch to generate inconsistencies in the system.

In the following sections, we will describe how these attacks can affect both server-side and client-side systems and how different BFT solutions cope with them.

4. SERVER-SIDE MODEL

BFT protocols that follow the server-side model are entirely run by servers. Clients do not participate in the protocol execution; they only submit requests to one or more servers and wait for at least $f + 1$ equal replies from distinct servers before executing an operation. Typically, server-side BFT protocols use the state machine replication approach [Lamport 1978; Schneider 1990] for managing replicated states. Correct servers start from the same state and modify their states only when they execute client operations (i.e., updates). State changes are deterministic: if two servers execute the same operations in the same order, they will move through exactly the same sequence of states. Maintaining consistent state in server-side protocols typically requires interaction among servers.

An example of a server-side protocol is PBFT [Castro and Liskov 1999]. It uses $n \geq 3f + 1$ servers, where f is the maximum number of servers that can be Byzantine.

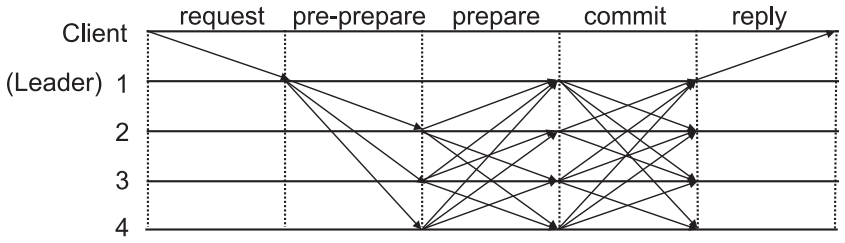

Fig. 1. Execution of PBFT with $f = 1$ and $n = 4$. The protocol is composed of *pre-prepare, prepare*, and *commit* phases.

Any number of clients can be Byzantine. Clients and servers use digital signatures [Diffie and Hellman 1976; Rivest et al. 1978] to authenticate the messages they send. The service uses a client's identity to deny access to a client that is not allowed to invoke an operation.

Figure 1 depicts an execution of PBFT, with $n = 4$ and $f = 1$. One of the servers plays the role of the *proposer*, also known as *leader*. The leader initiates and coordinates an agreement protocol that orders client updates by assigning a sequence number to each of them.

A client sends a request to the leader and waits to receive at least $f + 1$ identical replies from different servers before considering that operation to be executed. If the client does not receive replies before the expiration of a timeout, it resends the request to $f+1$ distinct servers. The ordering round consists of three phases: *pre-prepare, prepare,* and *commit*. The pre-prepare and prepare phases order client operations within a view, while the commit phase guarantees order across different views (i.e., when the leader changes). The leader is replaced if the system does not make progress before the expiration of a timeout. In this case, a new leader is elected and the timeout is doubled (i.e., the protocol pessimistically assumes that there was not enough time to make progress).

4.1. Possible Attacks on the System

Performance Attacks. Previous work [Amir et al. 2008] has shown that PBFT is vulnerable to *Timeout Manipulation* and *Pre-Prepare Delay* attacks, two DoS attacks that a malicious leader can launch to make the system progress slow.

—*Timeout Manipulation*: this attack exploits the fact that when the leader is replaced the timeout doubles. A malicious client launches a DoS attack in order to slow down the protocol and force view changes. This attack stops when the timeout is large enough and a malicious server plays the role of the leader.
—*Pre-Prepare Delay*: this attack exploits the fact that the leader has to send at least one PRE-PREPARE message before the expiration of the timeout in order to stay in power. A malicious leader can slow down the protocol by injecting only one PRE-PREPARE message per timeout.

The combination of these two attacks can dramatically decrease the performance of the system. The adversary can first manipulate the timeout to grow arbitrarily, until a malicious server plays the role of the leader. Then, the malicious leader can send a PRE-PREPARE message just before the expiration of the timeout, such that it will never be suspected by correct servers. Under these two attacks, PBFT may slow down to 1% of the performance it achieves during normal operations [Amir et al. 2008]. In the next subsection we describe how these performance attacks can be addressed so as to guarantee good performance even in the case of compromises.

Correctness Attacks. A correct server completes prepare and commit phases when it receives at least $2f + 1$ PREPARE and COMMIT messages that say the same thing. This means that if the number of faulty servers is no higher than f, the PBFT protocol guarantees *safety*, that is, all correct servers maintain consistent state, and *liveness*, that is, eventual progress. In the same way, if the number of faulty servers is no higher than f, malicious clients cannot violate *safety* because it is impossible to find two sets of $2f + 1$ servers that assign two different sequence numbers to the same update. Even if a malicious client cannot leave the system in an inconsistent state [Castro and Liskov 1999], a malicious client may still erase the state of the system by proposing a *null* value (this can be avoided by having the application checking the content of an operation before executing it).

4.2. Server-Side Model Optimizations

In this subsection, we survey some optimizations to the server-side model with the goal of improving system security or performance. All the algorithms we present use PBFT as the baseline agreement protocol, except for Fast Byzantine Paxos (FaB) [Martin and Alvisi 2006], which instead extends Paxos [Lamport 1998] to tolerate Byzantine faults.

Replacing Signatures with MACs. PBFT can be configured to use Message Authentication Codes (MACs) [Bellare et al. 1996] instead of digital signatures [Castro and Liskov 2002]. MACs are computationally less expensive than digital signatures. A server i shares a pair of keys with each other server j, and a single key with each client. However, when using MACs instead of signatures, a faulty client can force the replacement of a correct leader. The faulty client may send a request with a correct authenticator to at least one correct server, while sending an incorrect authenticator to the leader. This way, the leader cannot propose that request for ordering and will eventually be replaced. This is a kind of DoS attack that a malicious client can launch, generating view changes continuously to slow down the system. This problem can be avoided by requiring clients to sign the messages they send [Garcia et al. 2013].

Proactive / Reactive Recovery. PBFT can be combined with proactive recovery [Castro and Liskov 2002]: periodically, one server at a time is rejuvenated from a clean execution environment and application state. This way, if that server was compromised, after rejuvenation it is once again correct.

Recovery operations can be optimized by reactively recovering servers that show arbitrary behavior [Sousa et al. 2010]. In the proposed solution, the system is composed of two distinct subsystems: (i) payload, and (ii) wormhole. The payload is an *any-synchronous* subsystem that is composed of $n = 3f + 2k + 1$ servers, where k is the maximum number of servers that rejuvenate at the same time. The payload implements the BFT replication engine. The wormhole is a *synchronous* subsystem composed of $n = 3f + 2k + 1$ trusted components connected by a synchronous and private (i.e., assumed secure) network. At most f wormholes can crash (fail-stop). The wormhole subsystem coordinates rejuvenations of the servers in the payload subsystem. Each server in the payload subsystem is associated with a trusted component in the wormhole subsystem. A server and its associated trusted component can run on the same machine or on different machines.

Performance Guarantees while Under Attack. Prime [Amir et al. 2008], Aardvark [Clement et al. 2009b], and BFT-Mencius [Milosevic et al. 2013] overcome the performance limitations of PBFT under the attacks described in Section 4.1 by limiting the power of malicious servers.

Prime ensures that, if the network is stable enough, every client operation will be ordered by correct servers within a bounded delay that is a function of the latency

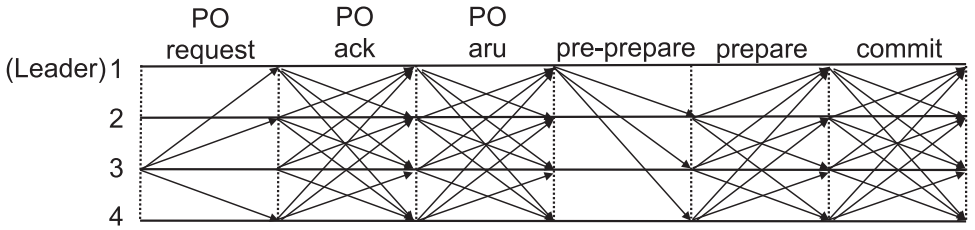

Fig. 2. The preordering phase in Prime is used to discover a malicious leader that drops client requests. The ordering phase works as in PBFT ($f = 1, n = 4$).

among those servers. To do so, Prime servers run a background protocol that monitors the activity of the leader and replaces a leader that is performing too slowly, according to the current network conditions. Moreover, Prime servers run a preordering phase to discover a malicious leader that drops client requests. A client sends a request to one server, which includes that request in a PO_REQUEST message. Periodically, the most recently received client requests are cumulatively acknowledged by each server in a PO_ARU message. A correct server includes in the PO_ARU message all the client requests that it expects from the leader during the ordering phase. If the leader skips some of these client requests, it is suspected and replaced. The ordering phase in Prime works as in PBFT. Preordering and ordering algorithms are shown in Figure 2.

BFT-Mencius [Milosevic et al. 2013] offers the same bounded-delay guarantee as Prime. However, unlike Prime, BFT-Mencius is a multileader protocol that does not use any preordering phase. BFT-Mencius relies on Abortable Timely Announced Broadcast (ATAB) [Milosevic et al. 2013], a BFT reliable broadcast primitive that servers use to order client updates. The implementation of the ATAB protocol is similar to PBFT, with different servers leading different parallel instances of the protocol. With respect to Prime, the absence of the preordering phase in BFT-Mencius reduces the number of message exchanges. Hence, BFT-Mencius reaches a tighter upper bound on the execution delay of client updates introduced by correct servers.

Aardvark, instead, guarantees that even in the presence of some malicious servers, over a sufficiently long period of time the throughput remains within a constant factor of the throughput that could be achieved if the system were composed of only correct servers. Aardvark is built around three key principles:

(1) *Resource Isolation*. Separate network interface controllers (NICs) and wires are used to connect each pair of servers. A separate NIC for clients is also used. Messages coming from different NICs are placed on different queues.
(2) *Regular View Changes*. The level of work expected from the leader is gradually increased. When the current leader is not able to sustain that throughput, a new leader is elected. Moreover, regular view changes ensure that a malicious leader is replaced in a short amount of time.
(3) *Limiting the Impact of Malicious Cclients*. A decision tree is used to limit the number of operations and resource utilization in response to requests from faulty clients.

The agreement protocol in Aardvark is the same as PBFT (see Figure 1).

Reducing the Number of Communication Steps. Fast Byzantine Paxos (FaB) [Martin and Alvisi 2006] extends the crash-tolerant Paxos replication protocol [Lamport 1998] to tolerate Byzantine faults. FaB is composed of $3f + 1$ *proposers*, $5f + 1$ *acceptors*, and $3f + 1$ *learners*. Typically, one proposer, one acceptor, and one learner are deployed on the same physical machine; hence, FaB requires $5f + 1$ servers in total. In the common

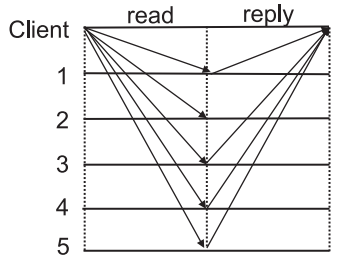

Fig. 3. Read operation with $f = 1, n = 5$.

case, an ordering round in FaB completes in only two phases. In the first phase, one proposer (i.e., the leader of that ordering round) proposes a value to the acceptors. In the second phase, the acceptors accept that value and forward it to the learners, which learn the accepted value when they receive $4f + 1$ ACCEPT messages from different acceptors. Compared with the BFT protocols discussed so far, FaB reduces the number of communication rounds at the cost of $2f$ additional servers in the system. Finally, FaB uses MACs during normal case operations and digital signatures during a view change.

5. CLIENT-SIDE MODEL

Unlike the server-side model, in the client-side model clients play an active role. They act either as *proposers* (i.e., they initiate an instance of the protocol by sending an operation and a sequence number to servers), *repairers* (i.e., they coordinate system reconfiguration in case of conflicting updates or failures), or both. As we will see in the remainder of the section, shifting part of the work to the client side requires less server-to-server interaction and provides better scalability on Wide-Area Networks (WANs) than server-side protocols. However, involving clients in the computation enlarges the boundaries of the system. System builders have to provide additional security mechanisms to prevent malicious clients from violating system correctness.

5.1. Clients as Proposers

When a client plays the role of *proposer*, it acts as leader of its own operations. This approach has been used to build consensus objects [Malkhi and Reiter 1998c]. The baseline protocol uses the abstraction of Byzantine Quorums [Malkhi and Reiter 1998a] to mask Byzantine failures in the emulation of a distributed register. This register stores a single-writer, multireader object for each client called *timed append-only array*. A client can update its own object by appending a new value to the array in sequential order. The system requires $4f + 1$ servers, where f is the maximum number of Byzantine servers. When the client submits an operation (*read* or *append*), it needs to receive a consistent reply from a quorum of $2f + 1$ servers before moving on to the next phase of the protocol. *read* and *append* operations do not require server-to-server or client-to-client interaction. All the messages exchanged between clients and servers are digitally signed. The *read* operation proceeds as shown in Figure 3: the client sends a request to all servers in the system to read the i^{th} element of an array and waits to receive a consistent reply from a quorum of servers.

The *append* operation is shown in Figure 4. It requires three rounds:

(1) The client reads a new time stamp from a quorum of servers. Signatures of the reply messages form a digital certificate.
(2) The client *appends* the i^{th} element to its own array by proposing the new value and the received time stamp (the client also attaches to the message the digital

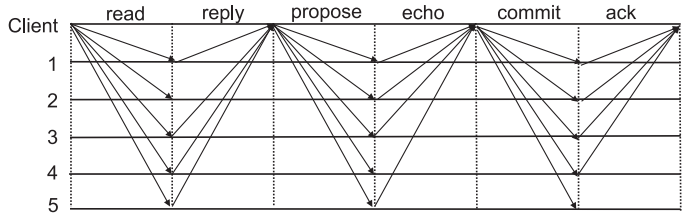

Fig. 4. Write operation with $f = 1, n = 5$. The client receives a new time stamp from a quorum of servers and writes the new value to another quorum.

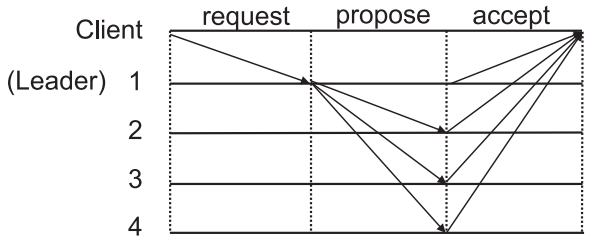

Fig. 5. Speculative execution in Zyzzyva ($f = 1, n = 4$).

certificate collected during the first round). The client waits for digitally signed ECHO messages from a quorum of servers (echo certificate).

(3) The client commits the *append* operation by sending the echo certificate back to the servers and waits for acknowledgments from a quorum of servers.

The consensus protocol is built on top of the baseline protocol composed of *read* and *append* operations. Each client starts the consensus protocol by proposing an initial value and reading all the values proposed by all other clients. The protocol proceeds in rounds: at each round, clients propose one of the values proposed in the previous round by some client, until all clients converge to the same value. This is the value that the consensus protocol *decides*. In the next section, we will see how this consensus protocol has been used to build a Byzantine fault-tolerant coordination system called Phalanx [Malkhi and Reiter 1998b].

5.2. Clients as Repairers

When the clients act as *repairers*, they drive system reconfiguration in the case of conflicting replies from servers or other failures. Examples of such a protocol are Zyzzyva [Kotla et al. 2007] and Scrooge [Serafini et al. 2010].

Zyzzyva [Kotla et al. 2007] is a BFT protocol that achieves fast agreement, also known as *speculative* execution, by ordering client requests in only three one-way message rounds. The protocol requires $3f + 1$ servers and can be configured to use digital signatures or MACs, while clients are always required to sign messages to enforce access control.

Figure 5 shows the speculative protocol: (i) the client sends a request to the leader; (ii) the leader proposes a sequence number; and (iii) all the other servers optimistically accept the sequence number proposed by the leader.

The speculative execution is possible when (i) the leader is correct, (ii) the client is honest, (iii) the network is synchronous enough, and (iv) the client receives $3f + 1$ replies. If one of these requirements fails, five communication rounds are necessary, as shown in Figure 6. In addition to the thre rounds described previously, the client

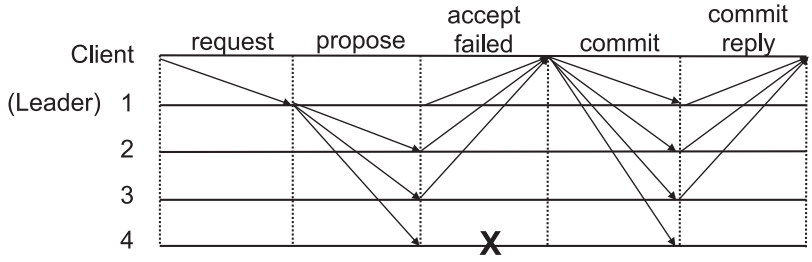

Fig. 6. Zyzzyva requires five rounds when the speculative execution is not possible ($f = 1, n = 4$).

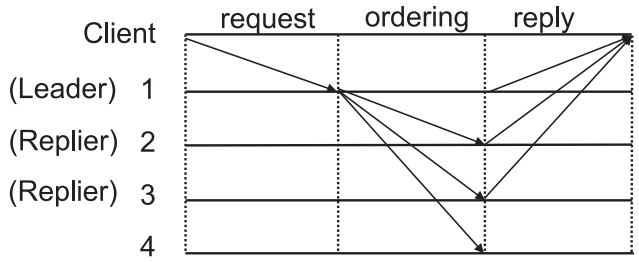

Fig. 7. Fast agreement in Scrooge obtained through a designated replier quorum ($b = 1$, $f = 1, n = 4$).

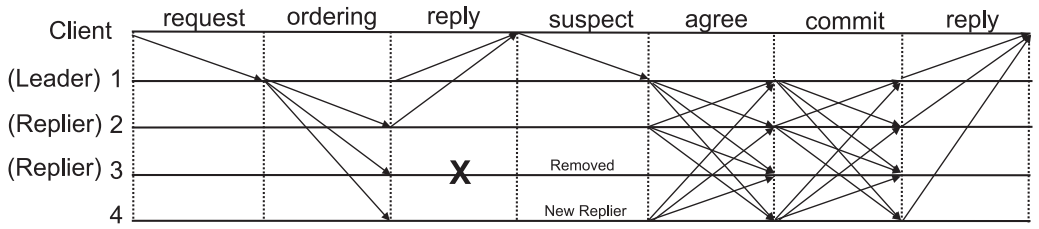

Fig. 8. A Scrooge client triggers explicit agreement and replier quorum reconfiguration if it does not receive $n - f$ replies within a timeout ($b = 1$, $f = 1, n = 4$).

builds and sends a commit certificate and waits to receive at least $2f + 1$ replies from different servers.

Moreover, correct clients lead the system reconfiguration (i.e., view change) in the presence of a faulty or slow leader. In case of faulty leader, a correct client can build a *proof of misbehavior* to prove that it received different sequence numbers for the same operation from the leader. When the leader is slow, a correct client resends an unordered operation to all the servers in the system, which will forward it to the leader and wait for the leader to start a new agreement phase within a timeout. If this does not happen, the leader is suspected and replaced.

As with Zyzzyva, Scrooge [Serafini et al. 2010] also provides speculative execution. The protocol requires $n = 2b + 2f$ servers, with $f \leq b$, where b is the maximum number of servers that can crash and f is the maximum number of malicious servers. Scrooge uses a replier quorum of $n - f$ servers known in advance to achieve fast Byzantine agreement. The execution is shown in Figure 7. A client sends a request to the leader. The leader establishes an order for that request and the servers in the replier quorum reply back to the client.

If the client does not receive $n - f$ replies within a timeout, it triggers an explicit agreement phase, as shown in Figure 8. The client also provides a list of suspected

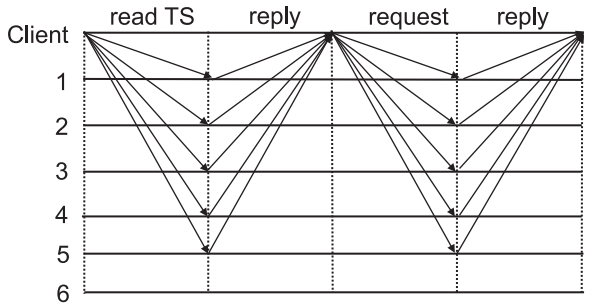

Fig. 9. Update operation in Q/U in the absence of conflicting requests ($f = 1, n = 6$).

servers to reconfigure the system and evict nonresponsive servers from the replier quorum. The *agree* and *commit* phases are used to establish an order for the client request and agree on a new replier quorum. Servers in the new replier quorum reply back to the client.

5.3. Clients as Proposers and Repairers

Query/Update (Q/U) [Abd-El-Malek et al. 2005] is a client-side BFT protocol that serializes query and update operations on data objects. Clients act as *proposers* and *repairers*. In Q/U each request is processed by only a subset of servers and server-to-server interaction is completely avoided. This makes the protocol efficient and scalable over WANs, at the cost of $5f + 1$ servers in the system. Clients and servers may fail arbitrarily. Moreover, each message exchanged by clients and servers carries a vector of authenticators.

Figure 9 shows the update operation during the normal case, that is, when client updates do not conflict. The client obtains a time stamp from a quorum of $4f+1$ servers and sends a request operation to a quorum of servers with the obtained time stamp.

The protocol can be further optimized by requiring clients to locally store the latest copy of the object they update, such that they do not need to read the time stamp from a quorum of servers in successive updates. This way, the update operation completes in only two one-way message rounds.

In the presence of concurrent update operations, correct servers may execute those operations in a different order. As such, a correct client *repairs* the system bringing correct servers in a consistent state. First, the client forces correct servers to suppress conflicting operations; then, the client imposes the same object versions onto correct servers.

5.4. Possible Attacks on Client-Side Protocols

Performance Attacks. Shifting part of the job onto the client side reduces both the number of protocol rounds and the server-to-server interaction, making the protocol more scalable on WANs. However, a malicious client may slow down the progress of the system by continuously triggering repair phases, which are typically more expensive than normal-case execution. For example, this happens when a malicious client fails to contact a full quorum, forcing a correct client to repair the system at a later time. In Zyzzyva and Scrooge, normal-case executions complete in three one-way message rounds. Malicious clients may force additional rounds (i.e., five rounds in Zyzzyva and seven rounds in Scrooge; see Figures 6 and 8, respectively) by dropping correct servers replies. In Zyzzyva this problem arises also when a malicious server delays the delivery of messages to a (possibly correct) client. Zyzzyva5 [Kotla et al. 2007] solves

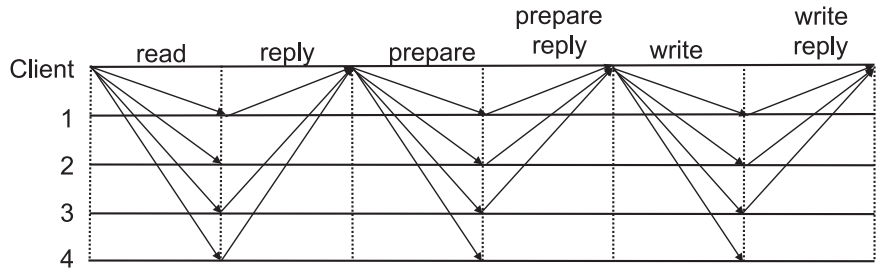

Fig. 10. Preventing inconsistencies from malicious clients in multiwriter, multireader applications. The proposed solution [Liskov and Rodrigues 2006] requires $n = 3f + 1$ servers (in this figure, $f = 1$, $n = 4$).

this problem by adding $2f$ servers to the system ($5f + 1$ servers in total). This way, agreement can always be reached in three one-way message rounds.[2]

Correctness Attacks. The lack of server-to-server communication in client-side protocols does not allow correct servers to detect inconsistent requests from malicious clients. A malicious client may send the same request with different sequence numbers to different sets of servers, leaving the system in an inconsistent state. As such, the *update* phase of client-side protocols should be designed to cope with malicious clients. As an example, PROPOSE and ECHO messages in the protocol for emulation of consensus objects (Figure 4) guarantee that a request associated with a time stamp is unique. This solution is suitable for single-writer, multireader protocols. A solution has been also proposed for multiwriter, multireader protocols [Liskov and Rodrigues 2006], as shown in Figure 10.

The client reads the most recently written value. Then, it sends a signed PREPARE message to all servers with the value to write. A correct server replies with a PREPARE-REPLY message if the received PREPARE message is valid (i.e., if the authentication succeeds). The writer collects $2f + 1$ PREPARE-REPLY messages and sends a signed WRITE message with the value to write (the same proposed in the previous PREPARE message). A correct server replies with a WRITE-REPLY message if the received WRITE message is valid. Finally, the writer waits for $2f + 1$ WRITE-REPLY messages before terminating the write operation.

6. HOW TO LABEL "HYBRID" PROTOCOLS

In the previous sections, we introduced protocols that follow the server-side or the client-side model. In this section, we describe a class of protocols that follow both approaches at the same time, trying to get the best of both worlds. They follow the client-side model in the absence of link/client/server failures and in the absence of concurrent requests to speed up normal-case executions, but they switch to the server-side model to guarantee safety and liveness in the presence of failures or conflicts.

In the remainder of this section, we first describe two protocols belonging to this class, namely, Aliph [Guerraoui et al. 2010] and HQ [Cowling et al. 2006], and then we motivate why they should be labeled as client side.

BFT protocols can be implemented as a collection of separate and independent instances [Guerraoui et al. 2010]. The rationale behind this is that it is very difficult to implement a single BFT protocol that achieves good performance in every scenario. By having different instances that are optimized to work well under different conditions (e.g., network synchrony, system load, fault model), the BFT protocol can transition

[2]Adding $2f$ servers to the system makes Zyzzyva5 a pure server-side protocol because clients are no more involved in repair actions.

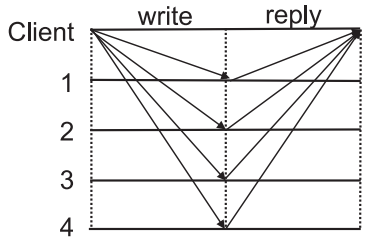

Fig. 11. *Quorum* instance in Aliph. It runs under ideal conditions (i.e., no failures and concurrent requests).

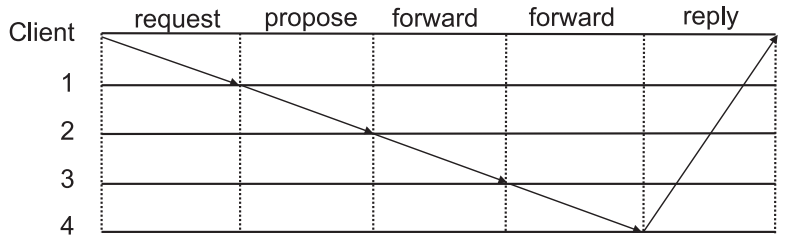

Fig. 12. *Chain* instance in Aliph. It runs in the absence of failure but in the presence of concurrent client requests.

from an instance to another in order to adapt to different scenarios, still maintaining good performance.

Aliph [Guerraoui et al. 2010] is a BFT protocol implemented using the approach described previously. It requires $3f + 1$ servers and is composed of three different instances:

—*Quorum*: which runs in the absence of link/client/server failures and concurrent client requests;
—*Chain*: which runs in the absence of link/client/server failures but in the presence of concurrent client requests; and
—*Backup*: which runs in the presence of failures and concurrent client requests.

The *Quorum* instance follows the client-side model, as represented in Figure 11. It runs under ideal conditions to achieve fast executions. The client acts as *proposer* and *repairer*. It first sends a request with a time stamp to all the servers, which speculatively execute the operation and reply to the client with a history of executed requests. In the absence of link/client/server failures and in the absence of concurrent client requests, *Quorum* requires only two one-way message transmissions.

If the histories provided by servers do not match, for example, because two concurrent operations have been executed in a different order by some servers, the client (as *repairer*) builds an abort history that allows servers to reconstruct a consistent order of operations. Then, the client switches to the next instance, that is, *Chain*, which is represented in Figure 12. *Chain* supports concurrent client requests by organizing servers as a pipeline. Each server executes a client operation and forwards that operation to the next server in the chain. The last server replies to the client.

Chain does not tolerate link or server failures. As an example, if the last server in the pipeline is malicious, it can fail to reply to the client. Hence, if the client does not receive a reply to its request or it receives an invalid reply, it switches to the *Backup* instance, which follows the server-side model by implementing PBFT. In this way, the protocol can tolerate up to f malicious servers and handle concurrent requests from different

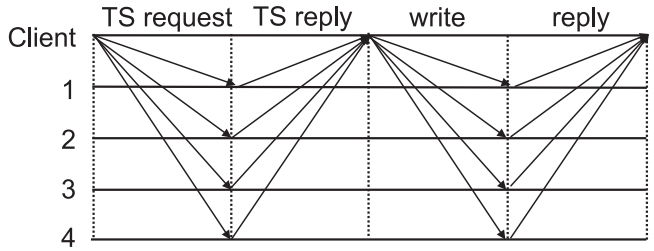

Fig. 13. Write operation in HQ. A client obtains a certificate for a new time stamp from the servers and writes a new value by attaching the received time stamp.

clients. *Backup* runs K PBFT ordering rounds before switching back to *Quorum*, with K being a parameter of the algorithm.

It is easy to see that a malicious client can launch the *performance* and *correctness* attacks described in Section 5. A malicious client can violate the consistency of the protocol by writing different values with the same time stamp to different sets of servers during the execution of a *Quorum* instance. The same attack can be used to slow down the progress of the system, by inducing servers to generate different histories so as to abort the *Quorum* instance and switch to *Chain*. During the *Chain* execution, the malicious client can discard replies sent by the last server in the pipeline, so as to trigger *Backup*, the most expensive instance among the three in terms of computation and message exchange.

Like Aliph, HQ [Cowling et al. 2006] follows the client-side model during normal-case executions and switches to the server-side model in the presence of failures or concurrent client requests. HQ requires $3f + 1$ servers in the system. The client acts as *proposer* and *repairer*. The write operation is shown in Figure 13 and is composed of two phases: (i) the client obtains a certificate containing the next time stamp to use from the servers; (ii) the client writes a new value by attaching the received time stamp and its certificate. In the presence of concurrency, the certificate provided by servers may not be valid. In this case, the client switches to PBFT. Since Aliph and HQ use PBFT as a base, their backup protocols are vulnerable to performance attacks as shown in Section 4.1.

As with Aliph, HQ is also vulnerable to malicious clients. A malicious client can slow down the progress of the system by continuously switching to PBFT. To do so, a malicious client may induce servers to create invalid time stamp certificates by sending different requests to different servers.

As we saw in this section, clients actively participate in Aliph and HQ's executions. According to the definitions of server-side and client-side models that we gave in Section 3, *we label these protocols as client side*. A system builder that wants to use them has to consider clients as part of the system and take additional security measures to cope with.

7. APPLICATIONS USING SERVER-SIDE OR CLIENT-SIDE MODEL

In this section, we present some BFT applications that use the server-side or client-side model. For each application we highlight the protocol configuration and the deployment strategies. For applications that have been implemented on both server- and client-side models, such as data storage systems, we highlight the trade-offs between the two approaches in terms of performance and system configuration. Finally, we conclude the section by presenting the differences between applications that use the server-side model and applications that use the client-side model.

Table I. Applications Using the Server-Side Model

Paper	Number of servers	Application	Deployment
Byzantine NFS [Castro and Liskov 2002]	$n = 3f + 1$	Distributed BFT filesystem	LAN
FARSITE [Adya et al. 2002]	$n = 3f + 1$	Distributed BFT filesystem	LAN
UpRight Hadoop [Clement et al. 2009a]	$n = 3f + 1$	Distributed BFT filesystem	LAN
UpRight Zookeeper [Clement et al. 2009a]	$n = 3f + 1$	Distributed BFT lock service	LAN
Privacy firewall [Yin et al. 2003]	$n = 3f + 1$	Distributed BFT firewall	LAN
CIS firewall [Sousa et al. 2010]	$n = 3f + 2k + 1$	Distributed BFT firewall	WAN-of-LANs
BFT firewall [Roeder and Schneider 2010]	$n = 3f + 2k + 1$	Distributed BFT firewall	LAN
Firewall for SIEMs [Garcia et al. 2013]	$n = 3f + 1$	Distributed BFT firewall	LAN
BFT SCADA [Zhao and Villaseca 2008]	$n = 3f + 1$	Intrusion-tolerant SCADA	LAN
Survivable SCADA [Kirsch et al. 2014]	$n = 3f + 1$	Intrusion-tolerant SCADA	LAN

7.1. Server-Side Applications

Table I reports some applications that use the server-side BFT model.

The server-side model has been adopted to build applications like distributed file systems [Castro and Liskov 2002; Adya et al. 2002; Clement et al. 2009a], distributed locking services [Clement et al. 2009a], distributed firewalls [Sousa et al. 2010; Roeder and Schneider 2010; Garcia et al. 2013], and intrusion-tolerant SCADA systems [Zhao and Villaseca 2008; Kirsch et al. 2014].

The Network File System (NFS) has been replicated using PBFT to make it intrusion tolerant [Castro and Liskov 2002]. The NFS is composed of $3f + 1$ servers, while clients use a BFT client library to communicate with the file system. The BFT protocol is used to serialize the access of multiple clients and operations to files.

FARSITE [Adya et al. 2002] is a replicated file system that achieves reliability and data integrity through BFT. The assumed workload consists of high access locality, low update rate, and read/write operations that are typically sequential and rarely concurrent. The BFT protocol is used to consistently replicate files across the system. A local cache on the client side is used to improve the performance of read operations.

Distributed BFT firewalls aim to protect a network from two different attacks: (i) external attackers that try to inject malicious packets into the network; and (ii) an internal attacker that tries to manipulate the firewall filtering rules. The OpenBSD packet filter, *pf*, has been replicated to tolerate malicious intrusions [Roeder and Schneider 2010]. The firewall uses PBFT as a replication engine, proactive recovery to increase the resiliency of the system, and leader rotation [Dwork et al. 1988] to reduce the delay that can be introduced by malicious leaders. In this work, leader rotation is enhanced with leader adjustment: correct servers skip over rejuvenating servers in leader rotation, so as to elect as leader only the servers that are currently in the system.

The hybrid wormhole-payload approach [Sousa et al. 2010] described in Section 4 has been used to build a firewall-like protection service for Critical Infrastructures called CIS. CIS follows a hybrid deployment approach. BFT firewalls in different local clusters are connected through wide-area links to build a protection mechanism that spans multiple geographic locations. Each payload intercepts all packets destined to the Local-Area Network (LAN) in order to verify if they respect some specified security policies. Every accepted packet is signed by the local wormhole. Wormholes are trusted components connected through a private network. They implement a voting mechanism to approve messages before forwarding them to the application.

A BFT firewall has also been built for protecting Security Information and Event Management (SIEM) systems [Garcia et al. 2013]. This firewall has two filtering components: (i) prefilters, which perform basic filtering actions and use Byzantine total order multicast [Sousa and Bessani 2012] to forward accepted messages to the filters; and (ii) filters, which implement a state machine replication service to filter messages based on application knowledge. Each filter applies exactly the same rules to the messages received from prefilters.

Unlike the firewalls described previously, which aim to prevent the injection of malicious messages and the manipulation of filtering rules, a BFT firewall has also been used to protect the confidentiality of information [Yin et al. 2003]. Typically, clients of a server-side BFT protocol implement a voting scheme that selects $f + 1$ matching replies from different replicas before executing an operation. However, this does not protect against a malicious server that leaks sensitive information from the system and sends it to a malicious client as a reply message. The main innovative aspect of this work is that agreement is separated from execution. A set of $3f + 1$ servers form a BFT agreement engine that can be shared across multiple applications. The protocol used is BASE [Rodrigues et al. 2001], a library that integrates PBFT with data abstraction techniques to mask potential software errors. Clients communicate only with the servers of the agreement engine. A set of $2g + 1$ servers, where g is the maximum number of Byzantine servers, form the execution engine, which is specific to the application they implement. Note that, because servers in the execution engine do not run any agreement protocol, the cost of replication is reduced. Servers in the execution engine are the only ones to apply changes to the application state. The privacy firewall lies in between the agreement engine and the execution engine. It is composed of $h + 1$ rows with $h + 1$ filters each, where h is the maximum number of Byzantine filters. Filters in row i, with $2 \leq i \leq h$, communicate with filters in row $i - 1$. Filters in row 1 communicate directly with servers in the agreement engine, while filters in row $h + 1$ receive replies from the servers in the execution engine. This structure guarantees that there is at least one path composed of only correct filters that connects the execution engine to the agreement engine. Filters in the privacy firewall use threshold cryptography to build a certificate for each correct reply and forward it to the agreement engine (the reply will eventually reach the client). Incorrect replies that may contain sensitive information are discarded along the path by correct filters.

The UpRight Cluster Service [Clement et al. 2009a] also benefits from the separation between agreement and execution. The UpRight agreement module implements a fast ordering protocol similar to Zyzzyva and ensures robust performance even in the presence of failures by using client request validation, resource scheduling, and request filtering as in Aardvark. Unlike Zyzzyva, where clients drive system reconfiguration, in UpRight possible conflicts that may happen during the agreement phase are solved by agreement servers when they checkpoint their states.

UpRight is used to implement a Zookeeper-like [Hunt et al. 2010] locking service and a Hadoop-like [Shvachko et al. 2010] BFT distributed file system. UpRight-Zookeeper replaces the Paxos-like [Lamport 2001] replication protocol in the original Zookeeper with the UpRight replication protocol described previously. Other modifications include checkpointing the application state in a deterministic manner across system servers using copy-on-write techniques. UpRight-HDFS enhances the original Hadoop File System (HDFS) by supporting redundant NameNodes (a NameNode is the entity that stores <*filename*, *block ID*> mappings), eliminating single points of failures. UpRight-HDFS is also resilient to faulty clients, NameNodes, and DataNodes (i.e., the entities that store data blocks).

PBFT and Prime have been used to build intrusion-tolerant SCADA systems [Zhao and Villaseca 2008; Kirsch et al. 2014]. SCADA stands for Supervisory Control And

Table II. Applications Using the Client-Side Model

Paper	Number of servers	Application	Deployment
PASIS [Goodson et al. 2004]	$n = 4f + 1$	Key-value datastore	LAN
Loft [Hendricks et al. 2007]	$n = 3f + 1$	Key-value datastore	LAN
BFT K/V store [Roeder and Schneider 2010]	$n = 3f + 1$	Key-value datastore	LAN
Depsky [Bessani et al. 2013]	$n = 3f + 1$	Key-value datastore	WAN
PoWerStore [Dobre et al. 2013]	$n = 3f + 1$	Key-value datastore	LAN, WAN
Phalanx [Malkhi and Reiter 1998b]	$n = 4f + 1$	Key management/CA system	LAN
COCA [Zhou et al. 2002]	$n = 3f + 1$	Key management/CA system	WAN

Data Acquisition, and is often used for monitoring critical infrastructures. The most important element is the master server, which collects data about the status of the infrastructure from remote equipment, processes that data, and provides the results to a human-machine interface for visualization. BFT is used to replicate the master server in an intrusion-tolerant manner. Replicas of the master server use the BFT protocol to order messages coming from remote equipment.

All these applications are deployed on LAN. The UpRight Cluster Services [Clement et al. 2009a] and the survivable SCADA system [Kirsch et al. 2014] benefit from Aardvark and Prime, respectively, to guarantee high throughput or low latency even while the system is under attack. Recovery mechanisms have been used to increase the resiliency of BFT firewalls [Sousa et al. 2010; Roeder and Schneider 2010], at the cost of $2k$ additional servers in the system ($3f + 2k + 1$ servers in total).

7.2. Client-Side Applications

Table II reports some applications that use the client-side BFT model.

The client-side model has been used to implement applications like key-value datastore systems [Hendricks et al. 2007; Roeder and Schneider 2010; Bessani et al. 2013; Dobre et al. 2013] and key management/certificate authority systems [Malkhi and Reiter 1998b; Zhou et al. 2002].

Client-side applications shift part of the job to the clients for performance optimization, even if this may require additional servers in the system and specific protocols to cope with malicious clients, as described in Section 5. As an example, PASIS [Goodson et al. 2004] implements a BFT key-value datastore in which the clients *propose* new updates and *repair* the system when they detect inconsistencies. PASIS combines erasure coding [Fragouli et al. 2006] and checksums to read and update fragments of data blocks from and to the storage system, which is composed of $4f + 1$ servers. Repair operations start when the fragments retrieved by a client are insufficient to recreate a correct object. The client has to retrieve additional fragments and initiate a correction process that ends when the object is completely repaired. The benefit of having clients involved in the computation emerges when PASIS is compared with a data storage system based on PBFT: PASIS shows higher throughput and lower latency as the number of faulty servers (i.e., the value of f) increases [Goodson et al. 2004], at the cost of f additional servers in the system.

As with PASIS, Loft [Hendricks et al. 2007] also implements an erasure-encoded Byzantine data storage. Loft supports multiwriter operations and defends against malicious clients by following the same approach we illustrated in Figure 10, Section 5 [Liskov and Rodrigues 2006]. Loft is optimized for normal-case executions, in which the number of failures and concurrent operations is low. Under these assumptions, the update operation completes in two communication rounds and the read operation completes in one communication round. The main difference with PASIS is that the

Table III. Differences Between Server-Side and Client-Side Applications

Differences	Server-side applications	Client-side applications
Impact of malicious clients	DoS attacks	DoS and correctness attacks
Impact of malicious servers	DoS attacks	DoS attacks
Number of servers	$3f + 1$	$3f + 1$ or $4f + 1$
Protocol implementation	Server-to-server communication	Reduced server-to-server communication
Deployment strategy	LAN	LAN, WAN

object verification during a read operation is done locally by Loft servers, while in PASIS the same operation is coordinated by clients and requires a higher number of checksum computations.

Depsky [Bessani et al. 2013] is a set of protocols for data storage over multiple clouds. These clouds are connected together to form a *cloud of clouds* in order to improve availability, integrity, and confidentiality of the data. Availability is achieved by replicating data over distinct clouds; integrity is obtained by applying a BFT replication protocol within each cloud; confidentiality is achieved by combining secret sharing [Shamir 1979] with network coding [Fragouli et al. 2006], such that data is stored in encrypted format. Moreover, Depsky separates data and metadata. This means that a client has to perform two distinct system accesses to read/write data and metadata.

PoWerStore [Dobre et al. 2013] is a BFT data store that provides high availability and strong consistency without sacrificing efficiency. Efficiency is achieved by replacing signatures with computationally less expensive authenticators. System robustness is obtained by using *Proof of Writing* (PoW) [Dobre et al. 2013], which is a two-round procedure in which the second round is used to prove to a client that the first round of the write operation has actually been completed. During the first round a client writes data, while in the second round it writes metadata.

Phalanx [Malkhi and Reiter 1998b] is a BFT distributed coordination system built on top of the consensus object emulation protocol [Malkhi and Reiter 1998c] described in Section 5. Phalanx has been used as a distributed voting system. For this reason, its baseline protocol [Malkhi and Reiter 1998c] has been extended to support *atomic* operations [Lamport 1986]: reads and writes are linearizable [Herlihy and Wing 1990]. To this end, reading an object x from a quorum Q_1 requires a write-back mechanism to ensure that the value v of x with the highest time stamp t is stored at some quorum Q_2.

COCA [Zhou et al. 2002] is a BFT certificate authority system. It is the first to integrate threshold cryptography and proactive recovery in client-side systems. This approach is extended with proactive obfuscation [Roeder and Schneider 2010] to build a key-value datastore in which servers are less likely to have shared vulnerabilities. A client writes a new certificate to a quorum of COCA servers. These servers use threshold cryptography to sign replies to clients retrieving certificates. COCA uses periodic proactive recovery at two distinct levels: (i) to refresh the shares of the key used to sign replies to clients; and (ii) to rejuvenate servers from a clean execution environment.

Unlike the server-side applications we analyzed, which are deployed on LAN, some client-side applications described previously, such as Depsky, COCA, and PoWerStore are deployed on WAN. The client-side model, in fact, reduces the server-to-server interactions, such that client-side applications scale better than server-side applications on WAN.

7.3. Differences Between Server-Side and Client-Side Applications

Table III highlights the differences between applications that use the server-side model and applications that use the client-side model in terms of *impact of malicious clients,*

impact of malicious servers, number of servers, protocol implementation, and *deployment strategy*.

Impact of Malicious Clients

Server Side: In the server-side model, malicious clients cannot violate safety. Correct clients, in fact, observe the operations executed by malicious clients in a consistent order [Castro and Liskov 1999]. Malicious clients, however, can slow down the system execution by launching performance attacks (e.g., by sending corrupted MACs to the leader [Castro and Liskov 1999] or by sending too many requests to the system [Clement et al. 2009b]).

Client Side: In the client-side model, malicious clients that act as *proposers* may leave the system in an inconsistent state by sending different updates to different sets of servers. Solutions to this problem require additional message rounds to guarantee that the time stamp associated with an update is unique [Malkhi and Reiter 1998c; Liskov and Rodrigues 2006]. Client-side protocols are also subject to performance attacks launched by faulty clients. A malicious client that acts as *proposer* may fail to contact a full quorum of servers, forcing some correct client to repair the system at a later time. A malicious client that acts as *repairer* can also launch performance attacks, by triggering continuous reconfigurations [Kotla et al. 2007] or introducing malicious servers in the preferred replier set [Serafini et al. 2010].

Impact of Malicious Servers

Server Side: In the server-side model, if the number of malicious servers is no higher than f, malicious servers cannot violate safety. However, malicious servers can slow down the progress of the protocol by launching performance attacks like the *Timeout Manipulation* and *Pre-Prepare Delay* attacks described in Section 4. Some of the server-side BFT protocols [Amir et al. 2008; Clement et al. 2009b] take countermeasures to avoid performance degradation while under attack.

Client Side: In the client-side model, if the number of malicious servers is no higher than f, malicious servers cannot violate safety. Malicious servers can still launch performance attacks. As an example, a malicious server can hold up-to-date objects and provide stale objects to clients. However, because clients receive enough replies, they can always obtain the most up-to-date values. Moreover, a malicious server may delay the delivery of messages to clients, forcing correct clients to eventually repair the system [Kotla et al. 2007; Serafini et al. 2010].

Number of Servers

Server Side: In the server-side applications we described, the number of servers is $3f + 1$, which provides optimal resiliency. $3f + 2k + 1$ servers are used in the presence of recovery operations [Sousa et al. 2010; Roeder and Schneider 2010].

Client Side: In the client-side applications we described the number of servers in the system is either $3f + 1$ or $4f + 1$, depending on the specific protocol. Some [Roeder and Schneider 2010; Zhou et al. 2002] achieve optimal resiliency (i.e., $3f + 1$ servers), while others [Goodson et al. 2004; Malkhi and Reiter 1998b] use $4f + 1$ servers to improve the performance of the system by completely removing server-to-server communication.

Protocol Implementation

Server Side: Server-side systems run an agreement protocol based on server-to-server broadcast communications. Additional subprotocols are implemented to guarantee performance under attack [Kirsch et al. 2014], improve resiliency through proactive recovery [Sousa et al. 2010; Roeder and Schneider 2010], and deal with pull-based client request mechanisms [Zhao and Villaseca 2008; Kirsch et al. 2014].

Client Side: Unlike server-side protocols, in client-side protocols clients participate actively. They can act either as *proposers*, *repairers*, or both. Server-to-server or

client-to-client communication is typically avoided, sometimes at the cost of having more servers in the system [Malkhi and Reiter 1998b; Goodson et al. 2004].

Deployment Strategy

Server Side: The server-side systems we analyzed are deployed on LAN. PBFT achieves good performance under normal operations, while Prime and Aardvark achieve good performance even while under attack.

Client Side: Some client-side applications, such as PoWerStore, Depsky, and COCA, are deployed on WAN. Because server-to-server interaction is reduced, the impact of wide-area links on the performance is lower, compared with server-side systems deployed on WAN.

8. SERVER-SIDE VERSUS CLIENT-SIDE: TRADE-OFFS

In this section, we describe some trade-offs that the analyzed BFT protocols present. In Sections 8.1 and 8.2, we describe how an existing reference server-side or client-side application chose these trade-offs, while in Section 8.3 we discuss how other existing server-side or client-side applications chose the trade-offs.

Trade-off 1: Optimal resiliency versus reducing the number of communication rounds

Prior work [Pease et al. 1980] proved that the BFT problem is solvable if the number of processors n is greater than or equal to $3f+1$. $n = 3f+1$ represents optimal resiliency, that is, solving the BFT problem with the minimum number of processors required. Some BFT protocols described in the previous sections trade optimal resiliency for performance: they use more servers (e.g., $n \geq 4f+1$, $n \geq 5f+1$) to reduce the number of communication rounds of the BFT protocol. This reduces the delay before the client obtains a reply, especially in the presence of high-latency links.

Trade-off 2: Asymmetric authentication versus symmetric authentication

Asymmetric authentication is obtained through digital signatures. Clients and servers have a private key that they use to sign messages. Digital signatures guarantee (with high probability) that signed messages are unforgeable. However, digital signatures represent the main performance bottleneck for many BFT protocols, such as PBFT. As an alternative, symmetric authentication can be obtained through MACs. Each server shares a key with every other server. A message signed by a server carries a vector of MACs, one for each recipient server. Moreover, a server also shares a key with each client. MACs are computationally less expensive than signatures, but they are not able to prove to a third party that a message is authentic. PBFT uses asymmetric authentication, but also proposes a variant that uses symmetric authentication. In this variant, digital signatures are still used to sign view-change messages, while MACs are used to authenticate all other messages. While this increases the performance of the BFT protocol under normal operations, a malicious client can still force view changes continuously by sending messages with malformed MACs in order to slow down the progress of the system. This problem can be avoided by requiring clients to sign the messages they send [Garcia et al. 2013].

Trade-off 3: Strong synchronization versus weak synchronization

BFT protocols that follow the server-side model impose strong synchronization among correct servers: a correct server moves forward only when it collects $2f+1$ messages that say the same thing. On the contrary, BFT protocols that follow the client-side model reduce (and, in some cases, remove completely) synchronization among servers. Server-to-server interaction can be used to propagate updates in the presence of malicious clients [Malkhi and Reiter 1998a] or for system reconfiguration [Serafini et al. 2010]. This approach scales better than the server-side model on WAN (a system that implements the server-side model makes progress as fast as the $f+1^{th}$ slowest server/link).

Trade-off 4: Performance guarantees under attack versus reducing protocol overhead/system configuration
Performance guarantees under attack are desirable, especially for critical applications [Kirsch et al. 2014], but they add some overhead to the BFT protocol or require specific system configurations. Prime [Amir et al. 2008] extends PBFT with subprotocols that monitor the behavior of the leader. These subprotocols introduce some overhead to the computation but they are necessary to evict a leader that is performing too slowly and to settle on a correct and fast leader that guarantees that clients' operations are executed within a bounded delay. Aardvark [Clement et al. 2009b] ensures constant throughput even while under attack. To do so, distinct network interface controllers and wires are used to connect each pair of servers. The system size is limited by the number of network interface controllers available and the addition/removal of a server requires manual reconfiguration.

Trade-off 5: Proactive recovery: Correctness versus correctness *and* continuous availability
Proactive recovery [Roeder and Schneider 2010] is a technique that periodically rejuvenates servers from a clean execution environment in order to increase the resiliency of the system over its lifetime. However, while this technique guarantees system correctness, it does not necessarily guarantee continuous availability. If a correct server rejuvenates and another f servers fails concurrently, the system may halt until the correct server completes rejuvenation. Continuous availability in the presence of rejuvenations comes at the cost of $2k$ additional servers in the system [Sousa et al. 2010; Platania et al. 2014].

8.1. How a Server-Side Application Chose Trade-offs

We now select one of the server-side applications presented in Section 7.1 and describe how this application chose the trade-offs discussed previously. The server-side application we select is the survivable SCADA system [Kirsch et al. 2014], which was integrated into the Siemens corporation commercial SCADA product for the power grid.

The survivable SCADA system uses the Prime BFT protocol to replicate the master server, which is the most important element of the system. The replicated master is connected to Remote Terminal Units (RTUs), which are the clients of the system. Periodically, the replicated master pulls information about remote equipment from the RTUs. Replicas of the master server use Prime to agree on the order in which this information has to be processed, in order to assess the status of the power grid.

Optimal Resiliency versus Reducing the Number of Communication Rounds. The survivable SCADA system chose optimal resiliency ($n = 3f + 1$). The algorithm is composed of a *preordering phase*, in which the servers exchange preliminary information about client updates, and an *ordering phase*, which implements the *pre-prepare, prepare, commit* pattern of PBFT. No optimization is used to reduce the number of communication rounds.

Asymmetric Authentication versus Symmetric Authentication. The SCADA system uses asymmetric authentication. All the messages exchanged among replicas of the master server and between clients (i.e., RTUs) and the replicated master are digitally signed.

Strong Synchronization versus Weak Synchronization. Prime implements state machine replication and imposes strong synchronization among correct servers. The replicated SCADA master is deployed on LAN, while RTUs can be deployed on different geographic locations and can be connected to the replicated master through WAN links.

Performance Guarantees under Attack versus Reducing protocol Overhead / System Configuration. Prime servers run a background protocol that monitors the activity of the leader and a preordering phase to discover a malicious leader that may drop client updates. When compared with PBFT, these subprotocols introduce some overhead in the computation during normal operations. However, the two subprotocols are necessary to limit the performance degradation that a malicious leader may cause. This makes Prime more than one order of magnitude faster than PBFT while under attack [Amir et al. 2008]. Guaranteeing good performance even in the presence of malicious attacks is a desirable property of a SCADA system.

Proactive Recovery: Correctness versus Correctness and Continuous Availability. The Prime protocol used to replicate the SCADA master [Kirsch et al. 2014] is an extension of Prime 1.0 [Amir et al. 2008]. Prime 1.0 does not use any kind of recovery mechanism. Prime 2.0 [Amir et al. 2014], a later iteration, supports proactive recovery and state transfer [Platania et al. 2014]. A SCADA system built on top of Prime 2.0 could achieve correctness and continuous availability at the cost of $2k$ additional servers.

The survivable SCADA system is strongly oriented to security and performance under attack, which are desirable properties of critical infrastructures like power grids. Regarding security, clients and servers sign all the messages they send. Moreover, Prime uses threshold cryptography [Shamir 1979; Blakley 1979] during the view change algorithm to unambiguously define a list of servers that should be contacted by all other servers to collect the Prime ephemeral state (e.g., the committed client updates). Regarding performance under attack, Prime introduces subprotocols that discover and replace a slow (and potentially malicious) leader. To achieve strong security and performance guarantees under attack, Prime gives up performance improvements under normal executions: Prime does not provide any mechanism to speed up the system in the normal case (e.g., no fast agreement, use of authenticators, loose synchronization among servers).

8.2. How a Client-Side Application Chose Trade-offs

The client-side application we select is COCA [Zhou et al. 2002], an online certification authority developed at Cornell University. The reason we choose COCA is that COCA addresses all the trade-offs described in this section, including the implementation of recovery techniques. This gives us the opportunity to describe practical aspects related to recovery mechanisms in real-world applications. COCA issues digitally signed certificates to associate a name (e.g., website) with a public key, and provides a means for clients to validate certificates.

COCA clients use an *Update* protocol similar to the write operation depicted in Figure 13 to store/update a certificate. Moreover, COCA uses a threshold cryptography mechanism to sign replies to clients and generate proofs of new certificates to servers in a quorum. COCA combines a client-side protocol, secret sharing, and proactive recovery. Periodically, servers are rejuvenated from a clean execution environment and the shares of the service's private key are refreshed.

Optimal Resiliency versus Reducing the Number of Communication Rounds. COCA chose optimal resiliency ($n = 3f + 1$). A client sends a request to $f + 1$ *delegate* servers. Each delegate manages a quorum of $2f + 1$ servers to execute that request and build a reply message. In addition, each delegate server runs a threshold signature protocol to sign the reply message to the client in cooperation with another f server.

Asymmetric Authentication versus Symmetric Authentication. COCA uses asymmetric authentication. Clients and servers sign all messages they send. As described

previously, threshold cryptography is used to sign replies to clients and generate proofs of new certificates to servers in a quorum.

Strong Synchronization versus Weak Synchronization. COCA reduces synchronization among servers and is suitable for deployment on LAN and WAN. However, some correct servers may store stale certificates because client requests are executed only by quorums of servers. Finally, it is important to note that, despite loose synchronization introduced by the BFT protocol, COCA uses threshold cryptography. The experimental evaluation shows that the threshold cryptography protocol has the largest impact on the performance of the system (on WAN it is second only to network delays) [Zhou et al. 2002].

Performance Guarantees under Attack versus Reducing Protocol Overhead/System Configuration. COCA does not explicitly define performance guarantees under attack, but part of the system is devoted to protecting against DoS attacks that may dramatically slow down the progress of the system. Most of the protocol's cost and complexity is concerned with defending against these attacks.

Proactive Recovery: Correctness versus Correctness and Continuous Availability. COCA uses two distinct forms of recovery to increase the resiliency of the system:

—Proactive secret sharing recovery: periodically, servers run an update protocol to refresh the shares of the service's private key used to generate partial signatures.
—Proactive server recovery: periodically, one server at a time is rejuvenated to clean that machine from potential intrusions.

COCA does not uses $2k$ additional servers in the presence of proactive recovery. This means that COCA is correct but not necessarily always available during its execution. A quorum in COCA is composed of $2f+1$ servers. If f servers are malicious and noncooperative while a correct server rejuvenates, the system will halt until the rejuvenated server completes recovery operations.

Because of the design choices made, COCA is a system that offers optimal resiliency and scales well on LAN and WAN. The use of digital signatures regulates the access of clients to the system and prevents malicious servers from fabricating certificates. In addition, proactive secret sharing recovery and proactive server recovery improve the resiliency of the system over its lifetime. COCA can tolerate any number of malicious servers, provided that no more than f failures occur within a *window of vulnerability* [Zhou et al. 2002]:

> ''Each window of vulnerability at a COCA server begins when that server starts executing the proactive recovery protocols and terminates when that server has again started and finished those protocols.''

8.3. How Other Applications Chose Trade-offs

Table IV shows how other server-side and client-side applications presented in Section 7 chose trade-offs. Each row of the table includes only the systems for which that specific trade-off is meaningful.

All server-side applications chose optimal resiliency over reducing the number of communication rounds. Applications that use recovery techniques [Sousa et al. 2010; Roeder and Schneider 2010] use $3f+2k+1$ servers (see later). Although the client-side model presents many optimizations to reduce the number of communication rounds, only PASIS [Goodson et al. 2004] and Phalanx [Malkhi and Reiter 1998b] give up optimal resiliency for better performance. In contrast to other client-side applications, which use $3f+1$ servers, PASIS and Phalanx use $4f+1$ servers and eliminate any server-to-server interaction. As an example, the PASIS experimental evaluation shows

Table IV. Trade-offs Chosen by Other Server-Side or Client-Side Applications

Trade-off 1	**Server-side systems**	**Client-side systems**
Optimal resiliency	Byzantine NFS [Castro and Liskov 2002], FARSITE [Adya et al. 2002], BFT SCADA [Zhao and Villaseca 2008], UpRight [Clement et al. 2009a], Firewall for SIEMs [Garcia et al. 2013], Survivable SCADA [Kirsch et al. 2014]	COCA [Zhou et al. 2002], Loft [Hendricks et al. 2007], BFT K/V store [Roeder and Schneider 2010], Depsky [Bessani et al. 2013], PoWerStore [Dobre et al. 2013]
Reducing the number of communication	None	Phalanx [Malkhi and Reiter 1998b], PASIS [Goodson et al. 2004]
Trade-off 2	**Server-side systems**	**Client-side systems**
Asymmetric authentication	FARSITE [Adya et al. 2002], BFT firewall [Roeder and Schneider 2010], Firewall for SIEMs [Garcia et al. 2013], Survivable SCADA [Kirsch et al. 2014]	Phalanx [Malkhi and Reiter 1998b], COCA [Zhou et al. 2002], BFT K/V store [Roeder and Schneider 2010], Depsky [Bessani et al. 2013]
Symmetric authentication	Byzantine NFS [Castro and Liskov 2002], BFT SCADA [Zhao and Villaseca 2008], UpRight [Clement et al. 2009a], CIS firewall [Sousa et al. 2010], Firewall for SIEMs [Garcia et al. 2013]	PASIS [Goodson et al. 2004], Loft [Hendricks et al. 2007], PoWerStore [Dobre et al. 2013]
Trade-off 3	**Server-side systems**	**Client-side systems**
Strong synchronization	Byzantine NFS [Castro and Liskov 2002], FARSITE [Adya et al. 2002], BFT SCADA [Zhao and Villaseca 2008], UpRight [Clement et al. 2009a], CIS firewall [Sousa et al. 2010], BFT firewall [Roeder and Schneider 2010], Firewall for SIEMs [Garcia et al. 2013], Survivable SCADA [Kirsch et al. 2014]	None
Weak synchronization	None	Phalanx [Malkhi and Reiter 1998b], COCA [Zhou et al. 2002], PASIS [Goodson et al. 2004], Loft [Hendricks et al. 2007], BFT K/V store [Roeder and Schneider 2010], Depsky [Bessani et al. 2013], PoWerStore [Dobre et al. 2013]
Trade-off 4	**Server-side systems**	**Client-side systems**
Performance guarantees under attack	UpRight [Clement et al. 2009a], Survivable SCADA [Kirsch et al. 2014]	None
Reducing protocol overhead/system configuration	Byzantine NFS [Castro and Liskov 2002], FARSITE [Adya et al. 2002], BFT SCADA [Zhao and Villaseca 2008], CIS firewall [Sousa et al. 2010], Firewall for SIEMs [Garcia et al. 2013]	Phalanx [Malkhi and Reiter 1998b], COCA [Zhou et al. 2002], PASIS [Goodson et al. 2004], Loft [Hendricks et al. 2007], BFT firewall [Roeder and Schneider 2010], BFT K/V store [Roeder and Schneider 2010], Depsky [Bessani et al. 2013], PoWerStore [Dobre et al. 2013]
Trade-off 5	**Server-side systems**	**Client-side systems**
Proactive recovery: *Only* correctness	None	COCA [Zhou et al. 2002]
Proactive recovery: Correctness *and* continuous availability	CIS firewall [Sousa et al. 2010], BFT firewall [Roeder and Schneider 2010]	None

how this choice allows PASIS to outperform PBFT in terms of latency and throughput [Goodson et al. 2004].

Regarding authentication, the applications we consider are split almost evenly between using digital signatures (asymmetric authentication) and MACs (symmetric authentication). Interestingly, this choice does not depend on the adopted system model (server side or client side). Applications that use MACs to authenticate messages seek to maximize performance during normal operations. However, some of these applications still use signatures for some *critical* message. As an example, the Byzantine NFS [Castro and Liskov 2002] uses private/public key cryptography to exchange the symmetric keys that are used to authenticate all other messages. The firewall for SIEMs [Garcia et al. 2013] uses both signatures and authenticators. The system is composed of *prefilters* and *filters*. Each message is signed with the private key of the sender. In addition, the sender attaches a vector of MACs to each message, which is computed over the message payload and signature. MACs represent an optimization to speed up the message verification. Prefilters drop messages with invalid MACs, while all other messages are forwarded to filters through a total order multicast channel. Filters check the authenticity of the received messages by verifying the signature of the sender. This second step is necessary because a malicious prefilter could alter the MAC vector computed by the original sender or invent new messages.

In contrast, the choice between strong and weak synchronization is strongly tied to the adopted system model. Server-side applications rely on server-to-server broadcast communications and require each (correct) server to process each request. The client-side applications we described in Section 7 rely on the clients to coordinate the ordering round of the BFT protocol. The lack of server-to-server interaction makes these applications scale better than client-side systems on WAN. The UpRight services deserve particular mention: although they rely on fast ordering and separate agreement from execution [Yin et al. 2003], correct execution servers still have to receive ordered requests from a quorum of order servers. Moreover, possible conflicts are resolved during the state checkpointing process, which requires additional communication between execution and order servers.

Performance guarantees under attack are an important requirement for practical applications. However, only the survivable SCADA system [Kirsch et al. 2014] and the UpRight versions of Zookeeper and the Hadoop file system [Clement et al. 2009a] offer these guarantees. At the heart of these solutions lie Prime [Amir et al. 2008] and Aardvark [Clement et al. 2009b], respectively. Prime extends the PBFT protocol with subprotocols that monitor the activity of the leader to guarantee performance in terms of latency. Aardvark uses resource isolation, regular view changes, and a prefiltering mechanism to ensure constant throughput. Performance guarantees under attack are obtained at the cost of higher overhead during normal operations when compared with other approaches. The majority of the BFT applications we discussed throughout the article are not willing to pay this cost. They focus instead on how to improve performance in the normal case.

Finally, the ability to automatically recover system servers is another key aspect for the implementation of practical, real-world applications. As with performance under attack, only a few systems use recovery mechanisms, namely, COCA [Zhou et al. 2002], CIS firewall [Sousa et al. 2010], and BFT firewall [Roeder and Schneider 2010]. Surprisingly, none of the applications that provide performance guarantees under attack implement recovery strategies and, vice versa, applications that implement recovery mechanisms do not provide any kind of performance guarantee under attack. CIS and BFT firewalls implement proactive recovery algorithms and use $3f+2k+1$ servers in order to guarantee correctness *and* continuous availability even during the rejuvenation of correct servers. The CIS firewall also implements a reactive recovery mechanism that

triggers the rejuvenation of a compromised server when a failure of that server is detected. On the contrary, COCA implements proactive recovery but uses only $3f + 1$ servers. Hence, COCA guarantees correctness but not necessarily continuous availability. The system halts if f out of $2f + 1$ servers are compromised and noncooperative, and a correct server is rejuvenating.

9. BUILDING A BFT APPLICATION: LESSONS LEARNED

In this section, we describe the main aspects that a system builder should consider for building a BFT application, based on the lessons learned in this article.

The first thing to consider is the system model, server side or client side. The choice depends strictly on the requirements of the application that the system builder wants to build. If the application needs a resilient agreement protocol to establish the order in which operations from multiple clients have to be executed, the server-side model is more appropriate. An example of such an application is the survivable SCADA system [Kirsch et al. 2014], which is integrated with a real Siemens SCADA product. Replicas of the SCADA master use Prime to order operations coming from RTUs. Prime implements state machine replication and guarantees that client operations are executed in the same order by all correct servers, and thus, that all correct servers are consistent. On the contrary, for applications in which a resource is updated by a single client or a few clients, the client-side model is more appropriate. An example of such an application is Phalanx [Malkhi and Reiter 1998b], which was part of a voting system for elections in Costa Rica. Phalanx implements a client-side protocol with single-writer, *write once* semantics in order to avoid multiple votes from the same person. Due to its favorable scalability properties, the client-side model is appealing for building geodistributed applications. However, unlike the server-side model, in which server-to-server communication helps detect conflicting updates from malicious clients, a system builder that wants to follow the client-side model should consider clients part of the system and provide the system itself with mechanisms that handle malicious clients.

Another important aspect to consider when building a BFT application is the behavior of that application under attack. Guaranteeing performance while the system is under malicious attack is a fundamental requirement of real-world applications. Currently, only a few solutions focus on this. Prime enforces latency guarantees by monitoring the activity of the leader, in order to replace a leader that performs too slowly and could potentially be malicious. Aardvark and COCA enforce constant throughput guarantees under attack through authorization mechanisms, resource management (i.e., different input queues to store messages coming from different clients and servers), and caching reply messages in order to avoid costly cryptographic operations in case of multiple requests of the same operation from malicious clients. While these protocols provide latency *or* throughput guarantees, the large gap in the current BFT literature is the absence of protocols able to provide latency *and* throughput guarantees while under attack. Combining solutions like Prime, Aardvark, and COCA, when possible, would allow a distributed system to tolerate a broader class of attacks, while still providing good performance even in the presence of a malicious adversary.

Another practical aspect to consider is how to make an application survivable over a long period of time. Modern systems are expected to operate for years (e.g., critical infrastructures). The vast majority of BFT protocols in the literature guarantee correct behavior if no more than f servers are compromised over the system lifetime but they do not adopt diversity or recovery mechanisms, which makes it hard to support the previous assumption in case of long-lived systems. In addition, current solutions either address performance guarantees under attack, or provide recovery mechanisms to increase the resiliency of the system. These aspects are considered together for the

first time in Prime 2.0 [Amir et al. 2014; Platania et al. 2014], which integrates the Prime replication engine, fine-grained software diversity [Cohen 1993; Giuffrida et al. 2012; Pappas et al. 2012] obtained at compile time [Homescu et al. 2013], and proactive recovery [Roeder and Schneider 2010]. Servers are periodically rejuvenated from a clean execution environment and application state. After each rejuvenation, a new variant of a Prime server is automatically generated with a MultiCompiler [Homescu et al. 2013] that guarantees, with high probability, that any two variants are different from each other. In this way, similar to COCA, a BFT protocol can tolerate any number of failures during the system lifetime, provided that no more than f failures occur within a vulnerability window. Guaranteeing continuous availability in the presence of recovery mechanisms requires $2k$ additional servers in the system [Sousa et al. 2010], where k is the number of servers that rejuvenate at the same time. Trading optimal resiliency for continuous availability is important for critical systems, in order to continue working even in the presence of f concurrent failures and the rejuvenation of k correct servers. Reducing the number of physical servers used in the system is a problem that can be addressed through modern virtualization technologies.

The last, but not least, aspect a system builder should consider for building a BFT application is the authentication mechanism, that is, digital signatures or MACs. In the previous section, we learned that BFT applications split almost evenly between these two approaches. MACs are computationally less expensive than signatures, which make them appealing for improving the performance of the system. However, this improvement is obtainable only during normal conditions: a malicious client can force view changes or system reconfigurations in order to slow down the progress of the protocol by sending correct authenticators to only a subset of servers. MACs do not protect against this kind of attack, nor do they provide nonrepudiation. To overcome these limitations, digital signatures are required. Improved performance, while still providing high security, can be achieved in two different ways:

—Developing flexible applications that can benefit from modern multicore technologies. In this way, several consensus instances can run in parallel [Behl et al. 2014] or one or more cores could be dedicated to specific subtasks [Schmidt and Suda 1993], such as cryptographic operations, as also suggested by previous work [Abd-El-Malek et al. 2005].
—Using a hybrid approach, similar to the PBFT variant in which MACs are used to authenticate server-to-server messages during normal case operations and signatures are used to authenticate some specific messages, such as view change messages and client updates (note that this PBFT extension uses MACs to authenticate client updates).

Guaranteeing performance under attack, improving resiliency through recovery mechanisms, and increasing security using asymmetric authentication come at the cost of lower performance during normal operations. However, independent of the adopted system model, those approaches are necessary to build applications that are truly usable in intrusion-prone environments.

10. CONCLUSION

In the last 20 years, a large number of BFT protocols have been presented, which differ from each other in many aspects. In this article, we classified BFT protocols based on their system model, that is, server side versus client side. We described server-side protocols as those in which the execution is entirely run by servers, and client-side protocols as those in which clients act either as *proposers*, *repairers*, or both. We argued that in the client-side model clients should be considered part of the system and, hence, mechanisms that cope with potentially malicious clients should be

provided. In addition to security aspects, we showed that shifting part of the job to the client side makes applications more scalable on WANs. We described some applications that use the BFT server-side or client-side model, highlighting differences and trade-offs. Finally, we presented some current research gaps in the literature of BFT protocols and discussed the main aspects that a system builder should consider when building practical BFT applications.

REFERENCES

Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 59–74.

Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 1–14.

Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2008. Byzantine replication under attack. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks with FTCS and DCC*. IEEE, 197–206.

Yair Amir, Jonathan Kirsch, John Lane, and Marco Platania. 2014. Prime 2.0. Retrieved from http://www.dsn.jhu.edu/download.html.

Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2014. Scalable BFT for multi-cores: Actor-based decomposition and consensus-oriented parallelization. In *Proceedings of the 10th USENIX Conference on Hot Topics in System Dependability*. USENIX Association, 9–14.

Mihir Bellare, Ran Canetti, and Hugo Krawczyk. 1996. Keying hash functions for message authentication. In *Advances in Cryptology—CRYPTO'96*. Springer, 1–15.

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* 9, 4 (2013), 12.

George Robert Blakley. 1979. Safeguarding cryptographic keys. In *Proceedings of the International Workshop on Managing Requirements Knowledge*. IEEE Computer Society, 313–317.

Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems* 20, 4 (2002), 398–461.

Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009a. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 277–290.

Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009b. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI)*, Vol. 9. USENIX, 153–168.

Frederick B. Cohen. 1993. Operating system protection through program evolution. *Computers and Security* 12, 6 (1993), 565–584.

James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 177–190.

Wagner Saback Dantas, Alysson Neves Bessani, Joni da Silva Fraga, and Miguel Correia. 2007. Evaluating Byzantine quorum systems. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*. IEEE, 253–264.

Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654.

Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. 2013. PoWerStore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 285–298.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM* 35, 2 (1988), 288–323.

C. Fragouli, J. Le Boudec, and J. Widmer. 2006. Network coding: An instant primer. *Computer Communication Review* 36, 1 (2006), 63.

Miguel Garcia, Nuno Neves, and Alysson Bessani. 2013. An intrusion-tolerant firewall design for protecting SIEM systems. In *Proceedings of the 2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 1–7.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the USENIX Security Symposium*.

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. IEEE, 135–144.

Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems (Eurosys)*. ACM, 363–376.

James Hendricks, Gregory R. Ganger, and Michael K. Reiter. 2007. Low-overhead Byzantine fault-tolerant storage. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 73–86.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–11.

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 8. 9.

Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. 2014. Survivable SCADA via intrusion-tolerant replication. *IEEE Transactions on Smart Grid* 5, 1 (2014), 60–70.

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 45–58.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565.

Leslie Lamport. 1986. On interprocess communication. *Distributed Computing* 1, 2 (1986), 86–101.

Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.

Barbara Liskov and Rodrigo Rodrigues. 2006. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 34–34.

Dahlia Malkhi and Michael Reiter. 1998a. Byzantine quorum systems. *Distributed Computing* 11, 4 (1998), 203–213.

Dahlia Malkhi and Michael K. Reiter. 1998b. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*. IEEE, 51–58.

Dahlia Malkhi and Michael K. Reiter. 1998c. Survivable consensus objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*. IEEE, 271–279.

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

Michael G. Merideth and Michael K. Reiter. 2010. Selected results from the latest decade of quorum systems research. In *Replication*. Springer, 185–206.

Zarko Milosevic, Martin Biely, and André Schiper. 2013. Bounded delay in Byzantine tolerant state machine replication. In *Proceedings of the IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 61–70.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 601–615.

Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *Journal of the ACM* 27, 2 (1980), 228–234.

Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. 2014. Towards a practical survivable intrusion tolerant replication system. In *Proceedings of the IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 242–252.

Ronald L. Rivest, Adi Shamir, and Len Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21, 2 (1978), 120–126.

Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. 2001. BASE: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 15–28.

Tom Roeder and Fred B. Schneider. 2010. Proactive obfuscation. *ACM Transactions on Computer Systems* 28, 2 (2010), 4.

Douglas C. Schmidt and Tatsuya Suda. 1993. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications* 11, 4 (1993), 489–506.

Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. 2010. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 353–362.

Adi Shamir. 1979. How to share a secret. *Communications of the ACM* 22, 11 (1979), 612–613.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.

João Sousa and Alysson Bessani. 2012. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 2012 9th European Dependable Computing Conference (EDCC)*. IEEE, 37–48.

Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2010. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (2010), 452–465.

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating agreement from execution for Byzantine fault tolerant services. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 253–267.

Wenbing Zhao and F. Eugenio Villaseca. 2008. Byzantine fault tolerance for electric power grid monitoring and control. In *Proceedings of the International Conference on Embedded Software and Systems (ICESS'08)*. IEEE, 129–135.

Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. 2002. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 329–368.

Received March 2015; revised November 2015; accepted January 2016