Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems

Jinyuan Li[*] and David Mazières
VMware Inc. and Stanford University

Abstract

Byzantine fault tolerant systems behave correctly when no more than f out of $3f + 1$ replicas fail. When there are more than f failures, traditional BFT protocols make no guarantees whatsoever. Malicious replicas can make clients accept arbitrary results, and the system behavior is totally unspecified. However, there is a large spectrum between complete correctness and arbitrary failure that traditional BFT systems ignore. This paper argues that we can and should bound the system behavior beyond f failures.

We present BFT2F, an extension to the well-known Castro-Liskov PBFT algorithm [6], to explore the design space beyond f failures. Specifically, BFT2F has the same liveness and consistency guarantees as PBFT when no more than f replicas fail; with more than f but no more than $2f$ failures, BFT2F prohibits malicious servers from making up operations that clients have never issued and restricts malicious servers to only certain kinds of consistency violations. Evaluations of a prototype implementation show that the additional guarantees of BFT2F come at the cost of only a slight performance degradation compared to PBFT.

1 Introduction

Applications with high security needs can reduce the danger of attacks through Byzantine-fault-tolerant (BFT) replication. A service replicated over several BFT servers can survive even when some fraction of the replicas fail (become malicious). Thus, failures whose likelihood is not correlated across machines have a much smaller probability of affecting overall application security. In particular, if replicas are separately administered, BFT replication protects against malicious or incompetent machine operators. To a limited extent, BFT replication also protects against software errors in components with different implementations on different machines.

BFT protocols generally aspire to two properties [6]: *linearizability* [9] and *liveness*. Linearizability, also called *ideal consistency* or *safety*, is the property that the service appears to all clients to execute an identical sequence of requests, and that this sequence preserves the temporal order of non-concurrent operations. Liveness means that the system makes progress in executing clients' requests, at least under some weak assumptions about eventual message delivery.

Unfortunately, all BFT protocols make a strong assumption that some predetermined fraction of server replicas are honest. In particular, the highest fraction of failures that an asynchronous BFT system can survive without jeopardizing linearizability or liveness is f out of $3f + 1$ replicas. The reason is that asynchronous communication makes it impossible to differentiate slow replicas from failed ones. To progress safely with f unresponsive replicas, a majority of the remaining $2f + 1$ responsive ones must be honest.

The security of today's best-known BFT algorithms fails completely given even just $f + 1$ compromised replicas. For example, an attacker who compromises two out of four servers can return arbitrary results to any request by any client, including inventing past operations that were never requested by any user, rolling back history to undo operations that were already revealed to clients, or producing illegal results that could not have arisen from any valid sequence of operations. However, depending on the situation, it may be harder for an attacker to compromise two thirds of replicas than one third. A system that guarantees some residual security properties when a third or more of the replicas fail could significantly improve security in such settings.

Fortunately, linearizability and total failure are not the only options. The goal of this work is to limit the damage when more than f out of $3f + 1$ servers in a replicated state system fail. Specifically, we explore a new, weaker consistency model, fork* consistency, a derivative of fork consistency introduced earlier [16]. With fork* consistency, it is possible to bound a system's behavior when between $f+1$ and $2f$ replicas have failed. When $2f+1$ or more replicas fail, it is unfortunately not possible to make any guarantees without simultaneously

[*]Research done while the author was visiting Stanford University.

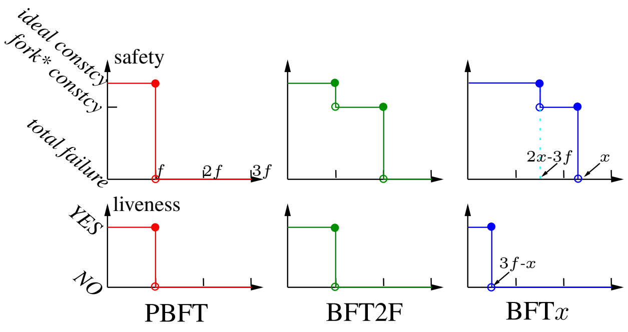


Figure 1: Comparison of the safety and liveness guarantees of PBFT, BFT2F, and BFTx. As we can see, BFT2F provides extra safety guarantees without compromising liveness, which is strictly better than PBFT.

sacrificing liveness for cases where f or fewer replicas fail.

We propose BFT2F, a protocol that provides exactly the same linearizability guarantee as PBFT [6] when up to f replicas have failed. When more than f but no more than $2f$ replicas have failed, two outcomes are possible. (1) The system may cease to make progress—In other words, BFT2F does not guarantee liveness when more than f replicas are compromised. Fortunately, for most applications, denial of service is less harmful than arbitrary behavior. (2) The system may continue to operate and offer fork* consistency, which again for many applications is preferable to arbitrary behavior.

Fork* consistency is perhaps best motivated by example. Consider a campus on which people swipe ID cards to open locked doors. Suppose the server that processes access requests is replicated on four machines. With PBFT, an attacker who compromises two replicas can open any door on campus—even doors he has never had access to—without leaving an audit trail. With fork* consistency, the attacker might be able to delay revocation requests and gain access to rooms he should no longer have access to. He may also complicate recovery by causing his accesses to be logged on only one correct replica instead of all correct replicas. However, he will not gain access to new rooms or be able to unlock doors without leaving some evidence behind.

In practice, surviving up to $2f$ malicious replicas enables replicated state systems for the first time to handle the case where a *majority* of the replicas may fail. Meanwhile, BFT2F does not compromise any guarantees that existing BFT systems provide. One might wonder if there exist protocols that guarantee fork* consistency with even larger numbers of failures, e.g., x failures for $2f < x \leq 3f$. The answer is yes. In fact, BFT2F can be easily parameterized to support that, which we term BFTx in Figure 1. However, achieving fork* consistency with more than $2f$ failures comes at the cost of sacrificing liveness when there are fewer than f failures. Administrators who care more about security than availability can tune the protocol to guarantee fork* consistency anywhere up to $3f$ failures. In fact, through a degenerate case, a system configured to guarantee fork* consistency with as few as one correct replica actually provides linearizability, at the cost of sacrificing liveness in the face of even one failure.

The paper is organized as follows. We start by specifying the details of the system model and our assumptions in Section 2. Section 3 gives a formal specification of fork consistency and proves that no protocols can guarantee fork consistency using only one round of communication when more than f servers fail. In Section 4, we relax the model of fork consistency further to arrive at fork* consistency, which is achievable using a one round protocol and still provides useful guarantees. We present BFT2F, an extension to the classical PBFT protocol to achieve fork* consistency when more than f servers are faulty in Section 5. We evaluate a prototype implementation of BFT2F in Section 6. Lastly, we discuss related work in Section 7 and conclude in Section 8.

2 Models and Assumptions

We use the same replicated state machine model as presented in past work [6, 24]. The network is unreliable and may replicate, discard, and reorder messages with the one exception that it cannot indefinitely prevent two honest parties from communicating with each other.

The system consists of a number of clients and $3f+1$ replicated servers, where f is a predetermined parameter. Clients make requests for operations to servers. An operation is *executed* when servers accept it and apply it to their state. An operation is *completed* when its originating client accepts the reply. We use lower-case letters that appear early in the alphabet such as a, b, c to denote clients, and letters that appear late in the alphabet such as p, q, u, w to denote servers. We also use the term replicas and servers interchangeably. A node is either a client or server. Each node has a public/private key pair. Each server knows the public keys of all nodes and each client knows the public keys of all servers.

We assume a Byzantine failure model. A faulty node may either halt or deviate from correct behavior arbitrarily, even colluding with other faulty nodes. However, a faulty node cannot forge digital signatures of correct nodes or find collisions of a cryptographic hash function.

3 Fork consistency

Figure 2 shows the server-side interface between a state-machine replication system (such as PBFT) and the replicated service (e.g., an NFS file server). The service starts in a fully-specified initial state and provides a single function, EXECUTE, that takes a request operation, *op*, as an argument, then deterministically updates its state and returns a result, *res*. In ordinary operation, clients submit requests through a client-side replication library. The server-side code communicates amongst replicas so as to invoke EXECUTE with an identical sequence of requests on every replica, thereby producing identical results to send back to the client. The client side library returns the results from the matching replies to the client.

Of course, a compromised server need not behave as described above. It can fail to respond to messages. Worse yet, it may return arbitrary results to clients; it may subvert network routing to intercept and tamper with packets; it may even collude with other compromised servers and send carefully-crafted messages that subvert honest replicas and ultimately convince the client library to return incorrect or inconsistent results. We need techniques that allow honest clients and replicas to minimize the damage that may be caused by malicious replicas.

A significant part of the challenge is ensuring that malicious replicas do not corrupt the internal state of honest replicas. Because an honest replica's internal state is entirely determined by the sequence of all operations it has ever executed, we introduce a term for this history: We call the sequence $L^{all} = \langle op^1, op^2, ..., op^n \rangle$ of all operations executed by a replica the *result list* of its latest operation op^n. This result list entirely determines both the result returned by the EXECUTE function for op^n and the state S^{n+1} of the replica after executing op^n. Whenever possible, we would like to ensure that honest replicas all have the same result lists and that these lists contain only requests legitimately issued by users.

To defend against illegitimate requests, each operation may be signed by the key of its originating user. Without knowing a user's private key, a bad replica cannot forge requests. By verifying the signatures on requests, honest replicas can limit themselves to calling EXECUTE only on operations legitimately requested by users. Should malicious replicas learn a compromised user's private key, they may be able to convince honest replicas to execute arbitrary operations with the privileges of the compromised user; effectively any well-formed operation by a compromised user is legitimate.

In an ideally consistent (well-behaved) system, every honest replica will have the same result list $L^{all} = \langle op^1, op^2, ..., op^n \rangle$ for operation op^n. Moreover, this list will preserve the temporal order of non-overlapping operations. Malicious replicas, of course, may behave as though op^n's result is *not* the outcome of executing L^{all}, but rather of some different result list L^{bad}. In fact, malicious server behavior may be incompatible with *any* possible sequence of legitimate requests; we therefore allow L^{bad} to contain illegal operations (such as "cause the next operation to return this particular result").

```
// At server side:
$(S^{n+1}, res) \leftarrow$ EXECUTE$(S^n, op)$;
```

Figure 2: Pseudocode for replicated state machines.

When a system undergoes total failure (as in the case of PBFT with more than f faulty replicas), an operation's result list even at an honest replica might not contain all previously issued requests; malicious servers may have succeeded in concealing the existence of legitimately issued operations. Worse yet, it may be that different replica's result lists contain the same operations in different orders, and that this convinces the client libraries on different machines to return inconsistent results. To mitigate these problems, we previously defined a consistency model, fork consistency [16], that can be realized in the presence of entirely malicious servers. An important property of fork consistency is that it helps provide server accountability by allowing clients to detect any past consistency violations on the part of bad servers.

In defining fork consistency, we make a distinction between operations issued by *correct* clients, which obey the protocol, and *malicious* or compromised clients that don't. We use the term result lists only for replies to operations issued by correct clients running our library, as it makes little sense to talk about the results seen by clients that do not implement the protocol or even necessarily use our software. However, we consider any result accepted by a correct client to have a result list. In the event that malicious servers completely break the system, they may induce clients to accept "impossible" results, in which case the corresponding result lists must have illegal operations.

Definition: A system is *fork consistent* if and only if it satisfies the following requirements:

- **Legitimate-request Property:** Every result accepted by a correct client has a result list L that contains only well-formed operations legitimately issued by clients.

- **Self-consistency Property:** Each honest client sees all past operations from itself: if the same client a

issues an operation op^i before op^j, then op^i also appears in op^j's result list.

This property ensures that a client has a consistent view with respect to its own operations. For example, in a file system, a client always reads the effect of its own writes.

- **No-join Property:** Every accepted result list that contains an operation *op* by an honest client is identical up to *op*.

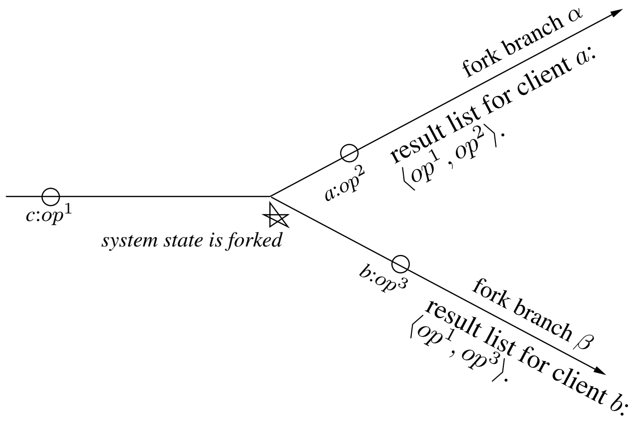
One corollary of fork consistency is that when two clients see each others' recent operations, they have also seen *all* of each other's past operations in the same order. This makes it possible to audit the system; if we can collect all clients' current knowledge and check that all can see each others' latest operations, we can be sure that the system has *never* in the past violated ideal consistency. Another corollary is that every result list preserves the temporal order of non-concurrent operations by correct clients, since an accepted result list cannot contain operations that have not yet been issued.

We say two result lists are *consistent* if one is an improper prefix of the other. We say two result lists are *forked* if they are not consistent. For instance, result list $\langle op^1, op^2 \rangle$ and result list $\langle op^1, op^3 \rangle$ are forked, while $\langle op^1, op^2 \rangle$ and $\langle op^1, op^2, op^3 \rangle$ are not. If clients accept forked result lists, the system has failed to deliver ideal consistency.

We say servers are in different *fork sets* if they reflect forked result lists. A fork set consists of a set of servers who return the same set of consistent result lists to all clients, though in an asynchronous system, some results may not reach the client before an operation completes. Intuitively, a correct server cannot be in more than one fork set. A malicious server, however, can simultaneously be in multiple fork sets—pretending to be in different system states when talking to different nodes. An attacker who controls the network and enough replicas may cause the remaining honest replicas to enter different forks sets from one another, at which point with any practical protocol clients can no longer be assured of ideal consistency.[1]

Figure 3 shows an example where fork consistency differs from ideal consistency. Initially, all clients have the same result list $\langle op^1 \rangle$. Client a issues a request op^2, and gets back the result list $\langle op^1, op^2 \rangle$ from fork set

[1]Technically, a contrived protocol could permit honest replicas to enter forked states that clients could always detect. However, such a protocol would be needlessly vulnerable; since honest replicas know the system's complete state, they can and should detect any attacks clients can, rather than purposefully entering divergent states that would be more difficult to recover from.



fork branch α
result list for client a: $\langle op^1, op^2 \rangle$.
$a{:}op^2$
$c{:}op^1$
system state is forked
$b{:}op^3$
fork branch β
result list for client b: $\langle op^1, op^3 \rangle$.

Figure 3: An example of fork consistency. Since the server deceives client b about a's operation op^2, client a's result list $\langle op^1, op^2 \rangle$, and b's result list $\langle op^1, op^3 \rangle$ are only fork consistent, not ideally consistent. (Strictly speaking, client a still has ideal consistency at this moment, should op^2 be ordered before op^3. However, client a will miss ideal consistency hereafter.)

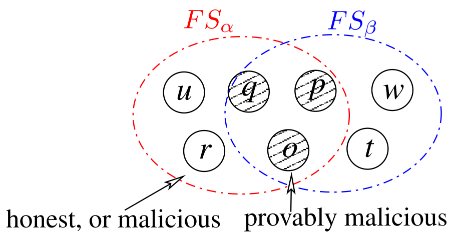


Figure 4: In a replicated state system, the intersection of two fork sets can only consist of *provably malicious* servers, which are denoted by shaded circles. The partition that excludes the intersection part might have honest servers and malicious ones that have not yet misbehaved.

FS_α; after that, client b's operation op^3 returns with the result list $\langle op^1, op^3 \rangle$ from fork set FS_β. Here the server deceives b about a's completed operation op^2. Therefore, client a's result list and client b's are forked. At this moment, one can verify that the system has delivered fork consistency, but failed to provide ideal consistency. It is worth pointing out that if a protocol returned the entire result list with each reply to an operation, a client could compare result lists to detect the attack whenever it received replies from two different fork sets. The BFT2F protocol presented in Section 5 uses hashes to achieve a similar guarantee without the efficiency or privacy problems of shipping around result lists.

Figure 4 shows a potential configuration that could produce the two fork sets in Figure 3. $FS_\alpha \bigcap FS_\beta$ must have *only* faulty servers that have already acted maliciously (e.g., p, q, o). $FS_\alpha - (FS_\alpha \bigcap FS_\beta)$ and $FS_\beta - (FS_\alpha \bigcap FS_\beta)$ can include either honest servers (e.g., u, w, t), or malicious servers who have not misbehaved so far (e.g., r could be such a server), and so who are externally indistinguishable from honest servers, though they might later deviate from the protocol.

3.1 Fork consistency examples

For some applications, fork consistency may be nearly as good as ideal consistency. In the example of a card-swipe access control service, if two out four replicas are compromised and an attacker forks the system state, it is likely only a matter of hours or days before people notice something is wrong—for instance that new users' cards do not work on all doors. At that point, the administrators must repair the compromised replicas. In addition, they may need to merge access logs and replay all revocations executed in either fork set by the two honest replicas. Even without server compromises, adding and removing door access is often not instantaneous anyway, both because of bureaucracy and because people do not always report lost or stolen ID cards immediately.

On the other hand, in other settings fork consistency may lead to quantifiable damage compared to ideal consistency. Consider a bank that uses four machines to replicate the service that maintains account balances. An attacker who compromises two of the machines and forks the state can effectively duplicate his account balance. If the attacker has \$1,000 in his account, he can go to one branch and withdraw \$1,000, then go to a branch in a different fork set and withdraw \$1,000 again.

Finally, of course, there are settings in which fork consistency on its own is nearly useless. For example, a rail signaling service split into two fork sets, each of which monitors only half the trains and controls half the signals, could easily cause a catastrophic collision. One way to mitigate such problems is to leverage fork consistency to get bounded inconsistency through heartbeats. A trusted heartbeat client could increment a counter every few seconds; trains that do not detect the counter being incremented could sound an alarm. The attacker would then need to compromise either another f replicas or the heartbeat box to suppress the alarms.

3.2 Impossibility of one-round fork consistency

Unfortunately, for a system to guarantee fork consistency when the fraction of honest replicas is too small for linearizability, clients must send at least two messages for every operation they perform. The most serious implication of this is that the system can no longer survive client failures—if a client fails between the first and second messages of an operation, the system may no longer be able to make progress. Another disadvantage is that slow clients can severely limit system throughput by leaving servers idle between the two messages. Finally, there are various implementation reasons why a one-round protocol is often simpler and more efficient, particularly if there is high latency between the client and servers.

Theorem: In an asynchronous system that provides liveness when up to f out of $3f + 1$ replicas fail, guaranteeing fork consistency in the face of more than f replica failures requires more than one round of client-server communication on each operation.

Proof Sketch: For simplicity, we consider the case of four replicas with $f = 1$, though the same argument applies to any number of replicas. In a single-round protocol, a client sends a single message that eventually convinces honest replicas to alter their state by executing an operation. Consider the case when two clients, a and b, issue two requests, op_a and op_b, concurrently. Neither client could have known about the other's request when issuing its operation. Thus, either op_a or op_b is capable of being executed before the other.

If, for instance, the network delays op_a, op_b could execute before op_a arrives and vice versa. Moreover, because of liveness, request op_a must be capable of executing if both client b and replica w are unreachable, as long as the remaining three replicas respond to the protocol. Figure 5 illustrates this case, where the result list $\langle op_b, op_a \rangle$ was eventually returned to client a.

The same reasoning also applies to client b, who might get result list $\langle op_a, op_b \rangle$ when replica u is unreachable. These two out-of-order result lists reflect the (malicious) servers' ability to reorder concurrent requests at will, if a single message from a client is allowed to change the system state *atomically*. This clearly violates the no-join property of fork consistency. ∎

3.3 A two-round protocol

The problem with a one-round protocol is that at the time a client signs a request for operation op, there may be previous operations by other clients it does not know about. Therefore, there is nothing the client can include in its signed request message to prevent the two honest replicas (u and w) from believing op should execute in two different contexts.

In a two-round protocol, the client's first message

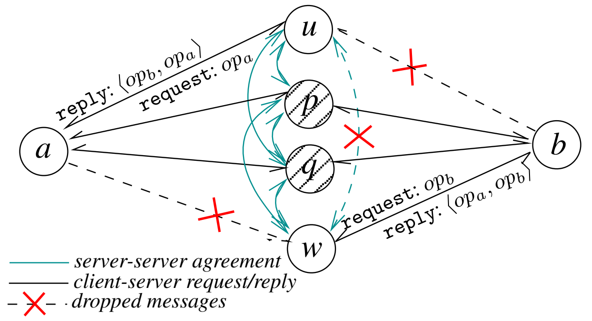

u
p
q
w
a
b
reply: $\langle op_b, op_a \rangle$
request: op_a
request: op_b
reply: $\langle op_a, op_b \rangle$
server-server agreement
client-server request/reply
dropped messages

Figure 5: Two malicious servers (p and q) wear different hats when talking to distinct honest servers (u or w) in the server-server agreement phase, during which all servers communicate with each other to achieve consensus on incoming requests. Meanwhile, they delay the communication between u and w. In this way, p and q, with u, return result list $\langle op_b, op_a \rangle$ to client a; p and q, with w, return $\langle op_a, op_b \rangle$ to client b.

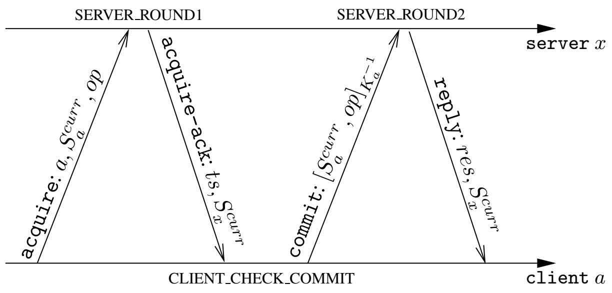

SERVER_ROUND1
SERVER_ROUND2
server x
acquire: a, S_a^{curr}, op
acquire-ack: ts, S_x^{curr}
commit: $[S_a^{curr}, op]_{K_a^{-1}}$
reply: res, S_x^{curr}
CLIENT_CHECK_COMMIT
client a

Figure 6: A two-round protocol.

can request knowledge about any previous operations, while the second message specifies both the operation and its execution order. Figure 6 shows such a protocol. A client sends an `acquire` request to acquire the system's latest state in the first round; a server replies with its current state. Disregarding efficiency, a straw-man protocol might represent this state as the previous operation's result list (i.e., the impractically long list of every operation ever executed). In the second round, the client commits its operation *op* by signing a message that ties *op* to its execution order. Again, in a straw-man, inefficient protocol, the client could append its new operation to the list of all previous operations and sign the whole list. Servers then sanity check the commit message, execute *op*, and send back the reply.

Note that servers are not allowed to abort an operation after replying to an `acquire` message—the `acquire-ack` is a commitment to execute the operation in a particular order given a signed `commit`. Otherwise, if servers could optionally abort operations, malicious replicas p and q could convince u and a that op_a aborted while convincing w to execute it, violating fork consistency. But of course a server cannot execute an operation until it sees the signed `commit` message, which is why clients can affect both liveness and throughput.

These problems can be overcome in application-specific ways. When it is known that an operation's result cannot depend on a previous operation's execution, it is safe to execute the later operation before the previous one commits. SUNDR performs this optimization at the granularity of files and directory entries, which mostly overcomes the performance bottleneck but can still lead to unavailable files after a client failure. Because SUNDR does whole-file overwrites, one can recover from mid-update client crashes by issuing a new write that either deletes or supersedes the contents of whatever file is generating I/O timeout errors.

HQ replication [8] similarly allows concurrent access to different objects. The technique could be used to improve performance of a two-round protocol. However, object semantics are general enough that there would be no way to recover from a client failure and still provide fork consistency with $f+1$ malicious replicas.

Figure 7 shows a two-round protocol in pseudocode. S_n^{curr} represents node n's latest knowledge of the system state. We say $S_i^{curr} \prec S_j^{curr}$ (meaning S_i^{curr} *happens before* S_j^{curr}) if one of node j's past system states is identical to S_i^{curr}—in other words, node j's current knowledge is consistent with and strictly fresher than i's. In our straw-man protocol, if S_n^{curr} is a list of all operations ever executed, then $S_i^{curr} \prec S_j^{curr}$ means S_i^{curr} is a prefix or S_j^{curr}.

4 Fork* consistency

Because of the performance and liveness problems of two-round protocols, this section defines a weaker consistency model, fork* consistency, that can be achieved in a single-round protocol. Intuitively, rather than have separate `acquire` and `commit` messages, our plan will be to have each request specify the precise order in which the same client's *previous* request was supposed to have executed. Thus, it may be possible for an honest replica to execute an operation out of order, but at least any future request from the same client will make the attack evident.

Turning back to the example in Figures 3 and 4, with a one-round protocol, after the system state has been forked, client c's operation op^4 might be applied to servers in both fork sets if c has no clue that the system state has been forked when it issues op^4. Then the honest servers in both FS_α and FS_β will update their system state, S_α or S_β, respectively, with op^4, violating the no-join property of fork consistency. However, client

```
//executed by server $x$ upon receiving acquire.
procedure SERVER_ROUND1($a, S_a^{curr}, op$)
 if $next = null$ and $S_a^{curr} \prec S_x^{curr}$
     Extract timestamp $ts$ from $op$;
     Agree with other servers that current state is $S_x^{curr}$
       and next operation to execute is $op$ from client $a$;
     $next \leftarrow [a, op]$;
     send [acquire-ack, $ts, S_x^{curr}$] to client $a$;


//executed by client $a$ to collect acquire-acks
//and generate commit.
procedure CLIENT_CHECK_COMMIT($op$)
 Decide on the system's current state $S^{curr}$ based on
   $2f + 1$ matching acquire-acks from servers;
 $S_a^{curr} \leftarrow S^{curr}$;
 send $[\texttt{commit}, S_a^{curr}, op]_{K_a^{-1}}$ to all servers;


//executed by server $x$ upon receiving commit.
procedure SERVER_ROUND2($a, S_a^{curr}, op$)
 if $next = [a, op]$ and $S_a^{curr} = S_x^{curr}$
     $(S_x^{curr}, res) \leftarrow$ EXECUTE($S_x^{curr}, op$);
     send [reply, $S_x^{curr}, res$] to client $a$;
     $next \leftarrow null$;
```

Figure 7: Pseudocode for a two-round protocol.

c is only going to accept the reply from one of the fork sets, e.g., FS_α, and adopt its new state, S_α, as its freshest knowledge of the system state. If the protocol can prohibit any correct node that has seen state in one fork set, namely FS_α, from issuing any operations that execute in another (e.g., FS_β), then all future operations from client c can only be applied to servers in fork set FS_α (or subsequent forks of FS_α).

Fork* consistency therefore relaxes the no-join property in fork consistency to a join-at-most-once property: two forked result lists can be joined by at most one operation from the *same* correct client.

- **Join-at-most-once Property:** If two result lists accepted by correct clients contain operations op' and op from the same correct client, then both result lists will have the operations in the same order. If op' precedes op, then both result lists are identical up to op'.

The join-at-most-once property is still useful for system auditing. We can periodically collect all clients' current knowledge to check that all have seen each others' latest operations. Suppose each client issues an operation for this purpose at times t^1, t^2 ... t^n, where t^n is the time of the latest check. The join-at-most-once property guarantees that if the checks succeed, then the system has *never* violated ideal consistency up to time t^{n-1}.

In the example above, for client a, the new result list is $\langle op^1, op^2, op^4\rangle$, and for b, it is $\langle op^1, op^3, op^4\rangle$. Yet, the system still delivers fork* consistency at this moment, just not fork consistency or ideal consistency. However, fork* consistency forces malicious servers to choose between showing c's future operations to a or b, but not both.

4.1 Fork* consistency examples

In the card-swipe example, fork* consistency has effectively the same consequences as fork consistency. It delays revocation and complicates recovery. The only difference is that after the attack one operation from each client may appear in both fork sets, so, for example, one user may get access to a new door in both fork sets, but no one else will.

In the case of the bank, fork* consistency increases the potential loss. For example, the attacker can go to one branch, withdraw \$1,000, then go to the other branch and deposit the \$1,000. The attacker can ensure this deposit shows up in both fork sets—allowing him to withdraw an additional \$1,000 from the first branch, as well as the \$2,000 he can get from the branch at which he made the deposit.

In the case of a heart-beat server used to sound an alarm, fork* consistency doubles the amount of time required to detect a problem, since one increment of the counter may show up in both fork sets.

5 BFT2F ALGORITHM

In this section, we present BFT2F, an extension of the original PBFT protocol [6] that guarantees fork* consistency when more than f but not more than $2f$ out of $3f + 1$ servers fail in an asynchronous system.

Like PBFT, BFT2F has replicas proceed through a succession of numbered configurations called *views*, and uses a three-phase commit protocol [22] to execute operations within views. The three phases are *pre-prepare*, *prepare*, and *commit*. In view number $view$, replica number $view \bmod 3f + 1$ is designated the *primary* and has responsibility for assigning a new sequence number to each client operation.

5.1 BFT2F Variables

Below, we describe major new variables and message fields introduced in BFT2F. We use superscripts to denote sequence number, e.g., msg^n refers to the message with sequence number n. We use subscripts to distinguish variables kept at different nodes.

Hash Chain Digest (HCD): A HCD encodes all the operations a replica has committed and the commit order. A replica updates its current HCD to be $HCD^n = D(D(msg^n) \circ HCD^{n-1})$ upon committing msg^n, where D is a cryptographic hash function and $\circ$ is a concatenation function. Replicas and clients use HCDs to verify if they have the same knowledge of the current system state.

Hash Chain Digest History: To check if a replica's current knowledge of the system state is strictly fresher than another replica's and not forked, each replica keeps a history of past HCDs. We denote replica p's history entry for msg^n as $T_p[n]$.

Version Vector: Every node i represents its knowledge of the system state in a version vector V_i. The version vector has $3f+1$ entries, one for each replica. Each entry has the form $\langle r, view, n, HCD^n \rangle_{K_r{}^{-1}}$, where r is the replica number, $view$ is the view number, n is the highest sequence number that node i knows replica r has committed, and HCD^n is r's HCD after n operations. The entry is signed by r's private key $K_r{}^{-1}$. We denote replica r's entry in V_i by $V_i[r]$, and use C structure notation to denote fields—i.e., $V_i[r].view$, $V_i[r].n$, and $V_i[r].HCD$.

We define several relations and operations on these data structures:

- Let V be a version vector. We define a *cur* function to represent the current state of the system in V as follows: If at least $2f+1$ entries in V have the same n and HCD values, $\mathrm{cur}(V)$ is one of these entries (e.g., the one with the lowest replica number). Otherwise, $\mathrm{cur}(V) = \mathbf{none}$.

- Let h and g be two version vector entries signed by the same replica. We say h *dominates* g iff $h.view \geq g.view$ and either the two entries have identical n and HCD fields or $h.n > g.n$.

- We say h *dominates* g *with respect to* hash chain digest history T iff h dominates g and the two entries' HCD fields appear at the appropriate places in T—i.e., $h.HCD = T[h.n]$ and $g.HCD = T[g.n]$. In other words, this means that g appears in the history T that leads up to h.

Whenever a client a sees a version vector entry h signed by replica p, it always updates its own version vector V_a by setting $V_a[p] \leftarrow h$ if h dominates the old value of $V_a[p]$. A server r similarly updates $V_r[p] \leftarrow h$ if h is recent (e.g., not older than the beginning of T_r) and dominates the old $V_r[p]$ with respect to T_r.

5.2 BFT2F Node Behavior

In describing BFT2F, we borrow heavily from PBFT [6]. However, we point out two major differences between BFT2F and PBFT. First, unlike in PBFT, BFT2F replicas do not allow out of order commits. This requirement does not pose much overhead as replicas must execute client operations in increasing sequence numbers anyway. Second, BFT2F requires clients to wait for at least $2f+1$ matching replies before considering an operation completed, as opposed to the $f+1$ matching replies required in PBFT.

5.2.1 Client Request Behavior

A client a multicasts a request for an operation $\langle \texttt{request}, op, ts, a, \mathrm{cur}(V_a) \rangle_{K_a^{-1}}$ to all the replicas, where ts is a monotonically increasing timestamp, $\mathrm{cur}(V_a)$ is a's last known state of the system, and the message is authenticated by a's signature key, $K_a{}^{-1}$.

We note that while PBFT uses more efficient MAC vectors to authenticate requests from clients, BFT2F requires public-key signatures. The reason is that faulty clients may interfere with the replicas' operation by issuing requests in which some MAC vector entries are valid and some are invalid. Such a partially correct MAC vector causes some replicas to accept the request and some to reject it. PBFT can recover from such a situation if $f+1$ replicas attest to the validity of their MAC vector entries. However, in BFT2F, we want to avoid executing illegitimate operations even with $> f$ faulty replicas, which means it is still not safe to execute an operation under such circumstances.

5.2.2 Server Behavior

Every server keeps a replay cache containing the last reply it has sent to each client. Upon receiving a request $\langle \texttt{request}, op, ts, a, \mathrm{cur}(V_a) \rangle_{K_a^{-1}}$ from client a, a server first checks the signature, then checks that the request is not a replay. Clients execute one operation at a time, so if ts is older than the last operation, the server ignores the request. If ts matches the last operation, the server just re-sends its last reply. Otherwise, the server checks that $\mathrm{cur}(V_a).HCD$ matches the HCD in the last reply the server sent to the client. If it does not, the client may be in a different fork set, or may be malicious and colluding with a malicious server. Either way, the server ignores this request. Once a message has been validated and checked against the replay cache, processing continues differently on the primary and other servers.

The primary replica, p, assigns the request a sequence number n and multicasts a `pre-prepare` message $\langle \texttt{pre-prepare}, p, view, n, D(msg^n) \rangle_{\sigma_p}$ to all other replicas. Here σ_p is either a MAC vector or a signature with p's private key, $K_p{}^{-1}$.

Upon receiving a pre-prepare message matching an accepted request, replica q first checks that it has not accepted the same sequence number n for a different message msg'^n in the same view $view$. It also ensures n is not too far out of sequence to prevent a malicious primary from exhausting the sequence number space. Replica q then multicasts a prepare message $\langle \texttt{prepare}, q, view, n, D(msg^n)\rangle_{\sigma_q}$ to all other replicas.

A replica u tries to collect $2f$ matching prepare messages (including one from itself) with the same sequence number n as that in the original pre-prepare message. When it succeeds, we say replica u has *prepared* the request message msg^n. Unlike PBFT, u does not commit out of order, i.e., it enters the *commit* phase only after having prepared the message *and* committed all requests with lower sequence numbers.

To start committing, replica u computes $HCD^n \leftarrow D(msg^n \circ HCD^{n-1})$ and sets $T_u[n] \leftarrow HCD^n$, updates $V_u[u]$ to $\langle u, view, n, HCD^n\rangle_{K_u{}^{-1}}$, and multicasts a commit message $\langle \texttt{commit}, \langle u, view, n, HCD^n\rangle_{K_u{}^{-1}}\rangle$ to all other replicas.

When replica w receives a commit message from u with a valid signature and HCD^n, it updates the entry for u in its current version vector, $V_w[u]$, to $\langle u, view, n, HCD^n\rangle_{K_u{}^{-1}}$. Replica w commits msg^n when it receives $2f+1$ matching commit messages (usually including its own) for the same sequence number n and the same HCD (HCD^n).

Replica w executes the operation after it has committed the corresponding request message msg^n. It sends a reply message to the client containing the result of the computation as well as its current version vector entry $V_w[w]$. Since w has collected $2f + 1$ matching commit messages, we know that these $2f + 1$ replicas are in the same fork set up to sequence number n.

5.2.3 Behavior of Client Receiving Replies

A reply from replica w has the format, $\langle \texttt{reply}, a, ts, res, \langle w, view, n, HCD^n\rangle_{K_w{}^{-1}}\rangle_{\sigma_w}$, where $view$ is the current view number, ts is the original request's timestamp, and res is the result of executing the requested operation. A client considers an operation completed after accepting at least $2f + 1$ matching replies each of which contains the same ts, res, n, and HCD^n. (Recall by comparison that PBFT only requires $f + 1$ matching replies.) This check ensures the client only accepts a system state agreed upon by at least $2f + 1$ replicas. Therefore, if no more than $2f$ replicas fail, the accepted system state reflects that of at least one correct replica. Client a also updates its $V_a[w]$ to $\langle w, view, n, HCD^n\rangle_{K_w{}^{-1}}$ for each w of the $2f + 1$ replies, ensuring that HCD^n becomes the new value of $\mathrm{cur}(V_a).HCD$.

To deal with unreliable communication, client a starts a timer after issuing a request and retransmits if it does not receive the required $2f + 1$ replies before the timer expires. Replicas discard any duplicate messages and can also fetch missing requests from each other in case the client crashes.

5.3 Garbage Collection

If a replica r has been partitioned from the network, it may have missed some number of successfully executed operations and need to learn them from other replicas. For small numbers of missed operations, the replica can just download the logged operations and commits and execute any operation that has $2f + 1$ commit messages with appropriate HCDs. However, if other replicas have truncated their logs, r may not be able to download all missing operations individually. It may instead need to do a bulk transfer of the entire state of the service from other replicas. The question then becomes how to authenticate the state in such a transfer.

In PBFT, r validates the state using stable checkpoints gathered every I operations (e.g., $I = 128$). A stable checkpoint is a collection of signed messages from $2f + 1$ replicas attesting that the service had hash $D(\mathrm{state}^n)$ at sequence n. r can then believe state^n. In BFT2F, the signed messages must additionally vouch that state^n is in the same fork set as r and allow r to bring its replay cache up to date. Our implementation accomplishes this by having each replica keep state back to n_{low}, the lowest version number in its version vector. This state may be required for application-specific recovery from fork attacks anyway. However, it requires unbounded storage while any replica is unavailable, which may be undesirable, so here we outline a way to achieve fork* consistency with bounded storage.

When replica u signs a checkpoint for sequence n, it includes its version vector V_u in the message. If no w has $V_u[w].n \leq n-2I$, then no honest replica will ever ask to be updated from a state $n-2I$ or older. If, however, there is a $V_u[w] \leq n-2I$, then, for each such w, u includes in the checkpoint one of its own old version vector entries h that dominates $V_u[w]$ with respect to T_u. Furthermore, it keeps enough commit messages around to roll w's state forward from $V_u[w].HCD$ to $h.HCD$, so that w can be assured it is in the same fork set as h. To ensure w does not progress beyond $h.n$, replicas execute no more than $2I$ operations beyond the last stable checkpoint that shows them up to date.

In detail, u's signed checkpoint has the form $\langle \texttt{checkpoint}, r, n, D(\text{state}^n), D(\text{rcache}^n), V_u, E\rangle_{K_u^{-1}}$. Here rcache^n is u's replay cache at sequence n (without the signatures or replica numbers, to make it identical at all unforked replicas). V is u's current version vector. E is a set of older values of $V_u[u]$ computed as follows. For each w in which $V_u[w].n \leq n - 2I$, E contains u's signed version vector entry from the next multiple of I after $V_u[w].n + I$. A stable checkpoint consists of `checkpoint` messages signed by $2f + 1$ different replicas with the same n, $D(\text{state}^n)$, $D(\text{rcache}^n)$, and E (modulo the replica IDs and signatures in E). Given a stable checkpoint, u can delete log state except for operations more recent than $n - 2I$ and the $2I$ operations leading up to each element of E.

When computing a stable checkpoint, it may be that replicas do not initially agree on E. However, each replica updates its version vector using any recent signed entry with a valid HCD in any other replica's `checkpoint` message and multicasts a new `checkpoint` message upon doing so. (To ensure the receiving replica can check the HCD, replicas ignore `commit` and `checkpoint` messages before the last stable checkpoint, so that once marked stale in a stable checkpoint, a replica can only change to being completely up-to-date.) Note that even after a stable checkpoint exists, a replica that was previously shown as out of date can cause a second stable checkpoint to be generated for the same sequence number by multicasting a `checkpoint` message with its up-to-date version vector.

5.4 Server View Change

In BFT2F, a replica r experiencing a timeout sends a `view-change` message $\langle \texttt{view-change}, \mathit{view} + 1, V_r[r], P\rangle_{K_r^{-1}}$ to all other replicas. Here $V_r[r]$ is the version vector entry for r's last committed operation, while P is a set of sets P_m for each prepared message m with sequence number higher than n. Each P_m contains the `pre-prepare` message for m and $2f$ corresponding matching `prepare` messages.

The primary p in the new view $\mathit{view} + 1$ checks all signatures (but not MACs) and validates the $V_r[r].HCD$ value in `view-change` messages it receives. If $V_p[p]$ dominates $V_r[r]$ with respect to T_p, then $V_r[r]$ is valid. Otherwise, if $V_r[r]$ dominates $V_p[p]$, then p requests from r an operation and $2f + 1$ matching `commits` with appropriate HCDs for every sequence number between $V_p[p].n$ and $V_r[r].n$. p then executes these operations, bringing T_p up to date so that $V_p[p]$ dominates $V_r[r]$ with respect to T_p. If p cannot download the missing operations from r, it does a bulk state transfer.

We say two `view-change` messages *conflict* if their P fields include different operations for the same sequence number. This may happen if honest replicas are forked because of more than f failures, or if the σ authenticators on `pre-prepare` and `prepare` messages have corrupted MAC vectors and malicious replicas claim to have prepared messages for which they didn't actually receive a `pre-prepare` and $2f + 1$ matching `prepares`. As long as there are $2f + 1$ honest replicas (without which we cannot guarantee liveness), we will be in the latter case and p will eventually receive $2f + 1$ valid and non-conflicting `view-change` messages (including one from itself), at which point it multicasts a `new-view` message $\langle \texttt{new-view}, \mathit{view} + 1, \mathcal{V}, O\rangle_{K_p^{-1}}$. Here $\mathcal{V}$ is the set of $2f + 1$ valid, non-conflicting `view-change` messages. O is a set of `pre-prepare` messages constructed as below:

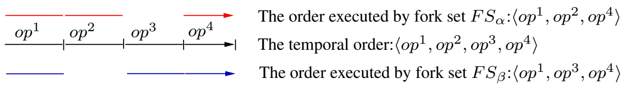
1. p determines *min-s* as the lowest sequence number of any version vector entry ($V_r[r]$) in an element of $\mathcal{V}$. p then determines *max-s* as the highest sequence number of any of the P_m sets in elements of $\mathcal{V}$.

2. For each sequence number n from *min-s* through *max-s*, p either (1) constructs a `pre-prepare` message in the new view, if a P_m in one of the view-change messages has a valid request for sequence number n, or (2) constructs a special *null* request $\langle\langle \texttt{pre-prepare}, p, \mathit{view} + 1, n, D(\mathit{null})\rangle_{K_p^{-1}}, \mathit{null}\rangle$ to fill in the sequence number gap.

A backup replica u in the new view validates the `new-view` message as follows. u checks the version vector in each element of $\mathcal{V}$ using the same checks p performed upon receiving the `view-change` messages. If u is too far out of date, it may need to do a state transfer. u also verifies that O is properly constructed by executing the same procedure as the primary. If u accepts the `new-view` message as valid, it sends a `prepare` message for each message in O and proceeds normally in the new view.

When there are no more than f faulty replicas, the above algorithm is essentially the same as PBFT with hash chain digests replacing the state hashes in PBFT's checkpoint messages. When more than f but no more than $2f$ replicas fail, there may be concurrent view changes in different fork sets. In the worst case when $2f$ replicas fail, up to $f + 1$ view changes may succeed concurrently, leading to $f + 1$ fork sets.

However, with no more than $2f$ failures, each fork set is guaranteed to contain at least one honest replica, r, which ensures fork* consistency. To see this, we con-



op^1
op^2
op^3
op^4
The order executed by fork set FS_α:$\langle op^1, op^2, op^4\rangle$
The temporal order:$\langle op^1, op^2, op^3, op^4\rangle$
The order executed by fork set FS_β:$\langle op^1, op^3, op^4\rangle$

Figure 8: Example of two forked result lists. The middle timeline shows the result list that should have been executed by a non-faulty system. The top timeline shows a forked result list that does not reflect op^3, while the bottom timeline shows another one missing operation op^2.

sider two cases. First, suppose r does not do a state transfer. Its hash chain digests are then simply the result of processing a sequence of operations in turn. Because r checks the $\mathrm{cur}(V_a)$ in a client's request against the last reply to that client, r will never accept a request from an honest client that has executed an operation in another fork set. At worst, r may execute the first request a client executes in a different fork set—hence achieving the join at most once property.

On the other hand, if r does a state transfer, this requires $2f + 1$ other replicas to vouch for r's new state. At least one of those $2f + 1$ replicas—call it u—must also be honest and in r's fork set. Since u is honest, it ensures that any operations r skips executing because of the state transfer do not violate fork* consistency.

5.5 An Example

We demonstrate the join-at-most-once property of BFT2F during normal case operation using a simple example. As in the example from Section 3, the system consists of four replicas u, p, q, w, with p being the primary in the current view and the two replicas p, q being malicious.

First we explain the intuition for why the join-at-most-once property can be achieved with a one-round protocol. Suppose the system is forked into two fork sets between two operations op^1 and op^4 issued successively by client c. Then c's second operation, op^4, might show up in two fork sets because the HCD c includes in the request for op^4 is taken from the reply to op^1, on which all replicas agreed. However, the replies c accepts for op^4 can only come from one or the other of the fork sets. Any subsequent operation c issues will only be valid in that particular fork set.

Now we consider this example in detail: client c issues the first (op^1) and fourth operation (op^4), and some other clients issue the second (op^2) and third (op^3) operations. The result list $\langle op^1, op^2, op^3, op^4\rangle$ would have reflected the order assigned in an otherwise non-faulty system, as shown in Figure 8. The malicious primary, p, assigns sequence number 1 to c's first operation op^1 and shows it to all other replicas. Subsequently, p only shows the second operation, op^2, to u and the third operation, op^3, to w, but it assigns both op^2 and op^3 the same sequence number 2. As a result, two fork sets FS_α and FS_β are formed, where FS_α contains u which has seen $\langle op^1, op^2\rangle$ and FS_β contains w which has seen $\langle op^1, op^3\rangle$.

Replica p then manages to join the two forked result lists for the first time with the operation op^4; the two result lists become $\langle op^1, op^2, op^4\rangle$ and $\langle op^1, op^3, op^4\rangle$, respectively. Suppose client c gets the required $2f+1 = 3$ replies for op^4 from fork set $FS_\alpha = \{u, p, q\}$. Then c's version vector will contain $HCD^3 = D(op^4 \circ D(op^2 \circ D(op^1)))$, while replica w in FS_β has a different version vector V_w containing $HCD'^3 = D(op^4 \circ D(op^3 \circ D(op^1)))$ (shown in Figure 9 Part (i)). Hereafter, if malicious servers p and q try to join the two forked result lists with a another operation by c, say op^5, the HCD^3 included in c's request would conflict with that in w's replay cache (Figure 9 Part (ii)).

5.6 Discussion

Ideal consistency requires *quorum intersection*; any two quorums should intersect in at least one non-faulty replica. Fork* consistency requires only *quorum inclusion*; any quorum should include at least one non-faulty replica. BFT2F uses quorums of size $2f+1$. However, as depicted in Figure 1, this can be generalized to a parameterized BFTx protocol that ensures quorum inclusion up to x failures with quorums of size $x+1$. With $3f+1$ replicas, any two quorums of size $x+1$ must overlap in at least $2(x+1) - (3f+1) = 2x - 3f + 1$ replicas, which guarantees quorum intersection up to $2x - 3f$ failures. Unfortunately, *quorum availability*, or liveness, requires that there actually be $x+1$ replicas willing to participate in the protocol, which is only guaranteed up to $3f + 1 - (x+1) = 3f - x$ failures.

BFT2F sets $x = 2f$. When up to f replicas fail, this provides quorum intersection. Because any two quorums share a non-faulty replica, the protocol can guarantee that any operation executed in one quorum is reflected in every subsequent quorum, just as in PBFT. When more than f but no more than $2f$ replicas fail, the quorums become fork sets. Quorum inclusion ensures at least one honest replica in each fork set. The honest replica in turn ensures that the quorum only executes legitimate operations and enforces join at most once, but it cannot guarantee ideal consistency or liveness.

BFT2F aims for full compatibility with PBFT and, subject to PBFT's liveness guarantees, an optimal guarantee beyond f replica failures. Another potentially useful construction is BFT3F, with $x = 3f$. BFT3F requires each quorum to be of size $3f+1$—i.e., to contain

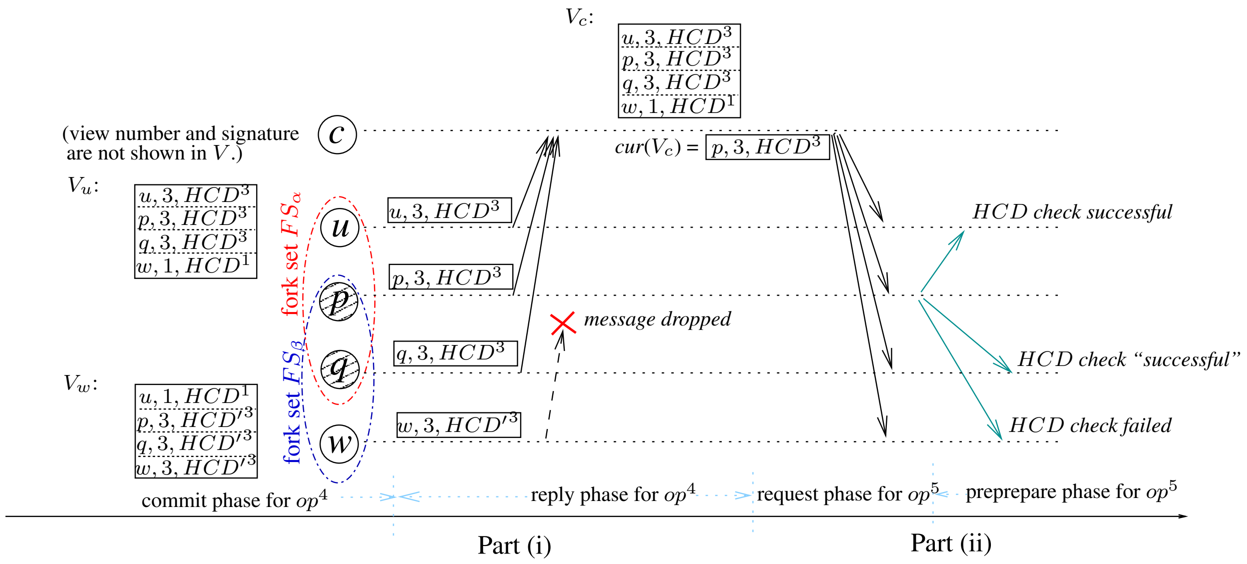


Figure 9: Example of the join-at-most-once property. op^4 is used to join two forked result lists as in Figure 8. Part (i) shows op^4's commit and reply phases. Since the result lists have already been forked, honest replicas in the two fork sets have different HCD histories by sequence number 3—$T_u[3] = HCD^3$ for u and $T_w[3] = HCD'^3$ for w—and thus include different HCDs in their replies. Client c accepts the reply from fork set FS_α and updates V_c accordingly. (c might detect a Byzantine failure upon seeing w's divergent reply in FS_β, but here the network drops the message.) Part (ii) shows that no future operation from c can execute in FS_β, since HCD^3 won't match w's last reply to c.

every replica in the system. This has two implications. First, since only *one* fork set can be formed, ideal consistency will be achieved with up to $3f$ failures. Second, the system will lose liveness with even one unresponsive replica, because no quorum can be formed under such circumstances. A system may want to use BFT3F if safety is the highest priority.

Both SUNDR [11] and BFT2F use version vectors to represent a node's knowledge of the current system state, with one major difference. SUNDR's version vectors have one entry per client, while BFT2F's use one per server. This difference brings BFT2F two advantages. First, BFT2F's version vectors are smaller, since there are typically fewer replicas than clients. Second, in both protocols, a node does not see any updates for version vector entries of nodes in a different fork set. In SUNDR, a stagnant entry could innocently signify that a client is simply offline or idle, both legitimate states. In contrast, BFT2F's replica servers should always remain online to process operations, so that a stagnant version vector entry is a good indication of failure.

6 Performance

We built a prototype implementation of the BFT2F algorithm on FreeBSD 4.11, based on our ported version of the BFT [6] library.

6.1 Implementation

BFT2F's additional guarantee over BFT [6] to allow detection of past consistency violations comes at the expense of much increased use of computationally expensive public key signatures instead of the symmetric session-key based Message Authentication Codes (MACs) used by PBFT. We use NTT's ESIGN with key length of 2048 bits in BFT2F. On a 3 GHz P4, it takes 150 μs to generate signatures, 100 μs to verify. For comparison, 1280-bit Rabin signatures take 3.1 ms to generate, 27 μs to verify.

All experiments run on four machines; three 3.0 GHz P4 machines and a 2.0 GHz Xeon machine. Clients run on a different set of 2.4 GHz Athlon machines. All machines are equipped with 1-3 GB memory and connected via a 100 Mbps Ethernet switch. Performance results are reported as the average of three runs. In all cases, standard deviation is less than 3 % of the average value.

6.2 Micro benchmark

Our micro benchmark is the built-in *simple* program in BFT, which sends a *null* operation to servers and waits for the reply. Appending version vector entries in `request` and `reply` messages has the most impact to the slowdown of BFT2F, compared to BFT.

req/rep(KB)	BFT	BFT(ro)	BFT2F	BFT2F(ro)
0/0	1.027	0.200	2.240	0.676
0/4	1.029	0.778	2.242	1.600
4/0	4.398	3.486	5.647	3.942

Table 1: Latency comparison of BFT and BFT2F (in milliseconds).

6.3 Application-level benchmark

We modify NFS to run over BFT2F, and compare it to the native BSD NFSv2, NFS-BFT running on 4 servers, and SUNDR (NVRAM mode) running on 1 server. The evaluation takes five phases: (1) copy a software distribution package `nano-1.2.5.tar.gz` into the file system, (2) uncompress it in place, (3) untar the package, (4) compile the package, (5) clean the build objects.

	NFSv2	NFS-BFT	NFS-BFT2F	SUNDR
P1	0.302	0.916	1.062	0.299
P2	1.161	3.546	4.131	0.520
P3	2.815	4.171	5.666	1.668
P4	3.937	4.296	4.922	3.875
P5	0.101	0.778	1.707	0.361
Total	8.316	13.707	17.488	6.723

Table 2: Performance comparison of different file system implementations (in seconds).

As Table 2 shows, the application-level performance slowdown in NFS-BFT2F relative to NFS-BFT is much less than that observed in our micro benchmark. This is because the high cost of public key operations and transferring version vector entries accounts for a smaller fraction of the cost to process requests. Both BFT2F and NFS-BFT achieve much lower performance than NFSv2 and SUNDR, reflecting the cost of replication.

7 Related work

Byzantine fault tolerant systems generally fall into two categories: replicated state machines [21] and Byzantine quorum systems [12, 13, 14, 25]. PBFT and BFT2F build on replicated state machines. By contrast, Quorums have simpler construction and are generally more scalable [1]. However, quorums usually provide only low-level semantics, such as read and write, which makes building arbitrary applications more challenging. Quorums also exhibit poor performance under contention. Replicated state machines generally deal with contention more efficiently, but scale poorly to larger numbers of replicas. Many other BFT systems [18, 10, 5] take this approach. Some wide area file systems [19, 2] run BFT on their core servers. HQ replication [8] unifies the quorum and state machine approaches by operating in a replicated state machine mode during high contention and in quorum mode during low contention.

Some work has been done to harden BFT systems against the probability of more than f simultaneous failures. Proactive recovery [7] weakens the assumption of no more than f faulty replicas during the lifetime of the service to no more than f failures during a window of vulnerability. It achieves this by periodically rebooting replicas to an uncompromised state. However, it's behavior is still completely undefined when more than f replicas fail in a given window. Furthermore, some problems such as software bugs persist across reboots. BASE [20] aims to reduce correlated failures. It abstracts well-specified state out of complex systems, and thus reduces the chances of correlated software bugs by allowing the use of different existing mature implementations of the same service.

By separating execution replicas from agreement replicas [24], one can tolerate more failures within execution replicas or reduce replication cost. BAR [3] takes advantage of the fact that selfish nodes do not fail in completely arbitrary ways. Dynamic Byzantine quorum systems [4] can adjust the number of replicas to tolerate varying f on the fly, based on the observation of system behavior.

Securing event history has been studied in the systems like timeline entanglement [15], which takes the hash chain approach as in BFT2F, and in [17, 23], which use version vectors to reason about partial ordering.

8 Conclusion

Traditional BFT algorithms exhibit completely arbitrary behavior when more than f out of $3f+1$ servers are compromised. A more graceful degradation could improve security in many settings. We propose a weak consistency model, fork* consistency, for BFT systems with larger numbers of failures. Fork* consistency prevents clients from seeing the effects of illegitimate operations and allows detection of past consistency violations. We present a new algorithm, BFT2F, based on PBFT, that provides the same guarantees as PBFT when no more than f replicas fail, but offers fork* consistency with up to $2f$ faulty replicas. While BFT2F does not guarantee liveness in the latter situation, denial of service is far preferable to arbitrary behavior for most applications. Evaluation of our prototype BFT2F implementation suggest its additional guarantees come with only a modest performance penalty.

ACKNOWLEDGMENTS

We thank Michael Freedman, Jinyang Li, Antonio Nicolosi, Junfeng Yang, Jinyuan's thesis committee, the anonymous reviewers, and our shepherd Barbara Liskov, for comments on the paper, as well as the National Science Foundation for support under grants CCR-0093361 and CNS-0430425.

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, Brighton, United Kingdom, October 2005. ACM.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.

[3] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 45–58, Brighton, United Kingdom, October 2005. ACM.

[4] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K. Reiter, and Rebecca N. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the the International Conference on Dependable Systems and Networks (FTCS 30 and DCCA 8)*, pages 283–292, June 2000.

[5] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 167–176, 2002.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[7] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 273–288, San Diego, CA, October 2000.

[8] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190, Seattle, WA, November 2006.

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

[10] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, 2001.

[11] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 121–136, December 2004.

[12] Dahlia Malkhi and Michael Reiter. Byzantine quorum system. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 569–578, El Paso, TX, May 1997.

[13] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[14] Dahlia Malkhi, Michael K. Reiter, Daniela Tulone, and Elisha Ziskind. Persistent objects in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2001.

[15] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[16] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002. The full version is available as NYU computer science department technical report TR2002-826, May 2002.

[17] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.

[18] Michael K. Reiter. The Rampart toolkit for building highintegrity services. *Lecture Notes in Computer Science 938*, pages 99–110, 1994.

[19] Sean Rhea, Patrick Eaton, and Dennis Geels. Pond: The OceanStore prototype. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[20] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

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

[22] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, April 1981.

[23] Sean W. Smith and J. D. Tygar. Security and privacy for partial order time. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, Las Vegas, NV, October 1994.

[24] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, NY, October 2003. ACM.

[25] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.