EFS: Efficient and Fault-Scalable Byzantine Fault Tolerant Systems Against Faulty Clients

Quanwei Cai[1,2,3], Jingqiang Lin[1,2](✉), Fengjun Li[4], Qiongxiao Wang[1,2], and Daren Zha[3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
{qwcai,linjq}@is.ac.cn, qxwang@is.ac.cn

[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing 100049, China

[3] University of Chinese Academy of Sciences, Beijing 100049, China
zhadaren@ucas.ac.cn

[4] The University of Kansas, Lawrence, KS 66045, USA
fli@ku.edu

Abstract. Byzantine fault tolerant (BFT) protocols enhance system safety and availability in asynchronous networks, despite the arbitrary faults at both servers and clients. A practical BFT system should be *efficient* in both contention-free and contending cases, and *fault scalable* (i.e., efficiently tolerating the increasing number of server faults). However, few existing BFT systems completely satisfy this robustness requirement of efficiency. In this paper, we propose EFS, the first BFT solution that provides good efficiency and fault-scalability, in various cases (i.e. faulty or not, contending or not). EFS is a hybrid BFT system consisting of an efficient and fault scalable quorum protocol for the contention-free case and a fast agreement protocol to resolve contention in a fault-scalable manner. More importantly, its server-directed mode switch does not rely on digital signature nor introduce any extra communication overhead. This lightweight switch counters the vulnerability in the existing hybrid BFT systems, where faulty clients can simply send contending requests to degrade the performance significantly. The experiment results on the EFS prototype demonstrate robust fault tolerance.

Keywords: Byzantine fault tolerance · Efficiency · Robustness · Fault-scalability

1 Introduction

Distributed services often encounter arbitrary failures (a.k.a. *Byzantine failures*) that are typically caused by software bugs, dynamic network delays, malicious actions of compromised nodes, etc. It is desirable that applications, especially the ones with high security requirements, are able to tolerate such Byzantine failures.

J. Tian et al. (Eds.): SecureComm 2014, Part I, LNICST 152, pp. 305–322, 2015.
DOI: 10.1007/978-3-319-23829-6_22

Many Byzantine fault tolerant (BFT) systems [1–6] have been proposed to provide reliable services using state machine replication (SMR) [7,8] – replicate the service on n servers starting from the same state and executing the same deterministic operations requested by clients. A certain number of servers (e.g., $n = 3f + 1$ in PBFT [1] or $5f + 1$ in Q/U [2]) cooperate to mask the negative impact of up to f Byzantine faulty servers. BFT systems shall guarantee *safety* and *liveness* in the presence of faulty servers and clients. For safety, non-faulty servers are expected to execute the requested operations in the same order. Liveness requires that a client can eventually receive a correct response through repeated requests with bounded message delays.

For practical BFT services, *efficiency* is another important consideration. BFT systems shall be efficient under the following typical scenarios [1,2,4,9].

- *Efficiency without fault or contention.* Although servers should agree on the execution order of concurrent operations in the presence of Byzantine failures, a BFT system usually runs efficiently for sequential requests without any fault.
- *Efficiency with faulty clients.* In BFT services for large-scale users, clients are more likely to be compromised than servers due to weaker protections, and the number of clients is much larger than that of servers.
- *Efficiency with the increasing number of server faults tolerated* (i.e., fault-scalability). For massive-scale services (e.g., Farsite [10] and OceanStore [11]) which are deployed in open environments, the number of faulty servers may increase dramatically due to network errors and component crashes.

Existing BFT systems cannot completely satisfy the above requirements. PBFT [1] creatively adopts keyed-hash message authentication codes (HMACs) to authenticate messages among servers and clients. It avoids digital signatures, the main performance bottleneck in previous systems [12,13]. Zyzzyva further improves the performance by using speculation [4]. PBFT and Zyzzyva reach their peak performance when no faults exist, but the throughput drops to zero if any faulty client crafts series of requests with partially-correct HMACs [9].

To defend partially-correct HMAC attack, Aardvark uses digital signatures instead of HMACs for authenticating clients' requests [9]. However, it offers poor fault-scalability as PBFT, because their agreement protocols require several rounds of server-to-server communications, introducing a total communication overhead of $\mathcal{O}(n^2)$. Q/U [2] proposed a quorum-based architecture with good fault-scalability and ideal response time (i.e., only two one-way latencies) when no fault nor contention exists; however, faulty clients can halt the services [2,9] by fabricating concurrent requests to a set of servers that intersect with every other quorum but never form one by themselves.

Hybrid BFT solutions such as HQ [3] and Aliph [6], employ the efficient quorum-based approach in the case of no faults nor contention, and switch to agreement-based protocols when there are contending requests. However, the client-directed switch requires a (non-faulty) client to collect digital signatures from servers, and push servers into the same mode through extra communicating steps. In addition, servers do not respond to other clients' requests until the mode switch completes. Therefore, a faulty client can sharply reduce the throughput by intentionally submitting concurrent requests, forcing the system frequently switch to the less efficient agreement mode at a heavy switch cost.

In this paper, we propose a BFT system called EFS, to provide fault-scalability and robust efficiency. EFS designs a server-directed lightweight switch to integrate two BFT approaches. When there is no contending request, the servers work in the efficient quorum mode. And when contention appears due to concurrent requests, the servers switch to the agreement mode to provide services with a predictable yet limited performance degradation.

EFS provides better efficiency and fault-scalability at a cost of using more servers. While some BFT systems require $3f+1$ servers (e.g., PBFT, HQ, Aliph and Aardvark) to tolerate f faulty ones, EFS needs $5f+1$ servers, similar to Q/U which is proposed for massive-scale distributed services in open clusters or over the WAN. In such settings, backup servers are sufficient. Moreover, as the server cost has decreased remarkably with the development of virtualization, we believe that the implementation cost should not be the main obstacle to affect the adoption of more efficient and reliable BFT solutions.

EFS achieves its peak performance in the case of no faults nor contention. Moreover, it offers robust efficiency under the following adversarial scenarios:

- *Benign contention from correct clients.* In this case, EFS adopts a similar approach as proposed in the FaB agreement algorithm [14] to reach consensus on the execution order, which is proved to be optimal to reach asynchronous Byzantine consensus in the number of communication steps (i.e., two steps).
- *Partially-correct HMAC attacks from faulty clients.* In EFS, a request is sent to a quorum of $4f+1$ servers to verify the correctness of HMACs and eliminate these attacks.
- *Malicious contention from faulty clients.* The switch in EFS is lightweight: it does not require costly digital signatures nor extra communication steps. Moreover, the switch is initiated by servers. Therefore, during mode switch, the servers can automatically collect later contending requests to enable batch executions, resulting in smaller operation delays.
- *The increasing number of server faults.* EFS's two work modes are both fault-scalable: each server responds to clients directly in the quorum mode, while in the agreement mode, it avoids expensive server-to-server broadcasting in FaB.

In particular, EFS adopts the quorum protocol of Q/U in the contention-free case, and implements the FaB algorithm in a fault-scalable manner in the agreement mode. Through the integration, the protocol of Q/U is also improved in EFS: (a) the observation of the system are excluded in each operations' logic timestamps to save the communication cost; and (b) the support of multi-object services is facilitated in terms of update locks and contention resolving. When contention appears, EFS servers not only implement the FaB algorithm to agree on the execution order, but also ensure the consistency between two work modes. A side benefit of the server-directed switch is *client-transparency*: clients use a uniform protocol for the two different modes and do not involve in the switch.

We implement the EFS prototype. Performance is evaluated and compared with other BFT systems. The evaluation results demonstrate that EFS achieves efficiency and fault-scalability under various scenarios.

2 Background and Related Works

The idea of using SMR to tolerate arbitrary faults in a subset of servers was proposed in 1980s [15]. Various BFT protocols [16,17] have been proposed to reach consensus on the execution order among servers [8,12,18,19]. However, due to the poor efficiency, the concept was considered impractical until Castro and Liskov's work on PBFT [1]. PBFT prototype used four servers to tolerate one faulty server, and achieved a performance that was only 3 % worse than the standard unreplicated system when no faults exist. While PBFT appears to be efficient, there is an ongoing competition on improving the efficiency of BFT systems. Among these solutions [2–6,9], Q/U [2] firstly proposed a quorum-based BFT system that provides *fault-scalablity*. However, Q/U suffers from the live-lock problem under concurrency workloads [2].

To address the performance limitations, HQ [3] proposed a hybrid approach, it employs PBFT to resolve contention and thus avoids the live-lock problem in Q/U. However, the adoption of PBFT makes HQ not fault-scalable in the contending case. Moreover, when contention exists, HQ needs to firstly achieve a consensus on the latest valid state, introducing extra rounds of processing. In this paper, we present EFS which avoids server-to-server broadcast communication either contention exists or not. EFS is more fault-scalable and efficient than HQ in both the contention-free and contending cases (see Table 1). Aliph [6] is an integrated system that combines three BFT approaches. However, Aliph is not fault-scalable in the presence of contention or failures (see Table 1).

Zyzzyva [4] avoids the server-to-server broadcasting in PBFT. But it depends on the client to detect inconsistency among servers, and requires three message delays for a request when no faults exist. It is less efficient than EFS, which only needs two in the quorum mode. Aardvark [9] improves PBFT by eliminating the optimization designs that lead to significant decrease of efficiency in the presence of faulty entities. It offers a stable performance but its peak throughput is much less than other protocols. The faulty primary is also considered in Aardvark and Prime [20], countermeasures are designed and verified. EFS pays more attention to impact from (faulty) clients than that from servers, and shares the same spirit with Aardvark somehow: the execution path is not determined by clients.

FaB [14] is the first protocol that reaches asynchronous Byzantine consensus in two communication steps when no faults exist. In FaB, each server only accepts the first value proposed by the primary and then sends responses directly. EFS cannot simply adopt FaB in the contending case, as it has to keep consistency between two work modes. Further, in the presence of faulty servers, the primary may need to modify the proposal to make it accepted by enough servers (detailed in Sect. 3.3). Moreover, we apply the theoretical FaB to provide practical BFT service and improve its fault-scalability.

CBASE [21] exploits application-specific parallelism for high throughput in BFT systems. [22] separates the servers executing the agreement protocol from the ones executing operations to reduce the cost for service replication, and deploys a firewall matrix to provide BFT confidentiality. These mechanisms can work compatibly with EFS, and will be included in our future work.

3 The EFS Protocol

3.1 System Model

Objects and Versions. In EFS, objects are replicated across n servers. Clients issue requests to servers to perform a *query* (i.e., read-only) or an *update* (i.e., modify) operation on an object, according to the *argument* in the request. The operation is completed once at least $4f + 1$ different servers having executed it.

Whenever a server executes an `update` operation on an object, it generates a new *object version* and assigns a *logical timestamp* (LT) to this new version. The operation is executed *conditioned on* the current object version with timestamp LT_{CO}, so the timing of the execution can be identified by the pair of timestamps (LT, LT_{CO}). LT is in the form of $(seq, clientID, method, argument)$, where seq initiates from 0 and increases by 1 after executing an operation on the object (i.e. $LT.seq = LT_{CO}.seq + 1$), $clientID$ is the identifier of the client who issues the request, $method$ is the exact method invoked on the object, and $argument$ contains the argument needed by the method. The comparison between LTs (i.e., $=$, $<$ and $>$) is based on comparing seq, $clientID$, $method$ (lexigraphy comparison) and then $argument$ (lexigraphy comparison) in order. For each object, a server maintains a *replica history* with ordered timestamp pairs to represent the execution order of `update` operations on this object. Initially, for each object, the first entry of the replica history on every server is set as $(0, \perp)$.

In response to a client's request, a server replies with its replica history. On the client side, each client maintains an *object history set* (OHS), which is an array of replica histories indexed by server. OHS is also included in the client's request to the servers. In the Byzantine faulty environment, not all servers could complete the requests, therefore, an OHS represents each client's local view of the server states. To compare two OHSs (i.e., $=$, $<$ and $>$), we compare the largest LT that appears at least $4f + 1$ times in each OHS.

Each server classifies the received OHS and determines the corresponding work mode. If there are different LTs conditioned-on the same LT_{CO} in the OHS but none of them appears at least $4f + 1$ times, this OHS is considered *incomplete*, which makes the server switch to work in the agreement mode (see Sect. 3.2.1); otherwise the OHS is classified as *complete*, and the server works in the quorum mode (see Sect. 3.2.1). To preserve the execution order of completed operations during the mode switch, each server caches the largest complete OHS that it has received (denoted as OHS_s) and the operation conditioned on OHS_s that it has executed (denoted as O_s). At the beginning, both OHS_s and O_s are set to null.

Faulty Entities. EFS builds SMR services using $n = 5f+1$ independent servers. Both the servers and clients might be Byzantine faulty and exhibit arbitrary, potentially malicious behaviors. Faulty entities (servers and clients) might also collude with each other. EFS can tolerate an arbitrary number of faulty clients and up to f faulty servers.

Message Authentication. Servers and clients are connected with unreliable, asynchronous links. As a result, messages may experience dynamic transmission delay, or be duplicated, reordered or dropped by the attacker. We adopt the fair link assumption [22] that a message will be received if it is sent sufficiently often. To prevent forged messages from faulty entities, we employ point-to-point message authentication using a keyed hash (HMAC). A secret key μ_{ij} is shared between the entity i and j to create the HMAC of message m, denoted as $[m]_{\mu_{ij}}$. Each server i holds a private signature key (denoted as σ_i) and signs a message m as $[m]_{\sigma_i}$. We will present digital signature based authentication for replica history and communication between servers in Sects. 3.2.1 and 3.2.2, and then discuss how to avoid these signatures in Sect. 3.3.

3.2 The Protocol

3.2.1 Contention-Free Case

In the contention-free case, EFS adopts a quorum-based protocol similar to Q/U [2]. Clients send requests in the form of $[clientID, method, argument, OHS_c]$ to a set of servers to invoke a *method* on an object, where OHS_c is the cached OHS. Each server first verifies the request with the HMAC, and compares its cached replica history with the corresponding one in the received OHS_c. A matched replica history denotes the client's view of the object state is consistent with the server's actual state. Then, the server sanitizes the entries of OHS_c (i.e., replica histories of other servers) by verifying the signature of each entry and removing invalid ones. The server classifies the sanitized OHS_c only when there are at least $4f+1$ entries remained. If OHS_c is *complete*, the server works in the **quorum mode** to execute the requested operation; otherwise, the server switches to the **agreement mode** to resolve contention (Sect. 3.2.2).

In the quorum mode, if this is an `update` operation, a new object version will be generated at each server independently, and assigned with LT_{new}. The new object version is conditioned on the version with the largest LT that appears at least $4f+1$ times in the OHS_c, the server adds (LT_{new}, LT) to its replica history, update the cached OHS_s and O_s. Then, each server returns the execution result and the signed replica history to the client. If at least $4f+1$ servers are working in the quorum mode, a same LT_{new} will be generated at different servers and a quorum of success responses with consistent replica histories will be returned to the client. The client updates its OHS with the received replica histories.

However, if the corresponding replica history extracted from OHS_c is outdated, the server will return a failure response with its current replica history to the client. Moreover, in the sanitization stage, if the sanitized OHS_c has less than $4f+1$ entries, it will be returned to the client in a failure response. If not enough (less than $4f+1$) success responses are received by the client, the request fails and the client needs to re-send the request with the updated OHS. Although the reason of failure varies, the client is not expected to distinguish the cause or take different actions. Therefore, the complexity at the client is reduced.

In the contention-free case, an operation is finished in *two* message steps if the client receives at least $4f+1$ success responses, otherwise, it will take *two* additional message steps to update the entries of OHS_c. Therefore, EFS quorum mode is as efficient as Q/U and more efficient than most of the other protocols.

3.2.2 Contending Case

Contention occurs if the OHS contained in the client's request is incomplete. It may be caused by multiple operations conditioned on a same version but fail to complete due to concurrent requests, network failures or faulty entities. To resolve the inconsistency, the server should switch to the **agreement mode**.

Each server moves through a succession of configurations, known as *views* [1]. A view is identified by a consecutively increasing sequence number v, initially set to 0 and increased by 1 for each *view change* (see Sect. 3.2.4). Servers store the current view number locally. In each view, the server with identifier p, where $p \equiv v \pmod{n}$, is the *primary*, and the others are called *backups*. The primary is responsible for keeping consistency between the two work modes, proposing an execution order for contending requests, and coordinating the backups to reach an agreement in *five steps of message exchanges*, as shown in Fig. 1:

1. *Initiate:* whenever a server detects contention in a request from client c, it switches to the agreement mode. It sends an `initiate` message, with cached OHS_s and O_s, to the primary of the current view.
2. *Propose:* the primary maintains an array called $InitiateArray$ to store the `initiate` messages from the servers. Upon receiving $4f+1$ valid `initiate` messages, the primary selects the largest complete OHS (denoted as $OHS^{[p]}$) from all the received OHS_s. Then, the primary proposes an execution order

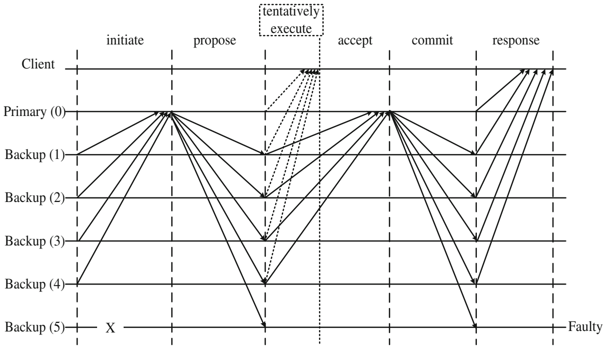


Fig. 1. EFS's process in the contending case

(denoted as $\overrightarrow{\mathbb{O}}^{[p]}$) for all the contending operations contained in at least $f+1$ `initiate` messages. The operation that appears at least $2f+1$ times in the $4f+1$ `initiate`s is selected as the first operation of $\overrightarrow{\mathbb{O}}^{[p]}$, and the order of other operations is determined arbitrarily. Finally, the primary includes $OHS^{[p]}$, $\overrightarrow{\mathbb{O}}^{[p]}$ and $InitiateArray$ in a `propose` message to all the backups.

3. *Accept:* a backup accepts the proposal if (1) $InitiateArray$ contains messages from at least $4f+1$ servers, (2) the proposed $OHS^{[p]}$ matches at least $2f+1$ OHS_s in $InitiateArray$, and (3) the operations in $\overrightarrow{\mathbb{O}}^{[p]}$ are selected and ordered correctly. For an accepted `propose`, the backup generates signatures of $OHS^{[p]}$ and $\overrightarrow{\mathbb{O}}^{[p]}$ and sends them in an `accept` message to the primary.
4. *Commit:* the primary stores the correctly signed `accept` messages in an $AcceptArray$. If at least $4f+1$ servers have accepted the proposal, the primary will send a `commit` message with $AcceptArray$ to all servers.
5. *Response:* finally, each server validates the integrity and consistency of the $OHS^{[p]}$ and $\overrightarrow{\mathbb{O}}^{[p]}$ in the `commit` message. Then the server retrieves the object version identified by $OHS^{[p]}$ and executes operations in $\overrightarrow{\mathbb{O}}^{[p]}$ sequentially to reach a consistent state, and switches back to the quorum mode.

The procedure is illustrated in Fig. 2. In the agreement mode, non-faulty primary can coordinate an orderly execution of contending operations at all servers and eventually bring the system back to a consistent state.

To improve efficiency, EFS supports *tentative execution*, which is also used in [1,14]. The tentative execution allows the backups to tentatively execute the operations in $\overrightarrow{\mathbb{O}}^{[p]}$ if it accepts the `propose`, and return the tentative results to the clients before a consensus is reached. The tentative execution is supported because EFS allows a client to receive multiple responses from a same server for a given operation, and overwrite an older response with a newer one. With tentative execution, all servers can reach a consistent state in two message delays (instead of four delays in the above flow in Fig. 1) with a correct `propose`.

3.2.3 Mode Switch

Unlike other BFT protocols, no special request from the clients is needed for mode switch. This avoids purposeful or delayed switch manipulated by a faulty client. Mode switch is only triggered by messages received at the servers:

To switch from the quorum mode to the agreement mode, a server needs to receive either a request with an incomplete OHS_c from a client, or a correct `propose` message from the primary. The primary, to be invoked to coordinate contention resolution, needs to receive $4f+1$ valid `initiate` messages.

To switch from the agreement mode to the quorum mode, the primary needs to receive $4f+1$ correct `accept` messages. A backup needs to receive either a correct `commit` message or a complete OHS that is larger than the cached OHS_s.

Due to network failures or faulty servers, a server may fail to receive the expected messages in time and thus stays in the less efficient agreement mode. To mitigate the performance degradation, a backup will call the PullCommit

```
procedure Backup.OnContend($OHS_s$, $O_s$, $i$)
    send $[\texttt{initiate}, i, OHS_s, O_s]_{\sigma_i}$ to $p$
    if timeout for valid update-OHS or commit then
        if (PullCommit() fails) then
            StartViewChange($v + 1$, $OHS_s$, $O_s$, $i$)
end procedure

$InitiateNum = 0$
$InitiateArray[n]$
procedure Primary.OnInitiate(void)
    if signature of $[\texttt{initiate}, i, OHS_s, O_s]$ error then
        return
    $OHS_p = OHS_s$ cached by primary
    $OHS_i = \texttt{initiate}.OHS_s$
    if $OHS_i$ is not complete || $OHS_i < OHS_p$ then
        reply $[\texttt{update-OHS}, OHS_p]_{\sigma_p}$
        return
    if $InitiateArray[i] == null$ then
        $InitiateNum{+}{+}$
    $InitiateArray[i]$ = (the signed initiate from server $i$)
    if $InitiateNum > 4f$ then
        ProposeOrder()
end procedure

procedure Primary.ProposeOrder(void)
    Set:=$\{OHS$: Order($OHS$,$InitiateArray$)$>2f\}$
    $OHS^{[p]}$ = LatestOHS(Set)
    if $\exists O$ : Order($O$,$InitiateArray$)$> 2f$ then
        $\overrightarrow{\mathbb{O}}^{[p]}[0] = O$
    define order of other operations in $\overrightarrow{\mathbb{O}}^{[p]}$
    for $i \leftarrow [0, n-1]$ do
        send $i$ $[\texttt{propose}, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}, InitiateArray]_{\sigma_p}$
end procedure

procedure Backup.PullCommit(void)
    send $[\texttt{pull-commit}, OHS_s, OHS_c]_{\sigma_i}$ to all servers
    if timeout for complete $OHS$ or commit then
            return true
    else
            return false
end procedure

procedure Order($element$, $Set$)
    /* Determine order of $element$ in the $Set$ */
end procedure
procedure Backup.OnPropose(void)
    if propose is not correctly signed then
        return
    if any initiate signed invalidly in $InitiateRecord$ then
        return
    if propose message accepted then
        send $[\texttt{accept}, i, p, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}]_{\sigma_i}$ to $p$
end procedure

$AcceptNum = 0$
$AcceptArray[n]$
procedure Primary.OnAccept(void)
    if signature of $[\texttt{accept}, i, p, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}]$ error then
        return
    if $AcceptArray[i]$ == null then
        $AcceptNum{+}{+}$
    $AcceptArray[i] = [\texttt{accept}, i, p, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}]_{\sigma_i}$
    if $AcceptNum \leq 4f$ then return
    for $i \leftarrow [0, n-1]$ do
        send $i$ $[\texttt{commit}, p, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}, AcceptArray]_{\sigma_p}$
end procedure

procedure Backup.OnCommit (void)
    for $i \leftarrow [0, n-1]$ do
        if Verify($i$,$[i, OHS^{[p]}, \overrightarrow{\mathbb{O}}^{[p]}]$,$AcceptArray[i]$) then
            $CorrectNum{+}{+}$
    if $CorrectNum > 4f$ then
        execute in order defined in $\overrightarrow{\mathbb{O}}^{[p]}$
end procedure

procedure Backup.OnPullCommit($i$)
    $OHS_l = \texttt{pull-commit}.OHS_s$
    if $OHS_s > OHS_l$ then
        reply $[\texttt{update-OHS}, OHS_s]_{\sigma_i}$
    if has received valid commit message then
        reply $[\texttt{push-commit}, \texttt{commit}$ message$]_{\sigma_i}$
    if $\texttt{pull-commit}.OHS_c$ is incomplete then
        OnContend($OHS_s$, $O_s$, $i$)
end procedure

procedure Verify ($i$,$m$,$sig$)
    /*Return true if $sig$ is signature of $m$ signed by $i$ */
end procedure
```

Fig. 2. Pseudo-code of EFS in contending case

process, if it does not switch to the quorum mode after pre-defined timeouts. In the PullCommit process, each server probes other servers to "pull" valid `commit`s from at least $3f + 1$ servers, and uses the consistent `commit` to update its own status. If the PullCommit process fails, which indicates the primary is faulty, the backup will select a new primary through *view change.*

3.2.4 View Change

If the primary is faulty, the view change will be triggered to select a new primary.

1. When a backup fails to receive a valid reply from the primary p_v of view v after the timeout, it notifies p_{v+1}, the primary of view $v + 1$, with a signed `start-vc` message $[\texttt{start-vc}, v + 1, OHS_s]$.
2. The new primary p_{v+1} validates the message and checks if the included OHS_s is the latest. Once p_{v+1} receives valid `start-vc` messages from $3f + 1$ servers, it sends a `new-view` message to all the servers. The $3f + 1$ `start-vc` messages

are included so that the new view will be unique at the presence of f faulty servers. p_{v+1} keeps sending the new-view message until it receives $4f + 1$ valid initiate messages.

3. Any server receiving a valid new-view message will mandatorily switch to work in the agreement mode. It updates its view number to $v + 1$, and constructs an initiate message to the new primary.

It might be possible that the new primary is still faulty and tampers with the view change process. So, the backups use another timer to limit the delay between sending the start-vc and receiving the new-view. Once timeout, the backup firstly actively probes the new-view from at least $3f + 1$ servers. If it fails, it considers the current view change as unsuccessful, and invokes a new view change to view $v + 2$. A backup may receive multiple new-view messages for different views, it only needs to respond to the primary of greater view.

3.3 Avoiding Signing

One important optimization presented in PBFT is using an HMAC array to replace the expensive digital signatures for message authentication. Similar ideas are adopted in the hybrid BFT systems [3,6]. However, since the hybrid systems rely on the clients to notify the servers about mode switch, they still employ the expensive digital signatures during mode switch, leaving a vulnerability for faulty clients frequently switching the work mode. Differently, EFS enables the servers to initiate mode switch. Therefore, it is possible to authenticate replica history, initiate, propose, accept, and commit messages with an HMAC array (denoted as authenticator), eliminating the signatures in the two work modes and mode switch. Each entry of authenticator is a HMAC for a same message using different keys shared between the sender and each of the other servers.

The challenge exists in this replacement. A faulty entity may deliberately generate valid and invalid HMACs for a same message to create inconsistent observations for different servers. First, a faulty entity may create valid and invalid HMACs of its replica history. Since OHS entries with invalid HMACs will be excluded during sanitization, the faulty entity can manipulate the OHS to be complete or incomplete to push a certain number of servers into the agreement mode while keeping the others in the quorum mode. If at least $4f + 1$ non-faulty servers consider the OHS incomplete and switch to the agreement mode, the contention can be resolved among $4f + 1$ servers. However, if less than $4f + 1$ non-faulty servers consider the OHS incomplete, there will not be enough servers to resolve the contention in the agreement mode. Here, we consider two different scenarios: (1) more than $3f$ but less than $4f + 1$ non-faulty servers receive incomplete OHS, and (2) no more than $3f$ non-faulty servers receive incomplete OHS. In the first scenario, the primary may never receive $4f + 1$ initiate messages needed for the *propose* stage. To cope with this problem, the primary is required to send the *InitiateArray* to the remaining servers that did not send the initiate, after receiving $3f + 1$ initiate messages. A non-faulty server verifies the $3f + 1$ entries of the *InitiateArray* (using authenticators), and replies a dummy `initiate` (whose operation is set to null) to the primary. In this way, the primary can actively "pull" `initiate`s to meet the $4f + 1$ requirement. In the second scenario, we modify the PullCommit process to let each server in the agreement mode "pull" either a valid `commit` or a same complete OHS which is larger than its own OHS_s, from at least $3f + 1$ servers. The latter can trigger a server to switch back to the quorum mode.

Faulty servers may send `initiate` and `accept` messages with valid HMACs for the primary but invalid HMACs for some backups, which in turn will report the suspicious messages to the primary. A message suspected by at least $f + 1$ different servers will be excluded from further processing. Furthermore, a faulty primary may send different *InitiateArray* to different backups, trying to construct two different *AcceptArray*s and make them accepted by different backups. To solve this problem, each `propose` should have a monotonically increasing propose number (denoted as *pn*), initially set to 0 in each view. The backup tracks the largest *pn* it has received in pn_b, accepts only the `propose` with a larger *pn* and *AcceptArray* for the `propose` whose *pn* is no less than pn_b.

During the mode switch, a primary may receive more than $4f + 1$ `initiates`. It is possible that there exist two different operations, each appearing in more than $2f + 1$ `initiates`. In such a case, we select the operation with the smaller hash value as the first operation in $\overrightarrow{\mathbb{O}}^{[p]}$ without loss of generality. However, as the faulty servers can insert invalid HMACs into the `authenticator` for `initiate`, some non-faulty backups may generate a different observation from the one of the primary (and others) so that they will suspect and refuse to accept the `propose`. As a result, the primary will fail to receive enough ($\geq 4f + 1$) `accept` needed in the *commit* step. To cope with this problem, we let a backup accept a latest `propose` that is also accepted by at least $3f + 1$ servers, considering the faulty server can only tamper with up to f backups in this type of attacks (otherwise the `initiate` will be reported suspicious by $f + 1$ backups and excluded from the processing).

3.4 Optimization

Reducing Communication. Each server can send clients the portion of their replica histories with LTs no smaller than the largest LT that appears at least $4f + 1$ times in its cached OHS_s, which reduces the size of the OHS transmitted between clients and servers. Moreover, the size of LT in the replica history can be reduced by replacing the `argument` and `operation` with their hash value respectively. In the agreement mode, each backup can send its replica history instead of the entire OHS_s to the primary, which will determine $OHS^{[p]}$ using the latest complete elements of a quorum of replica histories.

Automatic Batching. Servers in the agreement mode can still respond to the requests from clients, rather than locking the services, which makes the system more stable and eliminates the waiting time back to the quorum mode. In the agreement mode, if a request arrives before the `propose` message, each server automatically batches it in $\mathbb{O}_s$ and sends $\mathbb{O}_s$ to the primary who will propose the executing order for the batched operations that appear in the $\mathbb{O}_s$ from at least $f+1$ servers. If an operation in $\mathbb{O}_s$ is included in $\overrightarrow{\mathbb{O}}^{[p]}$, each server executes it in the proposed order and sends the result to the client. For those requests that arrive after the `propose` message and are not included in the $\overrightarrow{\mathbb{O}}^{[p]}$, each server returns its latest replica history to clients after executing all operations in $\overrightarrow{\mathbb{O}}^{[p]}$.

Multi-object Services. Multi-object `update` operation requires to update a set of objects as a whole. EFS supports multi-object `update` using lock mechanism. The server locks local object version only when all the OHS are complete, and releases the lock after finishing the execution. EFS does not enforce any lock order to avoid deadlock, as when contending requests exist, EFS will switch to the agreement mode and let the primary decide the execution order.

4 Correctness Analysis

4.1 Correctness Against Byzantine Faults

In SMR systems, *correctness* means the system should provide *safety* and *liveness* at the same time. EFS ensures correctness by requiring non-faulty servers to agree on a same conditioned-on version, before executing any operation O. To ensure correctness, EFS needs at least $n = 5f + 1$ servers to tolerate up to f faulty servers. To prove this, let us denote the latest completed operation as O'. It should be executed on a quorum of q servers, and observed by the primary from the `initiate` messages sent from another quorum of q servers in the agreement mode. In the worst case, the two quorums merely overlap, with the intersection of at most $2q-n-f$ non-faulty servers. Furthermore, to ensure O' is the conditioned-on operation of O, O' should be observed in more than half of the `initiate` messages, that is $(2q-n-f) > (n-f)/2 \iff n > 5f$.

Safety. The *safety* property requires that in any case, different requests conditioned on a same LT_{CO} should never be finished. In EFS, safety is guaranteed by ensuring that any completed operation O that is conditioned-on LT_{CO} is the only operation that can be completed on the LT_{CO}. We prove this property for EFS working in both modes, and in the transient state.

EFS works in the quorum (agreement) mode if at least $3f + 1$ non-faulty servers work in the quorum (agreement) mode; otherwise, EFS works in the transient state. For an operation O to be completed, at least $3f + 1$ non-faulty servers should execute O conditioned on LT_{CO}, and the client can receive at least $4f + 1$ consistent replies. Therefore, in the *transient state*, no operation can be completed due to the lack of enough consistent replies. In the *quorum mode*, an operation O'' requested to be conditioned on LT_{CO} (i.e., claiming LT_{CO} as the latest local timestamp) will receive failure messages with newer replica histories from at least $2f + 1$ non-faulty servers to update the dated OHS.

When EFS works in the *agreement mode*, at least $2f+1$ non-faulty servers will include O in the `initiate` messages as O_s and O is the only operation appeared at least $2f + 1$ times in *InitiateArray*. If the primary is non-faulty, O will be the first operation in $\overrightarrow{\mathbb{O}}^{[p]}$, and consequently the only operation to be finished conditioned on LT_{CO}. On the contrary, if the primary is faulty, its proposal will not be accepted by at least $2f + 1$ non-faulty servers, and thus triggers the view change protocol. Safety of the view change protocol is ensured by proving the system cannot be in two different views at the same time – different non-faulty servers are in different views due to different valid `new-view` messages. As the `new-view` message consists of at least $3f + 1$ valid `start-vc` messages, if there are more than one valid `new-view` messages, it means a least one non-faulty server participated in two view change at the same time, which never happens.

Liveness. *Liveness* denotes that no matter which mode the system is in, a non-faulty client can eventually receive $4f + 1$ consistent responses by sending its request repeatedly. Therefore, we prove the liveness of EFS in different cases.

Obviously, when there is no contending request, at least $4f + 1$ non-faulty servers will receive and execute the request if OHS_c from the client is latest and object versions at servers are also the latest. Otherwise, a failure message will be returned to update OHS_c, or object synchronization will be called to obtain the latest object version from $f + 1$ different servers. After the extra round of updating, the requested execution is performed.

When there are two contending requests from clients c and c', respectively, at most $4f$ non-faulty servers receive the request with a latest OHS_c, from client c prior to the request from c'. Otherwise, c can receive $4f + 1$ consistent responses to finish the request. Servers that receive the request from c later than c' return their replica histories to c to construct a new OHS'_c indicating contention. Then, c keeps sending new requests with OHS'_c, which will eventually trigger at least $4f + 1$ non-faulty servers to move into the agreement mode (and send out $4f + 1$ `initiate` messages). If the primary is non-faulty, liveness of the agreement protocol is guaranteed by non-faulty backups sending client c the consistent responses after they having received $4f + 1$ valid `accept` messages contained in the `commit` message (given at most f faulty servers).

In cases where the primary is faulty, non-faulty backups may not receive a valid `commit` message and thus trigger the view change by timeouts. Liveness of the view change process is guaranteed by pulling `commit` messages from others. A non-faulty server verifies at least $3f+1$ non-faulty servers in the contending case have not received `commit` messages if it fails to get a larger complete OHS or a valid `commit` messages. This guarantees that these $3f+1$ non-faulty servers in the contending case eventually help the new primary to construct a valid `new-view` message. Similarly, if any initiating backup fails to receive the `new-view` message before timeout, it will pull the `new-view` messages from others to decide to invoke further view changes. As the view change protocol will eventually succeed after a non-faulty primary is agreed, the liveness of EFS under view change is proven.

4.2 Efficiency Against Faulty Clients

Faulty clients may harm the efficiency of the system while the correctness can still be provided, for example by partially-correct HMAC attack or constructing malicious contention [9]. However, EFS still provides robust efficiency even when faulty clients exist.

Partially-Correct HMAC Attack. To launch this attack, faulty clients send requests with correct HMACs for some servers while incorrect for others, which results in contention and makes the system switch to work in the agreement mode seamlessly. In the agreement mode, faulty clients cannot start this attack, as the `primary` proposes $\overrightarrow{\mathbb{O}}^{[p]}$ according to the `initiate` messages from servers, who firstly check the HMACs of the requests from clients.

Malicious Contention. The faulty clients may attempt to degrade the system performance by triggering the system switch to the agreement mode frequently using malicious contention. However, due to the lightweight switch and efficient agreement-based protocol in the agreement mode, the performance degrades much gracefully under this attack.

5 Performance Analysis and Evaluation

5.1 Efficiency Under Faulty Client Attacks

Most existing BFT protocols suffer from faulty client attacks. Faulty clients can make the throughput of PBFT, Zyzzyva and Q/U drop to 0 [9]. Aardvark provides robust efficiency against faulty clients at the cost of degradation of peak throughput in the case of no faults. Only hybrid protocols can address such faulty client attacks by switching to a less efficient agreement mode at a mode switch cost, while keeping efficient when no faults exist. To evaluate the performance of EFS under the attack, we first compare EFS with two existing hybrid BFT protocols, HQ [3] and Aliph [6], and analyze the performance at the quorum mode and the agreement mode, and the cost due to mode switch.

Table 1. Cryptographic operations and messages used in three hybrid BFT protocols.

	HQ [3]			Aliph [6]				EFS		
	quorum	agreement	switch	quorum	chain	backup	switch	quorum	agreement	switch
MAC	$4+4f$	$2+\frac{8f+1}{b}$	$2f+1$	$6f+4$	$1+\frac{f+1}{b}$	$2+\frac{8f+1}{b}$	1	$8f+2$	$1+\frac{15f+3}{b}$	$4f+2$
Sign	0	0	5	0	0	0	1	0	0	0
Verify	0	0	$5f+4$	0	0	0	$2f+1$	0	0	0
Messages	4	4	6	2	$2+3f$	4	2	2	4	1

All three protocols work in the quorum mode in the case of no faults nor contention, and switch to the agreement mode (the chain mode in Aliph) when receiving contending requests. Aliph further switches to the backup mode even only one faulty entity exists, while others remain in the agreement mode. We summarize numbers of cryptographic operations (i.e., generating MAC, digital signature signing and verification) in the bottleneck server and one-way message transmits in the critical path required by each protocol for per request in Table 1, where b refers to the batch size. From Table 1, we see that HQ and Aliphi require expensive signature operations and more messages for mode switch, and thus

Table 2. The simulated processing time when $f = 2$ without batch (in ms).

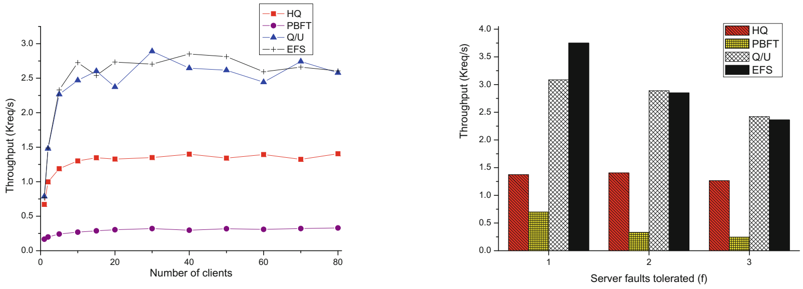
	HQ [3]			Aliph [6]				EFS		
	quorum	agreement	switch	quorum	chain	backup	switch	quorum	agreement	switch
MAC	0.072	0.114	0.03	0.096	0.024	0.114	0.006	0.108	0.204	0.06
Sign	0	0	11.685	0	0	0	2.337	0	0	0
Verify	0	0	1.82	0	0	0	0.65	0	0	0
Latency	1.348	1.348	2.022	0.674	2.696	1.348	0.674	0.674	1.348	0.337
Total	1.42	1.462	15.557	0.77	2.72	1.462	3.667	0.782	1.552	0.397

need a much longer duration to complete the switch (as shown in Table 2, 39.18 and 9.24 times of the time needed by EFS, respectively).

To measure the processing time per request, we simulate it by summarizing the time for cryptographic operations and message transmitting in the critical path. We adopt the commonly used signature and MAC algorithms implemented in OpenSSL v1.0.0i. Signing and verification for 1024-bit RSA require 2.337 ms and 0.13 ms, respectively, while SHA-1 digest of 1 KB blocks only needs 0.006 ms. Without loss of generality, we set b to 1. Assuming to tolerate 2 faulty servers (i.e., $f = 2$), the simulated processing time per request are shown in Table 2. The result shows that EFS has a similar performance as HQ and Aliph in the agreement mode, and outperforms HQ in the quorum mode.

5.2 Performance Evaluation

Settings. All the experiments ran with 16 servers and 80 logical clients in an isolated 1000 Mbps Ethernet. Servers ran on identical workstations with an Intel S1260 (2.0 GHz) CPU and 4 GB of memory. Logical clients were hosted on two machines, each with 8 GB of memory and two Intel Xeon E5620 (2.4 GHz) CPU. Network I/O and CPU process on the clients were not found to be a limiting factor in any of our experiments. The communication between clients and servers is implemented via TCP. A SHA-1 based HMAC is used to authenticate messages.



(a) Throughput with varying number of clients when $f = 2$.

(b) Peak throughput with varying number of server faults.

Fig. 3. Throughput of EFS, PBFT, Q/U and HQ in the contention-free case.

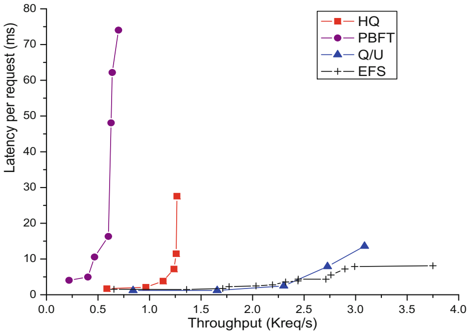


Fig. 4. Latency vs. throughput ($f = 1$).

We evaluate the performance of EFS on a counter service for the `update` operation in the contention-free case. Then we measure the extra time to resolve contention needed by EFS using micro-benchmarks [1]. We use 0/0 (i.e. the client sends a null request and receives a null reply) and 4/0 (i.e. the client sends a 4KB request and receives a null reply) micro-benchmarks to measure the overhead due to extra BFT computation and communication. For comparison purpose, we also evaluate the performance of PBFT, HQ and Q/U when all of the optimizations are adopted. We adopt an optimized PBFT implementation [3] which replaces the broadcast communication with point-to-point communication for better fault-scalability, Q/U version 1.3 [2] and an HQ implementation [3].

Contention-Free Case. In the contention-free case, each client with the latest OHS keeps issuing requests until it receives responses from at least $4f+1$ servers. As the provided implementions [3] of PBFT and HQ do not support the micro-benchmarks [1], we measure the throughput and latency using the counter server.

Figure 3(a) plots the throughput of EFS, PBFT, HQ and Q/U, with varying number of clients when $f = 2$. We observe that EFS achieves a similar peak throughput as Q/U (at most 1.3 % difference) and significantly outperforms both PBFT and HQ. The main reason is that each server in PBFT and HQ need to send $3f + 1$ and 2 messages respectively in responding to a client request, while the server in EFS only requires to send one message and thus greatly reduces the overhead due to network latency. Fig. 3(b), the throughput of EFS decreases sub-linearly with increasing number of tolerated faults.

We also study the response time of the four protocols as a function of the achieved throughput. As shown in Fig. 4, EFS and Q/U achieve consistently lower response time than HQ and PBFT. Moreover, EFS achieves lower response time than Q/U at a higher throughput (larger than 2.5 K requests/s) as OHS is excluded in LT which reduces the total communication in the system.

Contending Case. In this section, we study the additional time needed for the whole contention resolution which includes mode switch and making consensus on the execution order for contending requests. In EFS, when a server detects contending requests, the system switches to the agreement mode to reach a consensus coordinated by a primary. EFS supports *tentative execution* (denoted as EFS(opt)) in which contention can be resolved once $4f + 1$ non-faulty servers

Table 3. Average time to resolve contention (in ms).

Faults tolerated		1	2	3
EFS (opt)	0/0	4.5089	7.5551	10.2612
	4/0	5.5036	9.5530	12.196
EFS (full)	0/0	7.4030	13.0764	14.9806
	4/0	10.2309	13.5275	16.196

receive the same `propose` message. However, if a faulty primary is selected by chance, the contention can only be resolved after a `commit` is received (denoted as EFS(full)). We measure the time for the servers to reach a consensus using 4/0 and 0/0 micro-benchmarks. When $f = 1$, the optimized PBFT requires 4.04 ms for a single counter service in the case of no faults, while EFS with and without tentative operation take 4.5089 ms and 7.4030 ms respectively,

To study the fault-scalability of EFS, we measure the time to resolve contention with an increasing number of tolerated faults. Table 3 lists the average of 100 measurements when the batch size is set 2. With a larger batch size, the average time for contending requests will be further reduced. Table 3 shows that EFS is efficient even when contentions occur frequently in a large-scale service.

6 Conclusion

We propose EFS that aims to provide BFT service with robust efficiency in the presence of faulty clients. EFS uses an efficient quorum-based BFT system when there are no contending requests, and switches to a fast agreement protocol to resolve contention. The two modes are integrated using a server-directed and lightweight switch which avoids the switch becoming the bottleneck. Known attacks from clients cannot harm the efficiency of EFS. Moreover, EFS has good fault-scalability in both the contention-free and contending cases which ensures that EFS has robust efficiency in the large-scale service.

Acknowledgments. Q. Cai, J. Lin and Q. Wang were partially supported by National 973 Program of China under award No. 2014CB 340603. F. Li was supported by NSF under Award No. EPS0903806 and matching support from the State of Kansas through the Kansas Board of Regents.

References

1. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 173–186 (1999)
2. Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., Wylie, J.: Fault-scalable Byzantine fault-tolerant services. In: 20th ACM Symposium on Operating Systems Principles (SOSP), pp. 59–74 (2005)

3. Cowling, J., Myers, D., Liskov, B., et al.: HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 177–190 (2006)
4. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. (TOCS) **27**(4), 7 (2009)
5. Hendricks, J., Sinnamohideen, S., Ganger, G., Reiter, M.: Zzyzx: scalable fault tolerance through Byzantine locking. In: 40th International Conference on Dependable Systems and Networks (DSN), pp. 363–372 (2010)
6. Guerraoui, R., Knezevic, N., et al.: The next 700 BFT protocols. In: 5th European Conference on Computer Systems (EuroSys), pp. 363–376 (2010)
7. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
8. Schneider, F.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
9. Clement, A., Wong, E., Alvisi, L., et al.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 153–168 (2009)
10. Adya, A., Bolosky, W., et al.: Farsite: federated, available and reliable storage for an incompletely trusted environment. In: 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 1–15 (2002)
11. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the OceanStore prototype. In: 2nd USENIX Conference on File and Storage Technologies (FAST), pp. 1–14 (2003)
12. Reiter, M.: Secure agreement protocols: reliable and atomic group multicast in rampart. In: 2nd ACM Conference on Computer and Communications Security (CCS), pp. 68–80 (1994)
13. Malkhi, D., Reiter, M.: A high-throughput secure reliable multicast protocol. In: 9th IEEE Computer Security Foundations Workshop, pp. 9–17 (1996)
14. Martin, J.-P., Alvisi, L.: Fast Byzantine consensus. IEEE Trans. Dependable Secure Comput. **3**, 202–215 (2006)
15. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
16. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
17. Gafni, E., Lamport, L.: Disk Paxos. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 330–344. Springer, Heidelberg (2000)
18. Kihlstrom, K., Moser, L., Melliar-Smith, P.: The SecureRing protocols for securing group communication. In: 31st Hawaii International Conference on System Sciences (HICSS), vol. 3, pp. 317–326 (1998)
19. Reiter, M.: A secure group membership protocol. IEEE Trans. Softw. Eng. **22**(1), 31–42 (1996)
20. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Byzantine replication under attack. In: 38th International Conference on Dependable Systems and Networks (DSN), pp. 197–206 (2008)
21. Kotla, R., Dahlin, M.: High throughput Byzantine fault tolerance. In: 34th International Conference on Dependable Systems and Networks (DSN), pp. 575–584 (2004)
22. Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution for Byzantine fault tolerant services. In: 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 253–267 (2003)