Byzantine Fault Tolerance for Services with Commutative Operations

Hua Chai and Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115
wenbing@ieee.org

***Abstract*—In this paper, we present a comprehensive study on how to achieve Byzantine fault tolerance for services with commutative operations. Recent research suggests that services may be implemented using Conflict-free Replicated Data Types (CRDTs) for highly efficient optimistic replication with the crash-fault model. We extend such studies by adopting the Byzantine fault model, which encompasses crash faults as well as malicious faults. We carefully analyze the threats towards the operations in a system constructed with CRDTs, and propose a lightweight solution to achieve Byzantine fault tolerance with low runtime overhead. We define a set of correctness properties for such systems and prove that the proposed Byzantine fault tolerance mechanisms guarantee these properties. Furthermore, we show that our mechanisms exhibit excellent performance with a proof-of-concept replicated shopping cart service constructed using CRDTs.**

***Index Terms*—Byzantine Fault Tolerance, Optimistic Replication, CAP Theorem, Asynchronous Communication, Network Partitioning**

I. Introduction

Service-oriented computing is transforming the Internet from a predominantly publishing platform to a programmable distributed computing platform, which has led to more and more business activities conducted online via cloud services. Many such services are mission critical, such as those related to financial, healthcare, and customer confidential records. To ensure high availability of such services, replication has been pervasively used with a crash-fault only model [1], [2], [3]. We argue that due to the open environment in which theses services operate, a more general fault model, namely, the Byzantine fault model [4], ought to be used to achieve sufficient level of dependability and trustworthiness.

There are a large body of work on Byzantine fault tolerance [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. The most common approach is state-machine replication using a Byzantine agreement algorithm to ensure the total ordering of all incoming requests from various clients to a replicated service. Such an approach would require that all requests are executed sequentially according to the total order established. Unfortunately, this would render the technology impractical for many cloud-based services because it would impose severe runtime overhead. Recognizing this limitation, application semantics has been exploited to enable concurrent execution of independent requests [8], [11], [12], [13]. While this approach may improve the throughput of the replicated service significantly, it requires intimate knowledge on the application design and implementation, which may be expensive and error-prone. Furthermore, existing Byzantine fault tolerance solutions are conservative in nature and dictate strongly-coupled coordination among the replicas, which is orthogonal to common practices of relying on optimistic replication and eventual consistency for cloud-based services [2], [3].

Recent research suggests that a service may be constructed by using Conflict-free Replicated Data Types (also referred to as commutative or convergent replicated data types, or CRDTs in short) for highly concurrent optimistic replication with the crash-fault model [14], [15], [16], [17]. In this paper, we extend such studies by adopting the Byzantine fault model [4], which encompasses both crash faults and malicious faults. We carefully analyze the threats towards the operations in a replicated service constructed with CRDTs, and propose a lightweight solution to achieve optimistic Byzantine fault tolerance with low runtime overhead.

Our lightweight solution does not guarantee the total ordering of all messages, as it is not needed for commutative operations. The primary challenge is to handle an insidious threat imposed by an adversary: it may disseminate conflicting information to different replicas. We mitigate such threats and ensure eventual Byzantine consistency by engaging in on-demand and periodic synchronization of the replica states via a Byzantine agreement service. A round of state synchronization can be triggered on-demand by a client upon detecting the presence of conflicting requests, or periodically when the replica has executed a predefined number of requests.

This paper makes the following contributions:

- A detailed threat analysis is carried out on services constructed using CRDTs.
- A set of efficient mechanisms are proposed to ensure eventual replica consistency and the causality of the system despite Byzantine faulty clients and server replicas.
- A set of correctness properties are formulated and proofs of correctness of our mechanisms are provided with respect to the set of properties.
- A proof-of-concept application using CRDTs and the set of Byzantine fault tolerance mechanisms are implemented. Furthermore, the performance of the system is carefully evaluated with promising results compared with an alternative approach.

II. Background

A. Byzantine Fault Tolerance

The term "Byzantine fault" was coined by Lamport to represent an arbitrary fault [4], which might be a crash fault due to hardware failures, or a malicious fault due to software malfunction caused by an intrusion into the system. Byzantine fault tolerance refers to the capability for a system to provide correct services to its clients in the presence of Byzantine faults. Byzantine fault tolerance can be achieved by replicating a critical component (typically the server process) in a system, and by ensuring that all replicas reach an agreement on the total order of each incoming request despite the presence of Byzantine faulty replicas and clients. Such an agreement is referred to as a Byzantine agreement (BA) [4]. For state-machine replication [18], the BA algorithm is optimized to ensure the total ordering of a sequence of requests instead of a single value.

The first practical Byzantine fault tolerance algorithm (referred to as PBFT) is due to Castro and Liskov [5]. In [5], Byzantine fault tolerance is achieved via two sub-algorithms, one algorithm to ensure the total ordering of requests during normal operation, and the other algorithm to ensure liveness when the primary replica fails. Recently, Zyzzyva [6] further optimized PBFT such that it takes only one additional communication step to handle each client's request during optimal conditions. Both [5] and [6] are designed for state-machine replication, where $3f+1$ replicas are needed to tolerate up to f Byzantine faulty replicas.

In [9], Yin et al., pointed out that agreement and execution can be separated in a Byzantine fault tolerant system where a cluster of nodes are dedicated to ensure the total ordering of requests, and as few as $2f+1$ server replicas can be used for executing the requests from clients. This opens the door for providing and using Byzantine fault tolerance as a service [19], which is adopted in our work.

B. Optimistic Replication and CRDTs

Optimistic replication [20], where a replica is allowed to execute a client's request immediately, without block waiting until the request is totally ordered, is an attractive approach to practioners due to its low runtime overhead and robustness under network partitioning faults compared to traditional conservative approaches. Indeed, this is the case as seen from the development of the CAP theorem [2] and the optimistic replication strategy employed in practical cloud systems [3].

Optimistic replication aims to achieve eventual consistency among the replicas [20], *i.e.,* the states of the replicas will eventually converge when the clients stop issuing new requests and the network is reasonably connected so that a request executed at one replica can be propagated to all other nonfaulty replicas. However, enabling concurrent executions of potentially conflicting requests is challenging. Operational transformation was proposed to facilitate the convergence of the states of different replicas involved with conflicting operations [21]. However, it has been pointed out in [22] that most operational transformation algorithms for decentralized architecture are not correct.

In [17], Shapiro et al., proposed to use CRDTs to build replicated services without the need of total ordering of all requests, and without suffering from the problems related to operational transformations, under a crash-fault model. Some weak forms of synchronization are required to perform garbage collection under the crash-fault model. However, it can be done off the critical path, hence, would have minimum performance impact.

A list of CRDTs are introduced in [17], including various types of counters, registers, sets, maps, graphs, and sequences. It is shown that these CRDTs can be used to build practical applications. Here we explain the set type in more detail because it is used in our performance evaluation. A single regular set can only be used as a CRDT for add-only operations, which is not useful for practical applications. To add support for remove operations, two sets, referred to as A and R, respectively, are used together as a U-Set type [17]. New elements added to a U-Set are added to A, and removed elements are moved to R, which is also referred to as the tombstone set. Only elements in A can be removed. In response to a query for the current elements in the U-Set, the difference between A and R are returned.

III. System Model

We assume a client-server architecture where the server is replicated and one or more clients issue requests to the server replicas. The server is constructed using one or more CRDTs such that all update operations on the server replicas are commutative. A client may issue two different types of requests:

- Update requests. The requests would trigger update operations at the server replicas.
- Read-only requests. The requests would retrieve the state of the server replicas. The corresponding reply would contain values that reflect the state.

We assume that both the client and the server replicas are subject to Byzantine faults. The network may partition occasionally, but eventually the network connectivity is restored. All messages exchanged in the system are protected by a security token, such as a digital signature or a message authentication code, for the purpose of authentication and for protecting the integrity of each message. We assume that an adversary has limited computing power and cannot break the security token. A faulty client may collude with other faulty server replicas. However, an adversary cannot impersonate a nonfaulty client or a nonfaulty server replica.

We further assume the availability of an agreement cluster for ensuring Byzantine agreement. The agreement cluster is needed during replica state synchronization for the purpose of replica state convergence and garbage collection.

IV. Threat Analysis

In this section, we analyze potential threats that could compromise the integrity of the replicated service. We do not

consider general threats that could target any online services, such as distributed denial of attacks. We divide the threats under consideration into the following categories.

A. Threats from a Faulty Server Replica

1) Threats towards the client: A faulty replica could attempt to disrupt the service to a client in the following ways:

- The faulty replica may refuse to respond to the client's request. It may or may not actually execute the request.
- The faulty replica may issue a reply to the client after correctly executing a request, but the reply contains information that is inconsistent with the actual execution. For example, the replica could inform the client that an exception has occurred when in fact the execution is completed successfully.
- The faulty replica does not perform the operations intended by the client. For example, the replica may place an order for 2 units instead of 1 unit as the request indicated.

The first scenario is equivalent to a denial of service attack on the client. The remaining two scenarios aim to compromise the integrity of the service. All the above threats can be controlled by using sufficient number of server replicas and voting on the replies received from different replicas in response to a request at the client.

2) Threats towards other server replicas: A Byzantine faulty replica could lie about its execution status and the requests received when it engages in a round of state synchronization. State synchronization is necessary to ensure eventual replica consistency and garbage collection (the mechanism will be elaborated in Section V-B2). This attack could potentially prevent a nonfaulty replica from completing a round of a state synchronization, or result in divergent replica states at the end of a round of state synchronization. Consider the following attack scenarios:

- A faulty replica could report a nonexistent operation and does not respond to retransmission requests for the corresponding update request from other replicas. In this case, the replicas that requested retransmission for the nonexistent operation would not be able to complete the current round of state synchronization.
- A faulty replica could collude with a faulty client. The faulty client could provide a list of update requests with the same identifiers to the faulty replica, but not other replicas. The faulty replica then report a single operation to other replicas during state synchronization, but transmit different requests to different replicas, which could lead to divergent states at different nonfaulty replicas.

These threats must be addressed by using a Byzantine agreement service on the operation histories reported by each replica, as well as a carefully designed state synchronization mechanism.

B. Threats from a Faulty Client

A faulty client may attack server replicas in various ways. However, it cannot attack other clients directly. A faulty client may attempt to attack other clients indirectly by compromising the state of the server replicas. With a sound Byzantine fault tolerance mechanism, such attacks can be rendered ineffective.

1) Threats towards server replicas: The most insidious threat towards server replicas imposed by a faulty client is to send conflicting requests to different server replicas. Such threat, if not controlled, could lead to the divergence of the server states at different nonfaulty replicas. Controlling such threats would normally require the use of a conservative approach, *i.e.,* before a replica is allowed to deliver and execute a request, a Byzantine agreement must have been reached on the identity and the total order of the request to ensure that all nonfaulty replicas deliver the same set of requests in the same total order.

Fortunately, with commutative operations, the negative impact of executing conflicting requests at different replicas is minimized because a wrong update operation would have no impact on another update operation. The state divergence problem caused by such threats is mitigated by periodic and on-demand state synchronization among the replicas.

Besides sending conflicting requests to different replicas, a faulty client may also send malformed requests. Such threats can be controlled by conventional security mechanisms, such as the use of a reference monitor [23], and/or acceptance tests [24]. In this paper, we do not consider this type of threats.

V. Byzantine Fault Tolerance Mechanisms

In this section, we describe a set of lightweight mechanisms for achieving Byzantine fault tolerance for services with commutative operations. Our mechanisms require the use of $3f + 1$ server replicas to tolerate up to f faulty replicas. Each replica is assigned a unique id k, where k varies from 0 to $3f$. We first present the correctness properties of our framework, then we elaborate the Byzantine fault tolerance mechanisms, and finally we prove the correctness of the mechanisms.

A. Correctness Properties

P1. If a nonfaulty client issues a request, it will eventually receive $2f + 1$ matching replies needed to complete the request. Furthermore, if the request is an update request, it will eventually be executed at all nonfaulty replicas.

P2. The values contained in a reply to a read-only request reflect the state changes of all requests that causally precede it.

P3. If a faulty client issues conflicting requests to different nonfaulty server replicas, the state changes of at most one of the requests will be propagated to other nonfaulty clients.

Property P1 ensures eventual state consistency of nonfaulty server replicas. Property P2 ensures causality of the operations of the system, *i.e.,* if there exists an update request that causally precedes a read-only request, the reply to the read-only request must reflect the state changes made by the update request. Property P3 prevents cascading rollbacks when a faulty client that has issued conflicting requests is detected.

B. Mechanisms Description

Our mechanisms operate in two different modes: normal operation (in the absence of faulty clients or faulty server replicas) and state synchronization modes.

1) Normal Operation: During normal operation, our mechanism does not incur any additional communication step in the critical path of the request execution. A request issued by a client has the form $<\text{REQUEST}, o, t, c>_{\sigma_c}$, where o is the operation to be invoked at the server replica, t is a monotonically increasing timestamp, c is the identifier of the client, and σ_c is the security token used to authenticate the client and to protect the integrity of the message. The timestamp t is used to ensure exactly-once semantics for the execution of client requests and it could be the value of the client's local clock when the request is issued or a sequence number. Timestamps for a client's requests are totally ordered such that later requests have higher timestamps than earlier ones.

When a server replica receives a request, it validates the message by checking the timestamp and the security token for the request. The request is ignored if its security token is invalid. If the request's timestamp is not greater than that is last seen by the replica, the request is likely a retransmitted message, and the corresponding reply is fetched from its cache and sent to the client.

A server replica executes a valid request and sends the corresponding reply in the form $<\text{REPLY}, t, c, i, R>_{\sigma_i}$ to the client, where c is the client identifier, t is the timestamp of the corresponding request, i is the replica identifier, and R is the result of executing the requested operation. For an update request, the replica logs the request and a proof of execution $<t, c, d>$, where d is the digest of the original request from the client c. The replica also caches the reply generated.

The client waits for $2f + 1$ matching (and valid) replies from different replicas. Two reply messages sent by different replicas are said to match provided that:

- They are referring to the same request, *i.e.,* they have identical t and c.
- They have the same result R.

Once a client accepts the reply, it delivers the reply and may continue issuing the next request. If the client fails to receive sufficient number of matching replies within a predefined period, it retransmits the request to all replicas. This mechanism might produce duplicate requests. However, this will not cause any problem because the sever replica can detect duplicate messages based on their timestsamps and a cached reply is resent to the client if a duplicate request is received.

2) State Synchronization: With the crash-fault model, eventual consistency of the replicas' states can be achieved when the system is quiesced (*i.e.,* when clients stop issuing new requests for sufficiently long period of time) and each replica disseminates the requests it has executed to other replicas. However, under the Byzantine fault model, the states of the replicas might never converge even if the system is quiesced and all nonfaulty replicas disseminate the requests they have executed to other replicas, because a faulty client may send conflicting requests to different nonfaulty server replicas. Therefore, explicit rounds of synchronization of the state information must be used to achieve state convergence. Furthermore, periodic synchronization also helps on performing garbage collection to truncate logs at the server replicas.

Replica state synchronization may also be triggered by a client. Because a Byzantine faulty client may disseminate conflicting requests to different server replicas, a nonfaulty client may not be able to collect $2f+1$ matching replies to its request (however, it *is* able to collect $2f + 1$ replies from different replicas). After a predefined number of retransmissions, the client would demand a round of state synchronization using the collected mismatched set of replies ($2f + 1$ or more) as the evidence of the problem.

Before we proceed further, it is necessary to formally define what we mean by conflicting requests. Two requests issued by a faulty client are said to be conflicting if they have the same timestamp t and the same client id c, but with different operations.

A replica enters a synchronization round if it has executed a predefined a number of requests since the last round of synchronization, or when it has received a synchronization request from a client. Upon entering a round of state synchronization, a replica issues an agreement message to the agreement cluster and stops executing requests from clients. The agreement message has the form $<\text{AGREEMENT}, b, i, O>_{\sigma_i}$, where b is the round id, i is the replica id, and O is a list of operation entries. Each operation entry is a proof-of-execution record for an operation $<t, c, d>$ that is logged at a replica for each update operation.

Since the replicas execute clients requests in parallel, they may have different operation history O. By using an agreement cluster, it is guaranteed that all nonfaulty replicas receive the same set of totally ordered agreement requests. Upon receiving the first $2f + 1$ totally ordered agreement requests issued by different replicas, a replica computes two lists of operations: (1) the superset of the operation history, $Super\_Set\_Ops$, and (2) an undo list of operations that the replica has already executed but not included in the superset, $Undo\_List$, in the following ways:

- If an operation identified by t, c is included in operation histories, and all the records with t, c have the same digest d (*i.e.,* no conflicting records are detected), the operation record is included in the superset.
- If two or more records are detected with identical t and c, but different digest d, they are due to conflicting requests sent by a Byzantine faulty client. If $f + 1$ or more records with identical t, c, d are detected in the collected operation histories, the corresponding operation is included in the superset. Note that there can be at most one such operation for each t, c that may be included in the superset. Other conflicting operations are recorded in $Undo\_List$. Furthermore, the Byzantine faulty client that issued conflicting requests is blacklisted and no future

requests from that client will be accepted.

- After the replica has iterated all $2f + 1$ agreement messages, it examines each operation in $Undo\_List$ and purge the operation from $Undo\_List$ if has not executed the operation locally.
- It is possible that the replica's own agreement message is not one of the first $2f + 1$ agreement messages. If this is the case, the replica examines the operations included in its own agreement message and add each such operation that is not included in $Super\_Set\_Ops$ to $Undo\_List$.

At the end of the above computation, the replica performs an undo operation for each operation in $Undo\_List$, to rollback the state changes caused by the rejected operation. Furthermore, the replica derives a To_Do_List for operations that are included in $Super\_Set\_Ops$, but it has not yet executed. Then, the replica executes every operation in To_Do_List.

It is possible that a replica has not yet received one or more operations included in the superset, in which case, the replica asks for retransmission from one or more replicas that reported the operations. Upon receiving a retransmitted request, the replica must verify that it is indeed the missing request that matches the operation record, *i.e.,* the retransmitted request must have the same t, c, and the same digest d as those in the record. This is to prevent a faulty replica from causing state divergence by disseminating conflicting requests. All missing operations are executed as soon as the corresponding requests are received and verified.

When a replica has received and executed all missing operations included in the superset, $Super\_Set\_Ops$, and has rolled back the operations that are in $Undo\_List$ and that it has executed, if any, its operation history would consists of only those in $Super\_Set\_Ops$, as shown in Figure 1. Then it takes a checkpoint of its state and broadcasts a checkpoint message to all other replicas. The replica can now resume executing new requests from clients.

Upon receiving $2f + 1$ checkpoint messages from different replicas (including the one it has sent), the checkpoint becomes stable and the replica can garbage collect all logged messages and control records up to the checkpoint. Note that a replica cannot garbage the logged messages as soon as it has taken a checkpoint because it may need to respond to retransmission requests for missing messages at other replicas.

C. Optimizations

1) Client-assisted state synchronization: A server replica could piggyback its logged operation history with each reply sent to the client. The client aggregates such operation histories received from server replicas, and in turn piggybacks them with its requests. A very similar mechanism has been used in [25] to reflect message ordering information from one replica to other replicas via the clients. This mechanism facilitates the discovery of missed requests and the detection of conflicting requests without explicit state synchronization. A potential disadvantage of this approach is the increased message size of the requests and replies. A tradeoff can be made on the amount of operation history information included in each message and the interval of state synchronization.

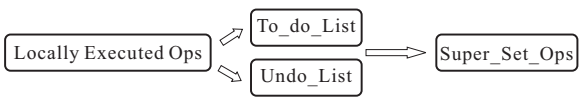

Fig. 1. During state synchronization, each replica ensures that its operation history contains only those in $Super\_Set\_Ops$ via executing operations in To_Do_List and undoing operations in $Undo\_List$, starting with its local history.

2) Non-blocking state synchronization: For the state synchronization mechanism described in Section V-B2, a replica would stop executing new requests until the synchronization for the round is completed. This blocking behavior is obviously undesirable. The mechanism can be modified to avoid blocking.

For non-blocking state synchronization, a replica would continue accepting and executing new valid requests during the synchronization. After it has collected $2f + 1$ totally ordered agreement messages, the replica would proceed to building the superset of operations as usual. Because the replica does not block after sending its agreement message, it may have executed one or more update requests that are not included in the superset, $Super\_Set\_Ops$. The replica needs to identify such operations and temporarily undos them before it takes a checkpoint. Once it takes a checkpoint, the replica re-execute the operations again.

D. Proof of Correctness

We now prove that our Byzantine faulty tolerance mechanisms satisfy the correctness properties laid out in section V-A.

P1. If a nonfaulty client issues a request, it will eventually receive $2f + 1$ matching replies needed to complete the request. Furthermore, if the request is an update request, it will eventually be executed at all nonfaulty replicas.

Proof: A server replica executes a client's request immediately unless it is in the middle of a round of state synchronization. The synchronization round will eventually terminate when the replica has collected $2f+1$ totally ordered agreement messages from different replicas. A nonfaulty client would repeatedly retransmit its request to all replicas until it has collected $2f + 1$ matching replies.

For an update request, as long as all nonfaulty server replicas receive and executed the request, the client is guaranteed to receive matching replies from a quorum $R1$ of $2f + 1$ replicas. Of course, up to f faulty server replicas might decide to execute the request as well. Hence, $R1$ may contain up to f faulty replicas. During the next round of state synchronization, every nonfaulty replica would collect agreement messages from a quorum $R2$ of $2f+1$ replicas. $R1$ and $R2$ must intersect in at least $f + 1$ replicas and one of them must be nonfaulty. This nonfaulty replica ensures the dissemination of the update request to all nonfaulty replicas.

For a read-only request, a client might not be able to collect $2f+1$ matching replies initially if a Byzantine faulty client has issued conflicting requests previously. However, the nonfaulty

client that issued the read-only request will always be able to collect $2f+1$ replies from different replicas. If there exist non-matching replies from the set of replies it has collected, the client would demand a round of state synchronization. When the round of on-demand state synchronization is completed, the client is guaranteed to receive $2f + 1$ matching replies because the states of nonfaulty replicas are converged. ■

P2: The values contained in a reply to a read-only request reflect the state changes of all requests that causally precede it.

Proof: An update request causally precedes a read-only request issued by a client c in either of the two scenarios:

1) The update request is issued by the same client c prior to the read-only request.
2) The update request is issued by another client c' and its state changes are read by another read-only request $ro1$ issued by the same client c prior to the current read-only request $ro2$.

In the first scenario, the client c must have received matching replies from a quorum $R1$ of $2f + 1$ replicas for the update request prior to the sending of the read-only request. Subsequently, the client must also received $2f + 1$ matching replies from a quorum of replicas $R2$ for the read-only request. Because there are $3f + 1$ server replicas, $R1$ and $R2$ must intersect in at least $f + 1$ replicas, which in turn means that at least one nonfaulty replica has executed first the update request and then the read-only request. This ensures the proper causality of requests processing.

For the second scenario, because we assume that the reply to $ro1$ reflects the state changes caused by the update request issued by client c', a quorum $R3$ of $2f + 1$ server replicas must have executed both the update request and the read-only request $ro1$. For the next read-only request $ro2$, client c would wait until it has collected a set of $2f + 1$ matching replies. This means that there exists a quorum $R4$ of $2f + 1$ replicas that executed $ro2$. $R3$ and $R4$ must intersect in at least $f + 1$ replicas, and one of them must be a nonfaulty replica. This nonfaulty replica ensures that the causal dependency is respected. This proves that our Byzantine fault tolerance mechanisms ensure property P2. ■

P3. If a faulty client issues conflicting update requests to different nonfaulty server replicas, the state changes of at most one of the requests will be propagated to other nonfaulty clients.

Proof: The state changes at the server replicas may be propagated to a client when the client issues a read-only request. The client would not accept a reply to its read-only request until it has collected $2f + 1$ matching replies from different replicas. If prior to this read-only request, a faulty client issued conflicting update requests to different replicas, we can deduce that at least $2f+1$ of the replicas have received consistent requests from the faulty client. Otherwise, they could not have generated matching replies to the later read-only request.

We prove the property P3 by contradiction. Assume that the state changes caused by two conflicting update requests with identical timestamp t and client id c, $rw1$ and $rw2$, respectively, have been propagated to one or more nonfaulty clients. This implies that a quorum $R1$ of $2f+1$ server replicas have executed $rw1$, and a quorum $R2$ of $2f+1$ server replicas have also executed $rw2$. The two quorums $R1$ and $R2$ must intersect in at least $f+1$ replicas. Hence, one of them must be a nonfaulty server replica. This means that the nonfaulty replica has accepted and executed both $rw1$ and $rw2$ that contain the same timestamp t from the same client c, which is impossible. This proves that our mechanisms satisfy property $P3$ ■

VI. Implementation and Performance

We have implemented our Byzantine fault tolerance mechanisms together with a shopping cart application constructed using the U-Set CRDT (as described in [17]) in Java. A U-Set consists of two grow-only sets, one for adding objects to the shopping cart, the other for removing objects from the shopping cart. The latter set is often referred to as the tombstone set. The U-Set ensures that the "add" and "remove" operations on the same shopping cart are commutative (the "add" and "remove" operations on different shopping carts are apparently commutative). In addition to the "add" and "remove" operations, the shopping cart service allows a client to query the content of a shopping cart via a read-only operation. The optimization mechanisms outlined in Section V-C are not yet implemented due to their complexity.

We choose to use the UpRight library [7] as the agreement cluster. UpRight is an open source library implementing the well-known Zyzzyva Byzantine fault tolerance algorithm [6]. The library offers highly efficient Byzantine fault tolerant total ordering service during normal operation.

The evaluation of the runtime performance of our mechanisms is carried out in a testbed that consists of 14 HP BL460c blade servers and 18 HP DL320G6 rack-mounted servers connected by a Cisco 3020 Gigabit switch. Each BL460c blade server is equipped with two Xeon E5405 processors and 5 GB RAM, and each DL320G6 server is equipped with a single Xeon E5620 processor and 8GB RAM. The 64-bit Ubuntu Linux server operating system is run at each node.

The server is replicated with 4 replicas to tolerate a single faulty replica (*i.e.,* $f = 1$). The agreement cluster is also configured to tolerate a single faulty node. The number of concurrent clients varies from 1 to 30. To simulate some substantial computation for each request at the server, we inject $0.5ms$ execution time in the form of busy loops. We experimented with three different system configurations: (1) unreplicated client-server application as it is (referred to as unreplicated); (2) the server is replicated, and all requests are totally ordered before they are executed, possibly concurrently (referred to as total ordering); and (3) the server is replicated, and our mechanisms are used with various synchronization periods. For all configurations, the end-to-end latency is measured at the client, and the system throughput is measured at the server replicas.

The end-to-end latency and system throughput results for all three configurations are shown in Figure 2. It can be seen

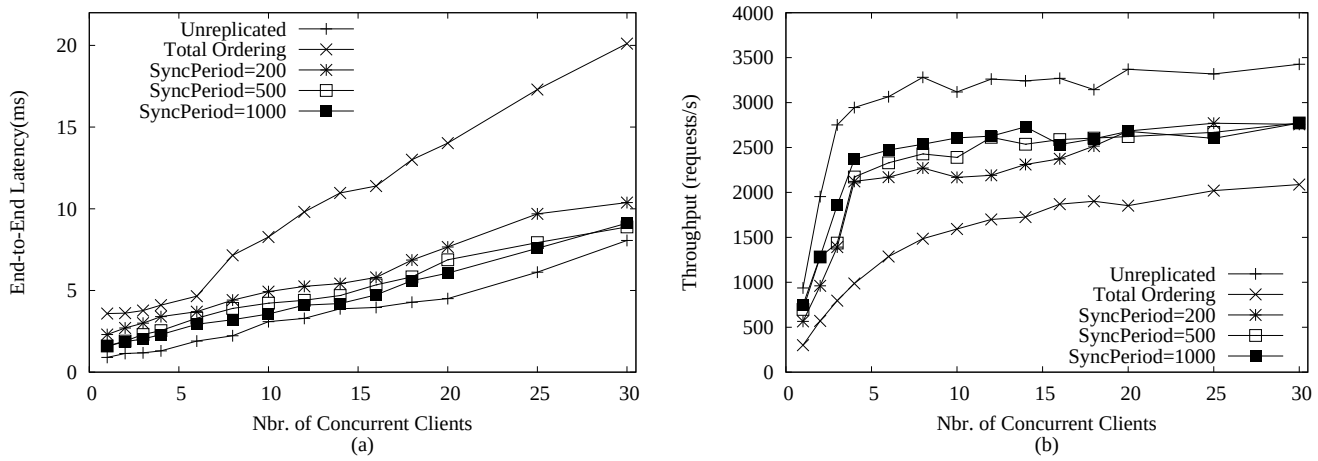

Fig. 2. Performance results for various configurations as indicated in the figure.

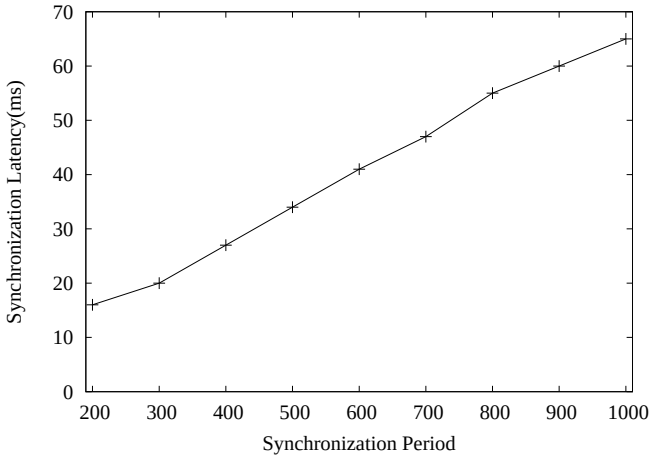

Fig. 3. State synchronization latency.

that our mechanisms reduced the peak system throughput only by about 20% (2800 vs. 3400 requests/sec) compared with the unreplicated configuration. This is significantly better than the performance of the total ordering configuration (*i.e.,* the approach employed in [8]), which reduces the throughput by about 40% (2080 vs. 3400 requests/sec). The average end-to-end latency overhead incurred by our mechanisms is even less prominent. As shown in Figure 2(a), our mechanisms incur nearly constant (about 1-2ms) additional latency for each request regardless of the number of concurrent clients and the synchronization periods, while the overhead incurred by the total ordering approach is not only higher, but it increases roughly linearly with respect to the number of concurrent clients.

The primary cost of our mechanisms are due to state synchronization. During a round of state synchronization, requests are queued but not executed, hence, the average system throughput is inevitably reduced. It is interesting to note from Figure 2(b) that the peak throughput for different synchronization periods (from 200 to 1000 requests per synchronization) remain roughly the same. It suggests that the cost of state synchronization for larger synchronization period is linearly higher, which is indeed the case as shown in Figure 3. This is not surprising because the number of operation records need to be exchanged increases linearly with the size of synchronization periods.

VII. Related Work

Byzantine fault tolerance has been of great research interest for the past several decades [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Many of the prominent research work (*e.g.,* [5], [6]) have focused on developing Byzantine fault tolerance algorithms for generic state-machine replication. Recently, researchers start to pay attention to ways of optimizing the runtime performance (system throughput in particular) by exploiting application semantics [8], [11], [12]. It was recognized that independent requests are commutative, and therefore, can be executed concurrently. It has been shown that this improvement can dramatically increase the system throughput.

In [17], Shapiro et al. presented a comprehensive study on CRDTs and pointed out the feasibility and benefits of using CRDTs to construct highly concurrent replicated distributed services. All update requests to a replicated service implemented using CRDTs are commutative. The discussions were based on the crash-fault model, similar to other works on CRDTs [14], [15], [16]. In this paper, we extended their study by using a Byzantine fault model.

Optimistic replication has long been studied and used in practical systems (see [20] for a comprehensive survey on the subject). Optimistic replication is also in-line with the insight from the CAP theorem [2] where temporary inconsistency must be tolerated for better availability and network partition tolerance. CRDTs were proposed as a new way to ensure the convergence of the replica states instead of operational transformation [21], which was shown that many such algorithms were incorrect in a decentralized architecture [22]. To the best

of our knowledge, our work is the first to achieve Byzantine fault tolerant optimistic replication using CRDTs.

Finally, the work in [26] appears to be very closely related ours because it also focuses on Byzantine fault tolerance for commutative operations. However, the approach in [26] is drastically different from ours. In [26], a Byzantine fault tolerant generic broadcast protocol was introduced to handle both commutative and non-commutative requests. It takes a conservative approach that is aimed for strong replica consistency and requires the use of $5f + 1$ replicas to tolerate up to f faulty replicas. Our approach is based on optimistic replication with eventual consistency guarantee and requires fewer replicas to tolerate the same number of faulty replicas. In particular, our approach does not incur any additional communication step during the critical path of request handling except during periods of state synchronization.

VIII. Conclusion

In this paper, we presented a study on Byzantine fault tolerant optimistic replication of services with commutative operations. Specifically, we assume that the services are constructed using conflict-free replicated data types. The insight to the problem was obtained via a comprehensive threat analysis on such a system. We designed a set of lightweight mechanisms that aim to achieve eventual replica consistency and preserve the causality of the system despite Byzantine faulty clients and Byzantine faulty server replicas. We formulated a set of correctness properties for the system studied and proved that our mechanisms satisfy the properties. We also built a proof-of-concept shopping cart application using CRDTs and implemented the set of Byzantine fault tolerance mechanisms. The evaluation of the system showed that indeed our mechanisms exhibit superior performance compared with an alternative approach.

Acknowledgment

This research is partially supported by a Dissertation Research Award from Cleveland State University (for the first author).

References

[1] W. Zhao, *Building Dependable Distributed Systems*. Wiley-Scrivener, 2014.

[2] E. A. Brewer, "Pushing the cap: Strategies for consistency and availability," *IEEE Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[3] R. Ramakrishnan, "Cap and cloud data management," *IEEE Computer*, vol. 45, no. 2, pp. 43–49, 2012.

[4] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, 1982.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007.

[7] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 277–290.

[8] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

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

[10] W. Zhao, "Design and implementation of a Byzantine fault tolerance framework for web services," *Journal of Systems and Software*, vol. 82, no. 6, pp. 1004–1015, June 2009.

[11] H. Chai, H. Zhang, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Toward trustworthy coordination for web service business activities," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 276–288, 2013.

[12] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Trustworthy coordination for web service atomic transactions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1551–1565, 2012.

[13] H. Chai and W. Zhao, "Byzantine fault tolerance for session-oriented multi-tiered applications," *Int. J. of Web Science*, vol. 2, no. 1/2, pp. 113–125, 2013.

[14] C. Baquero and F. Moura, "Using structural characteristics for autonomous operation," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 4, pp. 90–96, Oct. 1999.

[15] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 395–403.

[16] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354 – 368, 2011.

[17] M. Shapiro, N. Preguia, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Dfago, F. Petit, and V. Villain, Eds. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400.

[18] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computer Survey*, vol. 22, no. 4, pp. 299–319, 1990.

[19] H. Chai and W. Zhao, "Byzantine fault tolerance as a service," in *Computer Applications for Web, Human Computer Interaction, Signal and Image Processing, and Pattern Recognition*, ser. Communications in Computer and Information Science, T.-h. Kim, S. Mohammed, C. Ramos, J. Abawajy, B.-H. Kang, and D. Slezak, Eds. Springer Berlin Heidelberg, 2012, vol. 342, pp. 173–179.

[20] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005.

[21] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, Mar. 1998.

[22] G. Oster, P. Urso, P. Molli, and A. Imine, "Proving correctness of transformation functions in collaborative editing systems," INRIA, Rapport de recherche RR-5795, 2005. [Online]. Available: http://hal.inria.fr/inria-00071213

[23] J. P. Anderson, "Computer security technology planning study," USAF, Tech. Rep., 1972.

[24] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, June 1975.

[25] W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Low latency fault tolerance system," *The Computer Journal*, vol. 56, no. 6, pp. 716–740, 2013.

[26] P. Raykov, N. Schiper, and F. Pedone, "Byzantine fault-tolerance with commutative commands," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, A. Fernndez Anta, G. Lipari, and M. Roy, Eds. Springer Berlin Heidelberg, 2011, vol. 7109, pp. 329–342.