Scalable Byzantine Fault-Tolerant Storage

Ricardo Padilha, Fernando Pedone
University of Lugano
Switzerland
Email: {ricardo.padilha, fernando.pedone}@usi.ch

***Abstract*—Byzantine fault-tolerance is costly, both in terms of increased latency and limited scalability. Most recent contributions in the area have addressed the latency aspect, leaving the throughput scalability problem largely untouched. We propose in this short paper to build scalable Byzantine fault-tolerant storage systems based on the concept of mini-transactions. To achieve this goal we propose a novel atomic commit protocol that tolerates malicious clients and servers.**

***Keywords*-byzantine fault-tolerance, storage, scalability, mini-transactions**

I. Introduction

Byzantine fault-tolerance is a fundamental requirement for many contemporary services [1]. Shorter development cycles, bigger server exposure to attacks, and an inherently hostile network have demonstrated the weaknesses of replication mechanisms designed for benign failures. Unfortunately, Byzantine fault-tolerant (BFT) services usually have *increased latency*, when compared to simple client-server interactions, and *limited scalability*, in the sense that adding replicas does not translate into higher throughput.

Except for a few exceptions [2], [3] discussed later, most recent contributions in the area of BFT protocols have sought to address the latency problem [4], [5], [6]. The lack of scalability is derived from the fact that BFT services rely either on state-machine replication or primary-backup replication, and neither is scalable. With state-machine replication every operation must be executed by every replica; thus, adding replicas does not increase throughput. With primary-backup replication, the primary executes operations first and then propagates the state changes to the backups; system throughput is determined by the primary.

This paper addresses the scalability of BFT storage systems. Our approach builds on the paradigm of mini-transactions [7], developed in the context of fail-stop nodes. We propose to partition the storage state among servers and make each partition Byzantine fault-tolerant individually, by means of state-machine replication. We handle operations across partitions with a novel BFT atomic commit protocol. Scalability stems from the fact that partitions can execute mini-transactions in parallel.

The remainder of this paper is organized as follows: Section II defines the system model and Section III details the novel BFT atomic commit protocol. Section IV briefly evaluates the impact of contention. Section V reviews related work and Section VI concludes with some future work.

This work was partially funded by the Hasler Foundation, Switzerland, project number 2316.

II. System model and definitions

We assume an asynchronous distributed system where nodes communicate by message passing. There are no known bounds on processing times and message delays. Links may fail to deliver, delay, or duplicate messages, or deliver them out of order. However, links are fair: if a message is sent infinitely often to a receiver, then it is received infinitely often.

Nodes can be correct or faulty. A *correct* node follows its specification whilst a *faulty* node can present Byzantine (i.e., arbitrary) behavior. Byzantine nodes can be coordinated by an adversary that can also inject spurious messages into the network, or delay correct nodes in order to cause the most damage to the replicated service. However, adversaries cannot delay correct nodes indefinitely. We assume that it is not computationally feasible for adversaries to subvert the cryptographic primitives used for signing and authenticating messages. There is an arbitrary number of client nodes and a fixed number n of server nodes, where clients and servers are disjoint. An undefined but limited number of clients can be byzantine. We define the number of Byzantine servers next.

We consider a storage system composed of $(key, value)$ entries, distributed among the server nodes and accessed by the client nodes by means of mini-transactions. *Mini-transactions* offer ACID properties, and were originally proposed for the fail-stop failure model [7]. A mini-transaction can contain any number of $read(key)$, $write(key, value)$, and $cmp(key, value)$ operations. Operations of type cmp perform equality comparison and are executed first. If all cmp operations of a mini-transaction are successful, then $read$ and $write$ operations are executed. We assume that both key and value are of arbitrary type and length.

We also assume the existence of a BFT total order broadcast protocol, defined by the primitives broadcast(g, m) and deliver(m), where g is a group of servers and m is a message. Total order broadcast ensures that (a) a message broadcast by a correct client to group g will be delivered by all correct servers in g; (b) if a correct server in g delivers m, then all correct servers in g deliver m; and (c) any two

correct servers in g deliver messages in the same order. While several BFT total order broadcast protocols satisfy the properties above, we assume in this paper FaB [8]. FaB can deliver messages in two communication steps and requires $n_g = 5f_g + 1$ servers, where f_g is the number of Byzantine servers in g.

III. Scalable BFT storage

The approach we propose is to partition the key space and store each partition in a group of servers. Each group g of size n_g is implemented with state-machine replication and can tolerate f_g Byzantine failures.

Mini-transactions that involve multiple partitions are terminated with an atomic commit protocol (see Figure 1). The complexity of the approach lies in efficiently implementing the atomic commit protocol in the presence of Byzantine clients and servers.

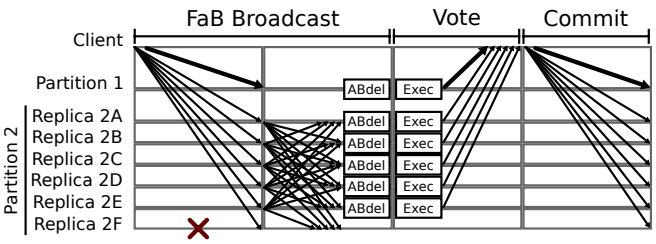

Figure 1. Overview of the scalable BFT protocol.

A. The normal case

A correct client broadcasts the operations of a mini-transaction, say t, to all involved partitions, performing one broadcast call per partition (Algorithm 1, lines 2–8). From t's operations, the node can determine the partitions involved in t (function $part(t.ops)$ at line 6) and t's unique identifier after computing a digest of t's operations (line 3). To ensure identifier uniqueness, we require each mini-transaction to be extended with a "nop" operation containing a random number.

Each correct replica p involved in t delivers it and computes t's unique id (lines 35–36). If p has not voted for t yet (line 37), p checks whether t conflicts with any pending mini-transaction u (line 38): t conflicts with u if t accesses some key that u modified. If there is no conflict, p executes t (line 39); if there is a conflict, p aborts t (line 42). In either case, p sends a response to the client (lines 40 and 43). If p votes to commit t, t becomes *pending* at p (line 44); else if p votes to abort t, t becomes a *decided* mini-transaction (line 45). Finally, if p has already voted for t (i.e., this is a client's retransmission), p simply resends its previous response (line 47).

The client receives the responses, and once it gathers $f_g + 1$ matching responses from replicas in partition g, it determines the vote and result from g (line 15). The client uses the collected votes from g to assemble a *vote certificate*, that is, $f_g + 1$ signatures per partition that will be used to prove the partition's vote (line 16). The outcome of t will be *commit* if the client collects commit votes from all partitions (lines 18–19) and *abort* if at least one partition votes to abort t (line 21). Whatever t's outcome, the client sends the vote certificate proving t's outcome to all replicas in the concerned partitions (line 11) and notifies the application (lines 12–13). Primitive sendall(g, m) (line 11) sends message m to all members of group g.

When p receives a valid vote certificate for t from the client (line 49), p checks whether t is a pending transaction and has not been already terminated (line 50), determines the outcome of t from the vote certificate (line 51) and proceeds accordingly: p either commits t's writes (line 51) or rolls back t's operations (line 52). In either case, t is no longer a pending transaction (lines 53–54).

As shown in Figure 1, in the best case clients observe a latency of three communication steps, as opposed to two in the original mini-transaction protocol for fail-stop failures [7].

B. Handling byzantine clients

Byzantine clients can attempt to subvert the mini-transaction termination protocol in two ways: (a) by presenting different partitions non-matching mini-transactions (e.g., by broadcasting different mini-transaction bundles to each partition) and (b) by not terminating or mis-terminating the protocol (e.g., not sending the signed votes back).

To address scenario (a), recall from the previous section that a mini-transaction is uniquely identified by its contents. Thus, non-matching mini-transactions yield different identifiers and are considered different transactions altogether by the partitions. Since to be terminated a mini-transaction requires a valid vote certificate, non-matching mini-transactions will lead to one or more unfinished mini-transactions (i.e., forever in the pending state), which is handled by scenario (b).

Our strategy to address scenario (b) is to rely on subsequent correct clients to complete mini-transactions left unfinished. To a certain extent, correct clients play the role of "recovery coordinators", as described in [7]. From the previous section, if a mini-transaction t conflicts with a pending mini-transaction u in some replica $p \in g$ (line 41), t is aborted by p (line 42). As part of the abort message sent by p to the client (line 43), p also sends information about u, so that the client can terminate u. The information contains u's identifier and the set of partitions concerned by u. Notice that since the client will propose to abort u, it does not need u's operations.

When the client receives the above message from $f_g + 1$ replicas in g (line 23), in addition to aborting t (line 26), the client starts the termination of u by contacting every partition h that is accessed by u (line 28) and proposing to

Algorithm 1 Scalable BFT storage

1: **Client-side algorithm:**

2: *To execute mini-transaction t with operations ops do:*
3: $t.id \leftarrow$ digest(ops); $t.ops \leftarrow ops$
4: $t.cset \leftarrow \emptyset$ // $t.cset$ has all groups g willing to commit t
5: $t.status \leftarrow \perp$ // $t.status$ is a value in $\{\perp$, commit, abort$\}$
6: **for all** $g \in part(t.ops)$ **do** // for each partition g in t
7: $t.cert[g] \leftarrow \perp$ // no vote certificate for t from g so far
8: broadcast$(g, t.ops)$ // broadcast $t.ops$ to g

9: *Function* finish$(t, outcome)$
10: $t.status \leftarrow outcome$ // $t.status$ is committed or aborted
11: **for all** $g \in part(t)$ **do** sendall$(g, t.id, t.cert)$
12: **if** $outcome = commit$ **then** notify (COMMIT, $t.res$)
13: **else** notify (ABORT)

14: *Task 1 (client: receive votes from each partition)*
15: **upon** receive$(\{t.id, t.vote\}_*, t.res)$ from f_g+1 replicas in g
16: $t.cert[g] \leftarrow \{t.id, t.vote\}_{p_1,\ldots,p_{f_g+1}}$ // add g's vote certificate
17: **if** $t.vote =$ commit **then**
18: $t.cset \leftarrow t.cset \cup g$ // add g to t's commit set
19: **if** $t.cset$=$part(t.ops)$ **then** finish(t,commit) // all voted
20: **else**
21: **if** $t.status$=$\perp$ **then** finish(t, abort) // one abort is enough

22: *Task 2 (client: recovery path)*
23: **upon** receive $(\{t.id,$ abort$\}_*, \{u.id, u.part)\}_*)$ from f_g+1 replicas in g
24: $t.cert[g] \leftarrow \{t.id,$ abort$\}_{p_1,\ldots,p_{f_g+1}}$
25: **if** $t.status = \perp$ **then**// if I did not abort t yet...
26: finish(t, abort) // ...do it, and...
27: $u.cset \leftarrow \emptyset$; $u.status \leftarrow \perp$ // ...try to terminate u
28: **for all** $h \in u.part$ **do** // for each partition h accessed by u
29: $u.cert[h] \leftarrow \perp$ // no vote certificate for u from h so far
30: broadcast$(h, u.id,$ abort$)$ // propose outcome, force termination of u in h

31: **Server-side algorithm:**

32: *Initialization:*
33: $pending \leftarrow \emptyset$; $decided \leftarrow \emptyset$

34: *Task 3 (replica p: normal case)*
35: **upon** deliver $(g, t.ops)$, broadcast by client c
36: $t.id \leftarrow$ digest$(t.ops)$
37: **if** $t.id \notin pending \cup decided$ **then**// if I did not vote for t yet
38: **if** $\nexists u.id \in pending : t$ conflicts with u **then**
39: $(t.vote, t.result) \leftarrow$ execute(t)
40: send$(\{t.id, t.vote\}_p, t.result)$ to c
41: **else**// there is a pending u that conflicts with t
42: $t.vote \leftarrow$ abort
43: send$(\{t.id,$ abort$\}_p, \{u.id, part(u.ops)\}_p)$ to c
44: **if** $t.vote =$ commit **then** $pending \leftarrow pending \cup t.id$
45: **else** $decided \leftarrow decided \cup t.id$ // t is decided with one abort
46: **else**// if I voted for t already...
47: send$(\{t.id, t.vote\}_p, t.result)$ to c // ...resend respone

48: *Task 4 (replica p: termination)*
49: **upon** receive $(t.id, t.cert)$ from client c **and** valid$(t.cert)$
50: **if** $t.id \in pending$ **and** $t.id \notin decided$ **then**
51: **if** outcome$(t.cert)$ = commit **then** commit(t)
52: **else** rollback(t)
53: $pending \leftarrow pending \setminus t.id$ // t is no longer pending...
54: $decided \leftarrow decided \cup t.id$ // ...it was committed or aborted

55: *Task 5 (replica p: recovery path)*
56: **upon** deliver $(h, u.id,$ abort$)$, broadcast by client c
57: **if** $u.id \notin pending \cup decided$ **then**// if I did not vote for u yet
58: $u.vote \leftarrow$ abort // accept client's proposal, force termination by voting u's abort
59: $decided \leftarrow decided \cup u.id$
60: send$(\{u.id, u.vote\}_p, u.result)$ to c

abort u (line 30). If a vote request for u was not previously delivered in h (e.g., not broadcast by the client that created u), then the members of h will vote abort (lines 58–59). Otherwise, members of h will return the result of the previous vote (line 60). In any case, eventually the client will gather enough votes to complete the pending mini-transaction, following the same steps as the normal case.

Byzantine clients can also try to prevent progress of correct clients: (a) by submitting an unbounded number of transactions simultaneously to the replicas (e.g., not waiting for the termination of a transaction before submitting another transaction) and (b) by proposing to abort transactions of other clients before they are delivered to all involved partitions.

Although Algorithm 1 does not currently address scenario (a), a trivial way of tackling the problem is to limit the number of pending transactions originating from a single client. Each replica could keep track of how many transactions have been submitted by a single client and immediately abort any transactions beyond a system-dependent threshold (e.g., a single transaction at a time).

Scenario (b) is a more subtle attack vector. The scenario requires the collusion between a Byzantine server and a Byzantine client, and can be described in this manner: after a correct client submits a cross-partition mini-transaction to one of the involved partitions, the Byzantine server in that partition notifies the Byzantine client who then sends recovery messages to all the other partitions before the correct client is able to submit it to them. By doing so, the Byzantine client forces the other partitions to vote abort on the yet unseen transaction, leading them to believe that the correct client is Byzantine.

It is important to notice that this attack is only possible in mini-transactions that involve several partitions. Mini-transactions limited to a single partition have their ordering and delivery guaranteed by the BFT total order broadcast, and as such, are impervious to this attack scenario. The weakness here stems from the loose coupling between partitions. Once delivered to all involved partitions, however, a Byzantine client can no longer prevent progress since a replica's vote on the outcome of a transaction is final. This weakness is still an open point in our protocol, but in our implementation it has been mitigated by the usage of IP multicast to submit transactions to all partitions simultaneously.

C. Optimizations

The algorithm presented in the previous sections can be optimized in a number of ways. For example, it has been observed that it is possible to separate BFT agreement from BFT execution [9]. Let n_g^A and n_g^E denote the number of servers in g for agreement and execution, respectively. While FaB requires $n_g^A \geq 5f_g + 1$, execution requires only $n_g^E \geq 2f_g+1$. For the special case of $f = 1$, we have $n_g^A \geq 6$ and $n_g^E \geq 3$, and therefore, we can divide the storage into

two execution partitions, each one with 3 replicas. While all servers participate in the execution of FaB and every mini-transaction is broadcast to both partitions (and discarded by a partition if it does not contain relevant operations to the partition), the execution of mini-transactions can be parallelized.

Currently, a client tries to terminate an unfinished mini-transaction by proposing to abort it. Alternatively, correct clients could try to terminate a pending mini-transaction u by proposing to commit u. In order to do so, a correct client would have to receive u from the replicas, instead of just u's id and involved partitions. This is needed because u may be known only to a subset of its involved partitions.

IV. Evaluation

To evaluate the impact of the partitioning technique, we have implemented a prototype of the special dual-partition case. We compared the throughput of a single pool of replicas, executing all mini-transactions, versus a dual partition layout, where we varied the number of cross-partition mini-transactions.

Our baseline is the throughput of a single group containing all replicas (see Table I, column "1 Partition") running the BFT mini-transaction protocol. This corresponds to a traditional full state-machine replication system. The values presented were taken at saturation point, where the replicas in the baseline configuration were performing at their maximum throughput.

	1 Partition	2 Partitions	
		worst case	best case
Throughput	100%	105%	230%

Table I
Improvements of partition arrangements.

We then split the six replicas in two partitions of three replicas each ("2 Partitions" column in Table I), and considered two setups. In the first setup, we kept each mini-transaction accessing both partitions, which corresponds to the maximum number of cross-partition mini-transactions (i.e., worst case). In this case we observed a 5% increase in throughput, which shows that our approach is better than the baseline, even in the worst case.

We then assigned mini-transactions to a single partition only, corresponding to zero cross-partition mini-transactions (i.e., best case). The throughput increase was significantly higher, at 230%. This confirms that partitioning the replicas in minimal execution quorums can yield significant performance gains.

V. Related work

Although there are comprehensive comparisons of several BFT protocols (e.g., [10]), very little literature exists on the topic of scalable throughput BFT systems. In fact, most of the work following Castro and Liskov's seminal publication (PBFT [11]) focus on state-machine replication and totally ordering of requests.

Most optimizations have focused on reducing the latency of state-machine replication, for example, by avoiding to order requests in the critical path of the execution, opting instead for speculative or optimistic execution [4], [5] or by settling for *eventual consistency* [6]. In any case, as pointed out in the introduction, state-machine is fundamentally non-scalable (see [12] for a more complete discussion in the crash-stop model).

Few works have addressed the problem of scalability in BFT systems with the purpose of increasing the overall system throughput. In [13] the authors introduce an application-dependent framework to support parallel execution of independent requests. In our system, this is done transparently by means of mini-transactions.

Although the separation of agreement and execution is presented in [9], it is only in [3] that we see the first application of the reduction to an $f+1$ execution quorum coupled with on-demand replica consistency. While the approach does provide throughput improvements, it does not ensure throughput scalability, and, as observed by the authors, may decrease throughput when compared to a baseline if many "cross-border" (i.e., inter-partition) requests are executed.

In [14] the authors present another take on the reduction of agreement and execution quorums, by separating replicas in tiers, and introducing delayed dissemination of updates. Although it does support ACID semantics, it is based on an update dissemination protocol that requires a complex model of replicas containing both *tentative* and *commited* data. If strong consistency is required, the system only executes as fast as the primary tier.

In [15] the authors propose a way to improve the scalability of the agreement protocol (i.e., the message ordering). As the authors indicate themselves, scalability of the agreement protocol does not imply scalability of the execution of requests. Their work is orthogonal to ours, since our model is agnostic to the underlying agreement protocol, and could use another agreement protocol, more scalable than FaB, without compromising on the execution scalability.

In [2] the authors propose the first solution focused on throughput scalability. The idea is to implement a centralized locking system that allows clients, once they acquired the required locks, to submit an unlimited number of sequential operations to a BFT quorum of *log servers*. BFT agreement is only executed at the end to ensure the client submitted the same operations in the same order to all log servers. Scalability is achieved by dividing the application state across several partitions of log servers. As pointed out by the authors, their solution is optimized for the specific single-client, low-contention, single-partition data scenario. It is also important to point out that throughput of the whole

system is bound by the throughput of the centralized locking system.

In [16] the first BFT two-phase commit protocol is presented. The BFT agreement proposed only extends to a subset of the participants in the transaction, and due to its hierarchical nature, it is not throughput-oriented.

State partitioning is a tried and true approach to improving scalability of data management systems [17], [18]. In [19] BFT groups are used to store the metadata of a distributed filesystem, while the data itself is replicated . Their approach allows load-distribution, but does not offer atomic, transactional execution across groups. Overall, our system is the first to provide scalable throughput while offering strong consistency and system-wide atomic operations without relying on specific application behavior.

VI. Final remarks

The field of throughput-scalable BFT storage is surprisingly unexplored. Recent works have attempted to address some of the issues, such as the overhead in latency, the replication cost, or even the throughput for specific scenarios. Our approach addresses the scalability problem for overall system throughput without emphasizing on specific application profiles.

In traditional BFT systems, the inclusion of an extra replica usually means increased availability only, but not necessarily increased throughput. We introduced state partitioning as the means to achieve greater throughput, while still supporting inter-partition atomic operations. Our preliminary evaluation indicates that throughput increases can be obtained "for free," just by reorganizing the replicas in separate execution partitions within the agreement quorum.

The usage of FaB allows us to provide fast delivery of messages within partitions, and the usage of mini-transactions allows us to consistently execute requests across several partitions. FaB totally orders all mini-transactions in a partiton, but could be easily replaced by any other agreement protocol, such as the one presented in [15]. More over, we are also investigating weaker ordering protocols, which consider mini-transaction operations and provide higher concurrency [20].

Acknowledgements

The authors wish to thank the anonymous reviewers for the insightful comments about this work.

References

[1] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin, "Bft: the time is now," in *Workshop on Large-Scale Distributed Systems and Middleware*, vol. 341. New York, NY, USA: ACM, September 2008, pp. 1–4.

[2] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, "Zzyzx: Scalable fault tolerance through byzantine locking," in *DSN '10: Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, July 2010, pp. 363–372.

[3] T. Distler and R. Kapitza, "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency," in *EUROSYS '11: Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems*. Salzburg, Austria: ACM, April 2011.

[4] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EUROSYS '10: Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. Paris, France: ACM, April 2010, pp. 363–376.

[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, December 2009.

[6] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," in *NSDI '09: Proceedings of the 6th Symposium on Networked Systems Design and Implementation*. Boston, MA, USA: USENIX, April 2009.

[7] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP '07: Proceedings of the 21st ACM Symposium on Operating Systems Principles*. Stevenson, WA, USA: ACM, October 2007, pp. 159–174.

[8] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, July 2006.

[9] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, USA: ACM, October 2003, pp. 253–267.

[10] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire," in *NSDI '08: Proceedings of the 5th Symposium on Networked Systems Design and Implementation*. San Francisco, CA, USA: USENIX, April 2008, pp. 189–204.

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI '99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. New Orleans, LA, USA: USENIX, February 1999, pp. 173–186.

[12] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *DSN '11: Proceedings of the 41th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Hong Kong, China: IEEE, June 2011.

[13] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN '04: Proceedings of the 34th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Florence, Italy: IEEE, June 2004, pp. 575–584.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," in *ASPLOS 2000: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, USA: ACM, November 2000, pp. 190–201.

[15] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service," in *Proceedings of the 6th Workshop on Hot Topics in System Dependability*. Vancouver, BC, Canada: USENIX, October 2010.

[16] C. Mohan, H. R. Strong, and S. J. Finkelstein, "Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors," in *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. Montréal, Québec, Canada: ACM, August 1983, pp. 89–103.

[17] S. A. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design," in *SIGMOD '82: Proceedings of the ACM SIGMOD International Conference on Management of Data*. Orlando, FL, USA: ACM, June 1982, pp. 128–136.

[18] H. Garcia-Molina and B. Kogan, "Achieving high availability in distributed databases," *IEEE Transactions on Software Engineering*, vol. 14, no. 7, pp. 886–896, July 1988.

[19] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. Boston, MA, USA: USENIX, December 2002, pp. 1–14.

[20] P. Raykov, "Byzantine generic broadcast," Master's thesis, University of Lugano, Lugano, Switzerland, June 2010.