Byzantine Chain Replication

Robbert van Renesse, Chi Ho, and Nicolas Schiper

Cornell University[*]
Ithaca, NY, USA
{rvr,chho,nschiper}@cs.cornell.edu

Abstract. We present a new class of Byzantine-tolerant State Machine Replication protocols for asynchronous environments that we term *Byzantine Chain Replication*. We demonstrate two implementations that present different trade-offs between performance and security, and compare these with related work. Leveraging an external reconfiguration service, these protocols are not based on Byzantine consensus, do not require majority-based quorums during normal operation, and the set of replicas is easy to reconfigure.

One of the implementations is instantiated with $t+1$ replicas to tolerate t failures and is useful in situations where perimeter security makes malicious attacks unlikely. Applied to in-memory BerkeleyDB replication, it supports 20,000 transactions per second while a fully Byzantine implementation supports 12,000 transactions per second—about 70% of the throughput of a non-replicated database.

1 Introduction

Byzantine-tolerant State Machine Replication (BSMR) is the only known generic approach to making applications (servers, routing daemons, and so on) tolerate arbitrary faults beyond crash failures in an asynchronous environment [1]. Various studies in complex systems have shown that crash failures constitute a minority of failures [2, 3], while trends in hardware increase the probability of transient hardware errors such as bit flips [4–6]. Worse yet, most replication protocols deployed in cloud centers provide weak consistency guarantees, meaning that they *introduce* inconsistencies even if there are no faults [7, 8]. Protocols such as Primary-Backup [9] and Chain Replication [10] make strong assumptions about failure detection that are easily violated in datacenter settings while conservative timeouts result in long recovery times.

While BSMR addresses all these problems, to the best of our knowledge there is no deployment of BSMR in today's datacenters. There are good reasons for this:

- Traditional BSMR require $3t+1$ replicas in order to tolerate t faults [11], whereas primary-backup replication protocols require only $t+1$ replicas [9].

[*] This work was supported in part by DARPA grants FA8750-10-2-0238, FA9550-11-1-0137 , FA8750-11-2-0256, NSF grants 1040689, 1047540, 54083, and DOE grant DE-AR0000230.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 345–359, 2012.

Also, the protocols may require expensive cryptographic operations and significant network bandwidth, further increasing cost;
- Significant progress has been made at reducing the cost of BSMR [12–14]. However, these protocols are complex [15]. Even basic (crash-tolerant) Paxos replication is difficult to implement and debug [16]. Multi-purpose Paxos and BSMR libraries have not proved successful, and few support basic operations such as reconfiguring the locations of the protocol participants, important to practical deployment.

Much of the difficulty stems from replicated services being designed to be *stand-alone*, but in practice the configuration of replicated services is usually managed by an external scalable configuration service. The configuration service is used only in the face of failures. Exploiting this, we can build replication protocols that do not use quorums in the steady-state when there are no failures. This reduces the number of replicas necessary, not only increasing efficiency but also robustness in the face of limited sources of diversity.

Our protocols are based on Chain Replication [10], which is increasingly being used in scalable fault tolerant systems [17–20]. Chain Replication also uses an external configuration service, but cannot tolerate even general crash failures as the protocol depends on accurate failure detection, an overly strong requirement in today's datacenters. In the face of a mistaken failure detection, a chain may split and inconsistent results may be returned to clients. To reduce the likelihood of such problems, timeouts used for failure detection have to be set conservatively (values that exceed one minute are not uncommon in practice), meaning that in the face of an actual crash it takes a long time before remedial action can be taken.

In this paper we make the following contributions:

- We present a new class of highly reconfigurable Chain Replication protocols that are easily reconfigurable, do not require accurate failure detection, and are able to tolerate Byzantine failures.
- We prove the correctness of this class of protocols;
- We present a Byzantine Chain Replication protocol called *Shuttle*;
- We briefly describe two simple implementations of Shuttle: one that can tolerate Byzantine failures in their full generality, and one that tolerates "accidental failures" such as crashes, bit flips, concurrency bugs, and so on;
- We present a performance evaluation of Byzantine Chain Replication applied to the popular BerkeleyDB database service [21];
- We compare the complexity with that of related protocols.

The Shuttle implementation that can tolerate arbitrary failures uses, in steady state, $2t + 1$ replicas, HMAC signatures, and has a message overhead of $2t + 2$ messages per operation. The implementation that is designed to deal with crash failures as well as bit flips and Heisenbugs uses only $t+1$ replicas, and its overhead is essentially the same as Chain Replication for update operations (using CRC checksums and $t + 2$ messages per operation). However, read-only operations are as expensive as update operations, whereas in Chain Replication read-only operations are only 2 messages.

Section 2 presents *Byzantine Chain Replication*, and Section 3 shows the Shuttle protocol. Implementation details are the topic of Section 4. Section 5 evaluates performance characteristics. A comparison with related work is described in Section 6. Section 7 concludes.

2 Byzantine Chain Replication

This section presents a class of replication protocols, rather than a specific protocol. The class is characterized by the state that correct replicas keep and how they order and persist *operations*. For simplicity, we consider a single object. We model the state of the object as a finite history $\mathcal{H}$ of operations, which is a set of $\langle s, o\rangle$ pairs, where s is a *slot number* and o an operation. It has the following property:

$$\forall s, o, o' : \langle s, o\rangle \in \mathcal{H} \wedge \langle s, o'\rangle \in \mathcal{H} \Rightarrow o = o' \tag{1}$$

That is, each slot can have at most one operation assigned to it.

Initially, $\mathcal{H}$ is empty. The only allowed transition on the object is to add an operation to the history at a particular slot (transitions are atomic):

```
specification Object:
    transition apply($s, o$):
        precondition: $\nexists o' : \langle s, o'\rangle \in \mathcal{H}$
        action: $\mathcal{H} := \mathcal{H} \cup \{\langle s, o\rangle\}$
```

Note that the history does not have to be filled sequentially, and thus "holes" in the history are allowed. However, if a running state is maintained, operations should be applied in the order of their slot number in the history.

To make the object highly available, we will replicate it and dynamically change the configuration in order to deal with failures. Similar approaches have been proposed for crash failures only [22, 23]. For now suppose there is an unbounded sequence of configurations $\mathcal{C} = C_1 :: C_2 :: C_3 :: ....$ The boolean function $succ(C, C')$ evaluates to `true` if and only if C' follows C directly in $\mathcal{C}$. Each configuration C_i is an ordered *chain* of replicas (see Chain Replication [10]), and we assume that the replicas of any two different configurations are disjoint. Given two replicas ρ and ρ' in the same configuration, we write $\rho \prec \rho'$ if ρ precedes ρ' in the chain.

Any replica ρ can create *statements* of the form $\langle d\rangle_\rho$ where d is arbitrary data. For now, assume that d is data signed by ρ using public key cryptography; more efficient schemes will be described later. If ρ is correct, only ρ can create those statements. We say that ρ *says* d. A global state variable *statements* contains all statements of correct replicas, but may also contain statements of faulty replicas.

```
specification Chain:
    transition orderCommand($\rho, C, s, o$):
        precondition:
            $\rho \in C \wedge \rho.mode = \texttt{ACTIVE} \wedge$
            $\forall \rho' \in C : \rho' \prec \rho \Rightarrow \langle \texttt{order}, s, o \rangle_{\rho'} \in statements \wedge$
            $\nexists o', v', \rho' : \langle s, o', \rho', C, v' \rangle \in \rho.history$
        action:
            $\rho.history := \rho.history \cup \left\{ \left\langle s, o, \rho, C, \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho \} \right\rangle \right\}$
            $statements := statements \cup \{ \langle \texttt{order}, s, o \rangle_{\rho} \}$

    transition becomeImmutable($\rho$):
        precondition:
            $\rho.mode = \texttt{ACTIVE}$
        action:
            $\rho.mode := \texttt{IMMUTABLE}$
            $statements := statements \cup \{ \langle \texttt{wedged}, \rho.history \rangle_{\rho} \}$

    transition switchConfig($C, C', H$):
        precondition:
            $succ(C, C') \wedge \nexists h : \langle \texttt{inithist}, C', h \rangle \in statements \wedge$
            $H \subseteq statements \wedge$
            $\exists Q \in \mathcal{Q}_C : |H| = |Q| \wedge$
                        $\forall \rho \in Q : \exists h : \langle \texttt{wedged}, h \rangle_{\rho} \in H \wedge validHist(h, \rho, C)$
        action:
            $hist := \{ \left\langle s, o, \max(C'), C', \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C' \} \right\rangle \mid$
                        $\exists h, v : h \in H \wedge \langle s, o, v \rangle \in h \wedge$
                        $\forall h', v', o' : h' \in H \wedge \langle s, o', v' \rangle \in h' :\Rightarrow |v| \geq |v'| \}$
            $statements := statements \cup \{ \langle \texttt{inithist}, C', hist \rangle_{\Omega} \}$

    transition becomeActive($\rho, C, hist$):
        precondition:
            $\rho \in C \wedge \rho.mode = \texttt{PENDING} \wedge \langle \texttt{inithist}, C, hist \rangle_{\Omega} \in statements$
        action:
            $\rho.history := hist$
            $\rho.mode := \texttt{ACTIVE}$
```

Fig. 1. State transitions allowed in Chain Replication

A correct replica ρ in configuration C has the following local state:

- $\rho.mode$: PENDING, ACTIVE, or IMMUTABLE. Initially all replicas in C_1 are ACTIVE, while replicas in all other configurations are PENDING;
- $\rho.history$: a set of *order proofs*. An order proof is a tuple of the following form:

$$\left\langle s, o, \rho, C, \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho \} \right\rangle . \quad (2)$$

Here s is a slot number, o an operation, and each $\langle \texttt{order}, s, o \rangle_{\rho'}$ is an *order statement* said by $\rho' \in C$. An order proof contains order statements from all replicas that precede ρ in C and from ρ itself. $\rho.history$ cannot contain conflicting order proofs, that is, order proofs for the same slot number but different operations.

In addition to replicas, there is an oracle Ω that can sign so-called `inithist` statements. Figure 1 shows what transitions are allowed in Chain Replication by correct replicas and by Ω:

1. `orderCommand`(ρ, C, s, o): Any active replica ρ in configuration C can say $\langle \texttt{order}, s, o\rangle_\rho$ if each preceding replica in C, if any, has done likewise and there is no conflicting operation for s in its history. ρ also adds a new order proof to its history.
2. `becomeImmutable`(ρ): An active replica ρ can suspend updating its history by becoming immutable at any time. Typically it will do this only if one or more of its peer replicas are suspected of being faulty. The replica signs a `wedged` statement to notify Ω that it is immutable and what its history is.
3. `switchConfig`(C, C', H): The oracle Ω waits for a set H of *valid* histories from a quorum of replicas in C. $\mathcal{Q}_C$ is the set of quorums defined for configuration C. A valid history contains at most one order proof per slot. The oracle then issues an `inithist` statement for configuration C' with the order proofs of maximal size for each slot s in the histories in H. $\max(C')$ is the last replica (the "tail") on chain C'. The oracle can issue at most one `inithist` statement per configuration.
4. **transition** `becomeActive`(ρ, C, hist): A pending replica ρ in configuration C can become active if the oracle has issued an `inithist` statement for C.

We require that each quorum $Q \in \mathcal{Q}_C$ contain at least one *honest* replica. An honest replica ρ is defined as follows: if issued, $\langle \texttt{wedged}, h\rangle_\rho$ includes all order proofs corresponding to all $\langle \texttt{order}, s, o\rangle_\rho$ statements that ρ issues. In other words, an honest replica cannot truncate its history, nor can it issue new `order` statements after it has become immutable and issued a `wedged` statement.

Clearly, correct replicas are honest, but replicas that suffer only crash failures are honest as well. Later we will argue that replicas that suffer from bit flips and other types of non-malicious failures can be transformed into honest replicas. Replicas that are not honest are easily identified: the combination of an $\langle \texttt{order}, s, r\rangle_\rho$ statement and a $\langle \texttt{wedged}, h\rangle_\rho$ statement that does not contain an order proof for $\langle s, r\rangle$ constitutes a *proof of misbehavior*.

The transitions specify what actions are safe, but not when or in what order to do them.[1] In other words, the system is *asynchronous*. For liveness, we assume that there is always eventually a configuration in which all replicas are correct and do not become immutable.

[1] The transition specifications are not pseudo-code, and should not be confused with implementation. For example, the code corresponding to the `orderCommand` transition would have to check that the order proofs of preceding replicas are signed correctly.

Safety

We show safety through a sketch of a refinement mapping of Specification Chain to Specification `Object`. The statements that are made by the (correct and faulty) replicas map to the state of the object as follows:

$$\mathcal{H} = \{\langle s, o\rangle \mid \exists C \in \mathcal{C} : \forall \rho \in C : \langle \texttt{order}, s, o\rangle_\rho \in \mathit{statements}\} \tag{3}$$

Thus an $\langle s, o\rangle$ pair is in the object history if all replicas of a configuration have issued `order` statements for that pair. We call the $\langle s, o\rangle$ pairs in $\mathcal{H}$ thus defined *persistent*. For this *refinement mapping* to be well-defined, we need to show that Equation 1 holds. We say that $\langle s, o\rangle$ *persists in* C if $C \in \mathcal{C}$ and $\forall \rho \in C : \langle \texttt{order}, s, o\rangle_\rho \in \mathit{statements}$.

Lemma 1. *If $\langle s, o\rangle$ persists in C and the precondition of* `orderCommand`(ρ', s, o') *holds, where ρ' is a correct replica in configuration C' that follows C in $\mathcal{C}$ (not necessarily directly), then $o = o'$.*

Proof. (Sketch) By definition of *persists*, all replicas in C issue an $\langle \texttt{order}, s, o\rangle$ statement. First assume C' is the configuration that directly follows C. Because the precondition of `orderCommand`(ρ', s, o') holds, ρ' is active, and thus the oracle Ω must have issued an `inithist` statement for C'. Thus a quorum of replicas in C must have issued `wedged` statements and Ω obtained order proofs of at least one honest replica in C. The oracle thus obtained an order proof for $\langle s, o\rangle$ from some honest replica ρ. It is not possible for faulty replicas to create a larger order proof for a different $\langle s, o'\rangle$, as that would require that ρ says $\langle \texttt{order}, s, o'\rangle_\rho$ for the larger order proof. This cannot happen because ρ is honest. So if there is a larger order proof, it must be for $\langle s, o\rangle$.

Iteratively, all correct replicas in all configurations that follow C will have an order proof for $\langle s, o\rangle$. As a correct replica never issues an order statement that conflicts with an order proof in its history, the lemma holds.

Theorem 1. *For all s, o, o', C, and C', if $\langle s, o\rangle$ persists in C and $\langle s, o'\rangle$ persists in C', then $o = o'$.*

Proof. (Sketch) By contradiction, consider some s, o, o', C, and C' such that $o \neq o'$. All correct replicas in C ordered $\langle s, o\rangle$, while all correct replicas in C' ordered $\langle s, o'\rangle$. There are two cases:

- $C = C'$: this means that all correct replicas in C ordered both $\langle s, o\rangle$ and $\langle s, o'\rangle$, which is impossible as a correct replica issues at most one `order` statement per slot number.
- $C \neq C'$: wlog., assume C' follows C in $\mathcal{C}$. Because all correct replicas in C' ordered $\langle s, o'\rangle$, they must each have issued `order` statements for $\langle s, o'\rangle$ and thus underwent `orderCommand` transitions for $\langle s, o'\rangle$. By Lemma 1, $o = o'$.

This demonstrates that Equation 1 holds under any Chain transition.

An important question is how to enforce that each quorum in $\mathcal{Q}_C$ contains at least one honest replica. One way is to have each quorum contain at least $t+1$ replicas and thus at least one correct replica. To guarantee liveness each configuration would have to have at least $2t+1$ replicas, or no sufficient number of replicas might issue a `wedged` statement for the precondition of `switchConfig` to become true.

Another option is to assume that all replicas are honest. Under this assumption, quorums can be singletons and, for liveness, each configuration would have to have at least $t+1$ replicas. Such an assumption could be reasonable if malicious failures are unlikely and each replica maintain a checksum of its history, reporting a failure when it detects that its history is compromised. Alternatively, automated approaches that transform hardware errors into crash failures could be used [5, 6]. There is no way to prevent malicious replicas from issuing truncated histories. However, if they do so they expose themselves as provably faulty.

3 Shuttle: A Byzantine Chain Replication Protocol

Shuttle is a BSMR protocol based on Byzantine Chain Replication. Each replica maintains a running state in addition to the local history of order proofs (we will show later how the history can be truncated periodically). A centralized configuration service called *Olympus* implements the oracle Ω and generates a series of configurations, issuing `inithist` statements (signed by a private key held by Olympus) for each such configuration. Any modern datacenter will contain a configuration service similar to the Olympus and can maintain the configurations of many objects. A configuration statement includes the sequence number of the configuration and an ordered list of replica identifiers.

Olympus generates a new configuration upon receiving a `reconfiguration-request` statement. To reduce the efficacy of trivial Denial-of-Service attacks on the object, Olympus only accepts reconfiguration requests that are sent by replicas in the current configuration or accompanied by a proof of misbehavior (such as the aforementioned conflicting `wedged` and `order` statements from the same replica, or any other set of conflicting statements from the same replica that Olympus recognizes). Olympus does not have to allocate a fresh set of replicas for each configuration—doing so would lead to significant overhead as state is transfered from old replicas to new replicas—however, reused replicas need to be made aware of their new configuration. For liveness, we assume that the Olympus eventually creates a configuration of correct replicas that do not issue reconfiguration requests.

3.1 Proofs

Before describing the Shuttle protocol, we generalize the notion of a proof. A *proof* of d is a tuple:

$$\left\langle d, C, \rho, \{\langle d\rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho\}\right\rangle. \tag{4}$$

where $C \in \mathcal{C}$ and $\rho \in C$. Typically, ρ is the issuer of the proof. An `order` proof is an example of a proof, that is, a proof of ρ ordering $\langle s, o\rangle$. We will say that a proof is *complete* if ρ is the tail of the chain in C, that is, all replicas in C have signed d.

3.2 Failure-Free Case

Suppose a client wants to execute an operation o and obtain a result. The client first obtains the current configuration from Olympus, and sends o to the head of the chain. The head orders incoming operations by assigning increasing slot numbers to them, and creating *shuttles* that are sent along the chain.

A shuttle contains two proofs: an order proof for $\langle s, o\rangle$ and a result proof for the result of o. Suppose that the head assigns o to slot s. Each replica ρ, including the head, does the following:

1. checks the validity of the order proof in the shuttle;
2. applies o to its running state and obtains a result r;
3. adds $\langle \texttt{order}, s, o\rangle_\rho$ to the order proof (the replica undergoes an `orderCommand` transition);
4. adds $\langle \texttt{result}, o, \mathcal{S}(r)\rangle_\rho$ to the result proof, where $\mathcal{S}$ is a cryptographic hash function;
5. forwards the shuttle to the next replica if any.

After the tail replica adds its `order` statement to the shuttle, the order proof is complete and $\langle s, o\rangle$ is persistent, and the tail forwards the result proof to the client along with the result r itself. The client believes the result if $\mathcal{S}(r)$ corresponds to all `result` statements in the result proof.

The tail also returns the shuttle with the completed proofs to the head along the chain in the reverse order. We will refer to this as the *result shuttle*. Each replica caches this shuttle (and the result r itself) in order to deal with potential failures, as described next.

3.3 Dealing with Failures

In the case of a replica or a network failure, a client may not receive a valid response. The client uses a timer to try to detect a failure. (The value of the timer does not affect safety and can thus be set aggressively in order to detect failures quickly—however, if set too aggressively the overhead of reconfigurations would negatively affect performance.) When the timer expires, the client retransmits its operation to all replicas in the chain. If a (correct) replica that receives the request has the result shuttle cached, it returns the result along with the result proof to the client. If the replica is immutable, it responds to the client with an `error` statement, in which case the client has to retrieve the latest configuration from Olympus and try again. In all other cases, the replica forwards the request to the head (if it is not the head itself), and starts a timer.

The head, if correct and upon receiving a retransmission (either directly from the client or indirectly through one of the peer replicas), distinguishes three cases:

1. it has cached the result shuttle corresponding to the operation;
2. it has ordered the operation but is still waiting for the result shuttle to come back;
3. it does not recognize the operation.

In the first case, it sends the cached result to the client. In the second case, it starts a timer. In the third case, it allocates a new slot number, starts the protocol from scratch, and starts a timer. In the latter two cases, if receiving the result before its timer expires, the head cancels its timer and responds to the client. Upon receipt of the result each replica does likewise, canceling its timer if it is still outstanding, and sending the result along with the result proof to the client. Should the timer expire at one of the replicas, then the replica sends a `reconfiguration-request` statement to Olympus.

To reconfigure, Olympus executes the following steps:

1. send signed `wedge` requests to each of the replicas in the current configuration. Correct replicas respond with `wedged` statements (`becomeImmutable` transition);
2. await responses from a quorum of replicas, and construct a history h by selecting the longest order proof for each slot;
3. allocate a new configuration C of replicas and seed those with h and a configuration statement for C. The replicas then become active (`becomeActive` transition).

3.4 Clients

After sending an operation, the client is waiting for a response that may never arrive. For liveness, the client checks periodically with Olympus to see if there is a new configuration, and if so retransmits its operation. Consequently, an operation may be executed more than once. Although not currently implemented, duplicate execution can be prevented if the service keeps track of which operations it has already executed (on a per-client basis) and treats duplicates as no-ops. The results of such operations may no longer be available, as caching results for ever would present a significant storage problem. Note that the problem and solution are the same for non-replicated servers [24].

Shuttle can tolerate Byzantine clients in that correct Shuttle replicas do not assume that clients follow a particular specification. In particular, Byzantine clients cannot compromise the integrity of the replicated object, for example, by sending conflicting operations to different replicas. This is a direct result of the head of the object making all ordering decisions within its configuration. However, Byzantine clients can send bogus operations to the replicas[2] and they can mount Denial-of-Service attacks by sending lots of bogus operations. Shuttle has no specific defense for such attacks, except that client operations are signed and thus the source is easily identified. It is assumed that such clients are shut down using external means.

[2] A faulty head of the chain can introduce bogus operations by itself.

4 Implementation

We have implemented two versions of the Shuttle protocol:

1. *CRC Shuttle*: A version that is intended to deal with unintentional failures such as power failures, Heisenbugs, bit flips, and so on, but all replicas are assumed to be honest. It uses $t + 1$ replicas in each configuration, singleton quorums, and CRC checksums for signatures.
2. *HMAC Shuttle*: A version that tolerates arbitrary failures, and uses $2t + 1$ replicas in each configuration, quorums of size $t + 1$, and signatures based on HMACs.

We do not have space to describe the entire implementations, but sketch some aspects below. For a complete description, see [25].

4.1 HMAC Vectors

In the case of HMAC Shuttle, a replica signs a statement with a vector of HMAC signatures, with one entry for each receiver that may need to receive the statement. It is thus possible that a faulty replica provides valid signatures for some but not for all destinations. Worse, a recipient of a statement with a valid HMAC signature in its entry of the vector cannot necessarily convince other processes.

Each replica ρ_i has a secret signing key $k^{\mathtt{r}}_{ij}$ for each other replica ρ_j and a secret signing key $k^{\mathtt{c}}_{ic}$ for each client c. The inter-replica signing keys $k^{\mathtt{r}}_{ij}$ are known to the Olympus as well, while the replica-to-client signing keys $k^{\mathtt{c}}_{ic}$ are created using the Diffie-Hellman [26] protocol. A `wedged` statement from a replica to the Olympus is signed by an HMAC vector with an entry for each other replica. The Olympus can check this signature as well as all order proofs as it is in possession of all the necessary keys.

In the current implementations, Olympus is a trusted server that is not replicated itself, but it could be replicated using something like PBFT [12]. Since Olympus has the inter-replica signing keys, a Byzantine replica of the Olympus service might try to leak these keys to Byzantine object replicas. It is thus important that outputs of the Olympus replicas go through a voting filter; only if $t + 1$ of the Olympus replicas send a copy of particular message, the filter will let it through.

4.2 Checkpointing

It would be infeasible in practice for each replica to maintain the entire history of operations, let alone pass on this history to new replicas during reconfiguration. Thus each replica actually maintains a checkpoint and a suffix of the history of operations that comes after this checkpoint. Periodically the head initiates a checkpoint by sending a shuttle down the chain with a *checkpoint proof* for a hash of its running state. Each replica ρ adds a $\langle \mathtt{checkpoint}, \mathcal{S}(\mathit{state}) \rangle_\rho$ to the checkpoint proof. The tail returns the completed checkpoint proof along the chain so each replica can remove the corresponding prefix from its history. The latest checkpoint proof along with the corresponding state is part of a `wedged` statement of a replica.

In HMAC Shuttle, it is not necessary for all $2t+1$ replicas to maintain running state [13]. The chain is split into two parts. The first $t+1$ (including the head) maintain the running state, but the final t (including the tail) do not. These *witness replicas* have to sign the order proofs in the shuttles, but they do so without knowing the running state. Witnesses do not sign the result proof as they cannot compute it. A client needs only $t+1$ matching copies of the result in order to accept it, knowing that at least one correct replica signed the result.

5 Experimental Evaluation

In this section, we evaluate the performance of HMAC Shuttle experimentally and compare it against Chain Replication [10], a protocol that tolerates only crash failures and assumes perfect failure detection. We expect the performance of Shuttle CRC to be close to Chain Replication: they both arrange $t+1$ replicas in a chain. Computing CRC checksums, an operation only carried out in Shuttle CRC, has negligible overhead. Both HMAC Shuttle and Chain Replication are implemented in Java. To evaluate the overhead of these protocols, we put them side-to-side with a non-replicated server in a failure-free scenario.

The code is deployed on a cluster of Linux machines connected by a Gigabit switch. Replicas and clients run on dual-core 2.8 Ghz AMD Opterons. We use 2 and 3 machines for the replicas of Chain Replication and HMAC Shuttle respectively ($t = 1$); clients run on a separate machine. All replicas maintain running state and HMACs are generated with the SHA-256 algorithm using 256-bit keys.

Each replica runs a copy of a BerkeleyDB, a transactional key-value store. The store consists of a list of accounts with their corresponding balances. We consider an *update-only* workload where transactions deposit a random amount of money into a randomly selected account. We configured BerkeleyDB to run entirely in memory as disk latencies tend to largely dominate and obfuscate the protocol overheads. In the benchmarks, each client submits 10 transactions in parallel and waits to receive all responses before submitting the next ones. Each client submits 30,000 transactions and the experiments comprised between 1 and 24 clients.

In Figure 2, we present the average number of committed transactions per second (throughput) versus the average latency for Chain Replication (CR) and HMAC Shuttle (HMAC). In the left graph of Figure 2, we include the performance of a single BerkeleyDB instance. In both graphs, we report the standard deviation of the latency, except for the single BerkeleyDB instance. The latter was consistently smaller than for the two other protocols.

Chain Replication has a slightly higher latency than the single BerkeleyDB instance for a given load: with Chain Replication the transaction goes through two servers instead of a single one. Surprisingly, Chain Replication supports up to 20,000 transactions per second, a higher load than a single BerkeleyDB

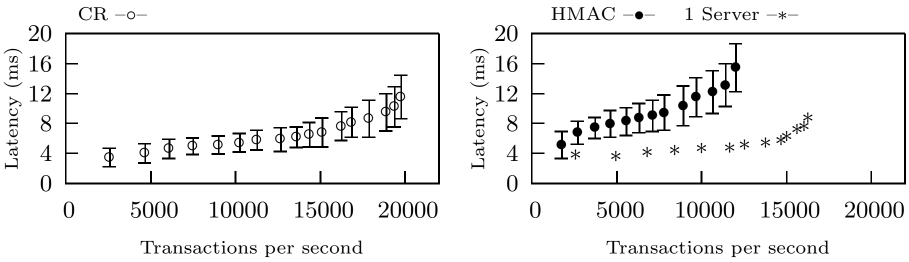

Fig. 2. Throughput vs. Latency of Chain Replication (CR) and HMAC Shuttle (HMAC)

server, which can support about 17,000 transactions per second. It turns out that with Chain Replication the overhead of communication with clients is shared between the head and the tail of the chain—the head receives transactions from clients and the head sends back results. HMAC Shuttle offers good performance for a Byzantine-resilient replication protocol: it supports loads up to 12,000 transactions per second. However, it exhibits higher latencies due to the overhead of verifying and signing statements.

6 Comparison with Prior Work

Many papers have described how to make Byzantine fault tolerance practical. Starting from PBFT [12], various proposals [14, 15, 27–29] aim to reduce latency and increase throughput. Aardvark [30] and Zyzzyvark [31] focus on sustainable performance rather than peak performance. Other proposals focus on reducing the number of full replicas [13, 32, 33].

Table 1 compares various aspects of Shuttle with related work. PBFT is the first Byzantine replication protocol designed for practical use. Zyzzyva [14] is optimized for peak performance. Aliph [15] uses a chain communication pattern, like Shuttle. Zyzzyvark is a recent protocol that tolerates client failures. We consider ZZ [33] the state of the art in reducing replication cost.

We consider the total number of replicas, the number of full replicas, the maximum number of HMAC operations at a replica, the number of message rounds (*aka* message latencies or network latencies), and the effect that Byzantine clients can have on a replicated object. As all of the approaches support batching of multiple requests in order to amortize CPU overhead, we include the effect of batching on the number of crypto operations. In PBFT and ZZ (based on PBFT), a faulty client can trigger reconfigurations in the replicated object. Zyzzyva uses speculative execution, and a faulty client can trigger a rollback that involves multiple rounds and expensive RSA signatures. In Aliph a faulty client can also trigger a rollback.

Table 1. Properties of state-of-the-art BFT replication approaches that tolerate t failures, avoid RSA signatures, and use a batch size b

	PBFT	Zyzzyva	Aliph	Zyzzyvark	ZZ	HMAC	CRC
						Shuttle	Shuttle
total #replicas	$3t+1$	$3t+1$	$3t+1$	$3t+1$	$3t+1$	$2t+1$	$t+1$
#full replicas	$2t+1$	$2t+1$	$3t+1$	$2t+1$	$t+1$	$t+1$	$t+1$
#crypto ops	$2+\frac{8t+1}{b}$	$2+\frac{3t}{b}$	$1+\frac{t+1}{b}$	$2t+\frac{3t}{b}+2$	$2+\frac{10t+3}{b}$	$2+\frac{2t}{b}$	$2+\frac{t}{b}$
#message rounds	4	3	$3t+2$	4	4	$2t+2$	$t+2$
effects of faulty clients	Reconfig.	Rollback	Rollback	None	Reconfig.	None	None

7 Conclusion

Byzantine fault tolerance is becoming increasingly important as we depend more on computer systems, and as those systems have more components that may fail. Byzantine Chain Replication is a new class of replication protocols that needs only few sources of diversity and has modest costs while tolerating a large class of failures. This paper has also presented Shuttle, a simple implementation of a Byzantine Chain Replication protocol, and compared two versions with related work. We find that Byzantine Chain Replication configured with $t = 1$ applied to an in-memory database can support about 70% of the throughput of a non-replicated database, albeit at about twice the latency due to the overhead of checking and verifying HMAC signatures.

References

1. Schneider, F.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
2. Gashi, I., Popov, P., Stankovic, V., Strigini, L.: On Designing Dependable Services with Diverse Off-the-Shelf SQL Servers. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 191–214. Springer, Heidelberg (2004)
3. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In: Proc. of the 21st Symp. on Operating Systems Principles, SOSP 2007, pp. 59–72. ACM (October 2007)
4. Shivakumar, P., Kistler, M., Keckler, S., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Dependable Systems and Networks, DSN 2002, pp. 389–398 (2002)
5. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D.: SWIFT: software implemented fault tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 243–254 (March 2005)

6. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 169–182. Springer, Heidelberg (2010)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proc. of 21st Symposium on Operating Systems Principles (2007)
8. Shafaat, T., Schütt, T., Moser, M., Haridi, S., Ghodsi, A., Reinefeld, A.: Key-based consistency and availability in structured overlay networks. In: Proc. of the 17th Int. Symp. on High-Performance Distributed Computing, HPDC 2008, pp. 235–236. ACM (June 2008)
9. Budhiraja, N., Marzullo, K., Schneider, F., Toueg, S.: The primary-backup approach. In: Mullender, S. (ed.) Distributed Systems, 2nd edn. ACM Press/Addison-Wesley, New York (1993)
10. Van Renesse, R., Schneider, F.: Chain Replication for supporting high throughput and availability. In: 6th Symp. on Operating Systems Design and Implementation, OSDI 2004 (December 2004)
11. Bracha, G., Toueg, S.: Resilient consensus protocols. In: Proc. of the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, pp. 12–26. ACM SIGOPS-SIGACT (August 1983)
12. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proc. of the 3rd Symposium on Operating Systems Design and Implementation, OSDI 1999, New Orleans, LA. USENIX (February 1999)
13. Yin, J., Martin, J., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution in Byzantine fault-tolerant services. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003, Bolton Landing, NY, pp. 253–268 (October 2003)
14. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. 27(4) (2009)
15. Guerraoui, R., Knezevic, N., Quema, V., Vukolic, M.: The next 700 BFT protocols. In: Proc. of the 5th ACM European Conf. on Computer Systems, EUROSYS 2010, Paris, France (April 2010)
16. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proc. of the 26th ACM Symp. on Principles of Distributed Computing, Portland, OR, pp. 398–407. ACM (May 2007)
17. Andersen, D., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: FAWN: A Fast Array of Wimpy Nodes. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles, Big Sky, MT (October 2009)
18. Terrace, J., Freedman, M.: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In: Proc. of the USENIX Annual Technical Conference, USENIX 2009, San Diego, CA (June 2009)
19. Fritchie, S.: Chain replication in theory and in practice. In: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (2010)
20. Escriva, R., Wong, B., Sirer, E.: HyperDex: A distributed, searchable key-value store. In: Proceedings of the SIGCOMM Conference, Helsinki, Finland (August 2012)
21. Olson, M., Bostic, K., Seltzer, M.: Berkeley DB. In: Proc. USENIX Annual Technical Conference (1999)
22. Lamport, L., Malkhi, D., Zhou, L.: Brief announcement: Vertical Paxos and Primary-Backup replication. In: Proc. of the 28th ACM Symp. on Principles of Distributed Computing (August 2009)

23. Birman, K., Malkhi, D., Van Renesse, R.: Virtually Synchronous Methodology for Dynamic Service Replication. Technical Report MSR-TR-2010-151, Microsoft Research (2010)
24. Saltzer, J., Reed, D., Clark, D.: End-to-end arguments in system design. Trans. on Computer Systems 2(4), 277–288 (1984)
25. Ho, C.: Reducing costs of Byzantine fault tolerant distributed applications. PhD thesis, Cornell University (May 2011)
26. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory IT-22, 644–654 (1976)
27. Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., Wylie, J.: Fault-scalable Byzantine fault-tolerant services. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005, Brighton, UK (October 2005)
28. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In: Proceedings of the Symposium on Operating System Design and Implementation, OSDI 2006. USENIX (2006)
29. Song, Y.J., van Renesse, R.: Bosco: One-Step Byzantine Asynchronous Consensus. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 438–450. Springer, Heidelberg (2008)
30. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: Proceedings of the USENIX Symposium on Network Design and Implementation, NSDI 2009 (2009)
31. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: UpRight cluster services. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009 (October 2009)
32. Li, J., Mazieres, D.: Beyond one-third faulty replicas in Byzantine fault tolerant systems. In: USENIX Symposium on Networked Systems Design and Implementation, NSDI 2007 (2007)
33. Wood, T., Singh, R., Venkataramani, A., Shenoy, P., Cecchet, E.: ZZ and the Art of Practical BFT. In: Proceedings of EuroSys 2011, Salzburg, Austria (2011)