Improving Independence of Failures in BFT

Ali Shoker* , Jean-Paul Bahsoun* and Maysam Yabandeh†
*IRIT, Toulouse, France, Email: firstName.lastName@irit.fr
† QCRI, Qatar, Email: myabandeh@qf.org.qa

***Abstract*—Independence of failures is a basic assumption for the correctness of BFT protocols. In literature, this subject was addressed by providing N-version like abstractions. Though this can provide a good level of obfuscation against semantic-based attacks, if the replicas know each others identities then non-semantic attacks like DoS can still compromise all replicas together. In this paper, we address the obfuscation problem in a different way by keeping replicas unaware of each other. This makes it harder for attackers to sneak from one replica to another and reduces the impact of simultaneous attacks on all replicas. For this sake, we present a new obfuscated BFT protocol, called OBFT, where the replicas remain unaware of each other by exchanging their messages through the clients. Thus, OBFT assumes honest, but possibly crash-prone clients. We show that obfuscation in our context could not be achieved without this assumption, and we give possible applications where this assumption can be accepted. We evaluated our protocol on an *Emulab* cluster with a wide area topology. Our experiments show that the scalability and throughput of OBFT remain comparable to existing BFT protocols despite the obfuscation overhead.**

***Keywords*—Byzantine fault tolerance, obfuscated BFT, independence of failures.**

I. Introduction

Byzantine fault tolerance [1] (simply BFT) is a replication technique with the aim of tolerating arbitrary failures. State-machine based services [2] are deployed on replicas in *partially synchronous* systems [3]. At most a fraction, often one third, of the replicas are assumed to be faulty [4], [5].

Independence of failures is a major assumption for the correctness of BFT protocols. Existing protocols [5], [6], [7], [8], [9], [10] take this assumption for granted. In fact, if system replicas are identical then failures are likely to occur on all of them. This compromises the correctness of the system since any assumption about the maximum fraction of Byzantine replicas, e.g., one third, cannot be guaranteed. Many solutions were proposed to improve independence of failures of BFT protocols using N-version like obfuscation abstractions as in [11] and [12]. Indeed, these solutions can provide a good level of obfuscation through running different implementation versions on different replicas; however, non-semantic attacks like DoS could not be completely handled by these solutions. We argue that if system replicas are known to each other, then one replica can leak information about the others and, consequently, compromising one replica will likely compromise the others.

In this work, we address the obfuscation problem in a different way. We try to make replicas completely independent from each other through removing inter-replica interaction from the communication logic. Using obfuscation abstractions like [11] and [12] together with our work can lead higher independence of failures. Thus, we do not claim to replace existing solutions, but we do complete their work. In [13], the authors came to a conclusion that client-based BFT protocols ensure higher level of independence. On the contrary, most classical BFT protocols [5], [7], [8], [9], [10] rely on inter-replica communication to reach agreement. Even existing protocols known as client-based, like Q/U [6] and Quorum [9], are not completely client-based since under failures they require replicas to interact directly to resolve the conflict.

In this paper, we propose a new BFT protocol, called *OBFT* (Obfuscated BFT), that avoids any direct inter-replica communication. Instead, replicas interact with each other through a *trusted* third party which is the client in our case. Since clients play a crucial role in OBFT, we do assume they can not be malicious, though they can fail by crashing. In fact, to maintain obfuscation in our context, i.e., avoiding inter-replica communication, the client must not be malicious; otherwise, a malicious client can violate consistency. The proof is simple: the client can send two different requests to two distinct subsets of the replicas and behaves against each subset as if there was a single request, thus causing different object versions on different replicas to deviate.

We argue that this assumption makes sense for applications where the customers are trusted members of the same organization; e.g., airline ticketing services that provide access to different agencies. In airline ticketing system, the company hosts its service on independent replicas. It allows access for the ticketing agencies. The ticketing agencies access the airline service via their secured and trusted servers, which are viewed as trusted clients by the BFT airline service. Another problem arises when a trusted client gets attacked, since access information about the replicas will be exposed. We propose a possible solution to this issue using anonymous communication in Section V where replica IPs remain hidden.

OBFT requires $3f$+1 replicas in order to tolerate f Byzantine faults. A client in OBFT communicate with $2f$+1 *Active* replicas in its *Speculative* phase. The client sends a request to a primary replica that executes it, assigns it a sequence number, and sends it back to the client. The client forwards the request to the other replicas that reply again to the client after executing the request command. Then, the client accepts the operation if all responses match. Otherwise it launches a *Recovery* phase on the $3f$+1 replicas to exclude the *Suspicious* replicas (either faulty or slow); and then resumes to the Speculative phase acting on the new $2f$+1 Active replicas in the current view.

Despite the fact that clients are trusted, many challenges make OBFT non-trivial. First, clients can still crash. To ensure

obfuscation, OBFT must tolerate a crashed client (without inter-replica communication) otherwise unique request ordering among replicas can be compromised by the other clients. Second, upon failure detection, recovery is needed. In OBFT, faulty replicas will be replaced by correct ones (this might change the *primary* replica also). Thus, correct replicas should preserve a unique configuration; and the clients should be kept updated with these information too. Third, OBFT handles contending clients that can force versions of an object on different replicas to skew.

We experimented OBFT on Emulab [14] on Xeon machines using WAN settings where each machine deploys a Debian OS. On such system, OBFT scaled to hundreds of clients, and its peak throughput significantly exceeded that of state of the art client-based protocols Q/U, and Quorum [1]. Also, we conducted some experiments to compare OBFT with other *primary-based* protocols to convey the obfuscation cost by using OBFT; our results show that the obfuscation overhead is tolerable.

The rest of the paper is organized as follows.Some related works are recalled in Section II. Section III presents OBFT. After discussing the evaluation results in Section IV, we propose a possible improvement in V, and then we conclude the paper in Section VI.

II. Related Works

Independence of failures is an essential assumption for the correctness of the BFT protocols [5]. Most BFT protocols take this assumption for granted. Practical experience shows that faults are likely to occur on all replicas if they are identical. Several works addressed this subject through providing different running versions on different replicas. Castro et al., in *BASE* [11], proposed running different existing implementations on different replicas; and they introduced an abstraction layer to unify the differences among them. Roeder et al., in [12], proposed an obfuscation abstraction method to develop different versions of implementations that are semantically identical. This approach is more effective than BASE since new implementations are proposed during recovery, i.e., at run-time. Both approaches deal mostly with semantic failures, whereas, non-semantic ones like DoS could not be effectively resolved since replicas can expose information about each other. Our approach adds to these solutions the possibility to keep replicas unaware of each other.

In [15], the authors discussed how to make intrusion fault systems by using BFT technology; and tried to maintain some diversity in system components to leverage the levels of independence. The authors defined two concepts:

- Axis of diversity: a component of a system that may be diversified (e.g., OS, Storge, etc.).
- Degree of diversity: the number of choices available for a specific axis of diversity (e.g., Linux, Windows, etc).

Therefore, the authors propose to use different hardware, platforms, operating systems, libraries, etc. for different replicas, and they suggest using multiple physical facilities to avoid natural disasters and other localized physical threats. Our approach is different from [15] since we address the subject theoretically by avoiding inter-replica communication. In fact, our work and the above existing works are complementary.

Another study in [13], categorizes the different levels of failure independence on different setups in the cloud or WAN. The study shows that the protocols that are sensitive to the replica-client delay, and loss, do not perform well since the clients are typically connected via a WAN. If the latency between the replicas is higher, PBFT [5] offers the best performance. However, Q/U [6] performs the best when replicas are geographically distributed, because of the absence of communication between the replicas. The authors conclude that: in order to achieve the highest failure independence in WAN setting, inter-replica interaction should be avoided, and thus, they claim that the only available options are client-based protocols like: Q/U [6] and Quorum [9].

In fact, though client-based protocols like Q/U [6] and Quorum [9] have no primary replica, these protocols use inter-replica interaction to recover under failures. In essence, Q/U requires $5f + 1$ replicas to tolerate f Byzantine faults. Nevertheless, clients can only contact a *preferred quorum* (of size $4f$+1) for optimum performance. This could result in outdated histories in some replicas, which induce the cost of a *synchronization* phase to the protocol. In this phase, the outdated replica requests the up-to-date history from f+1 other replicas (to ensure that the history is not manipulated by some faulty replicas). Thus, although Q/U avoids direct inter-replica communication, it relies on replica interactions to *repair* from failures whenever the same copies on different replicas skew. Quorum [16] is another client-based BFT protocol built using the abortability approach [16]. Similar to Q/U, Quorum has the minimum latency among different protocols in contention-free networks. In the case of contention or Byzantine replicas, Quorum aborts and recovers to a *Backup* [16] similar to PBFT, thus using inter-replica communication again. Our proposed protocol is completely deprived from any inter-replica communication.

III. Design of OBFT

A. Model

Our system and fault models often comply with the traditional models (e.g., PBFT [5]). We assume a message-passing distributed system using a fully connected network of nodes, mainly a WAN, where nodes are: clients and servers. The network may (not infinitely) fail to deliver, corrupt, delay, or reorder messages. Faulty replicas may either behave arbitrarily, i.e., in a different way to their designed purposes, or they just crash (*benign* faults). A strong adversary coordinates faulty replicas to compromise the replicated service. However, we assume the adversary cannot break cryptographic techniques like: collision-resistant hashes, encryption keys, and signatures. Liveness, however, is guaranteed only whenever the system is *partially* synchronous [4], [5]; i.e., during intervals in which messages reach their correct destinations within some fixed worst case delay. Our fault model differs from existing ones by requiring that clients might fail by crashing but they do not behave maliciously (they are typically part of the same organization).

[1] We do not count other BFT protocols here since they are not client-based.

B. The Protocol

OBFT is a BFT protocol that avoids fault dependency among replicas and exhibits high performance in WANs. OBFT uses *Abortability* approach [16] to recover upon failures. OBFT requires $3f$+1 replicas to tolerate Byzantine replicas, where no more than f replicas can be Byzantine. However, using $2f+1$ replicas only at a time, it can sustain faults, but cannot ensure progress. Thus, OBFT launches the *speculative* phase on $2f+1$ *Active* replicas. Upon failure detection, it recovers by replacing the *Suspicious* replicas (i.e., either faulty or slow) with correct replicas from the f *Passive* ones; and then, resumes to the speculative phase on a new Active set (in a new *view*). The $2f+1$ replicas are enough to collect a correct *abort history*.

The algorithm of OBFT consists of two main phases: a *speculative* phase and a *recovery* phase. The messaging pattern of the speculative phase is depicted in Figure 1. In this section, we present the phases briefly (details are in later sections).

Speculative phase. The communication pattern of OBFT in a failure-free scenario is simple, and it is concerned with the Active set only ($2f+1$ replicas):

1) A client first sends its request to the *primary*.
2) The primary assigns a sequence number to the request, executes it, and sends a reply back to the client along with the assigned sequence number.
3) The client then sends the request together with the assigned order (previously done by the primary) to the remaining $2f$ replicas in the Active set.
4) Each non-primary replica executes the received requests by order, and returns the replies to the client.
5) The client commits the request only if all the responses of the Active replicas match; otherwise, it launches a recovery phase.

Recovery phase. This phase uses Passive and Active sets ($3f$+1 replicas).

1) Once the timer of a client expires waiting for $2f+1$ matching replies, the client sends a *Panic* message to the Active replicas.
2) Replicas, upon receiving the Panic messages, stop executing new requests and send an *Abort* message back to client with their signed *local histories*.
3) The client constructs an *Abort history* AH collected from the $f+1$ matching replies (more details later), and sends an $INIT$ request to all replicas ($3f+1$) along with AH.
4) The replicas execute the $INIT$ request (they append it to their local histories), and reply to the client with ACK_{init}.
5) As the client receives $2f+1$ matching ACK_{init} replies, it considers their corresponding replicas as Active, and the remaining f replicas as Passive (those are *Suspicious* replicas). The updated Active set becomes correct again (in the next *view*), and the speculative phase is resumed.

C. Algorithm Details

We describe here the protocol details. For simplicity, we assume no contention on replicas, i.e., a single client is accessing the service. Contention is addressed in later sections.

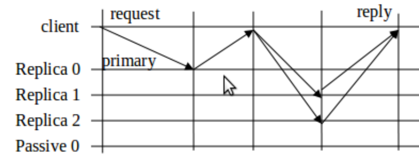

Fig. 1. Message exchange pattern of OBFT running on three Active replicas and one Passive.

Client role. To mitigate fault dependencies; OBFT clients enroll important tasks. First, the client issues the request towards the primary that assigns a unique sequence number. This is crucial to maintain consistency among different replicas. When the client receives the assigned reply from the primary, it validates its contents by verifying the Message Authentication Code (MAC). The client takes the grip again to resend the signed request to the other $2f$ Active replicas, however this time, accompanied with the sequence number. At that instant the client starts a timer, waiting for the replies.

The final decision is also taken by the client. Upon receiving $2f+1$ replies from the replicas before the timer expires; the client verifies their MACs and makes sure the replies are matching. If so, the client considers the request complete. Otherwise, the client launches a *recovery* phase by collecting an abort history, cleaning the Active set from *Suspicious* replicas, and switching to a new Active set in another view.

Ordering. Excluding the primary, all replicas validate the order of clients requests upon their receipt. They must verify requests with MACs and sequence numbers, and then execute them. The primary verifies the MACs only. Replicas discard a request r_{new} in case $o(r_{last}) > o(r_{new})$; where r_{new} and r_{last} are the assigned orders of the current request and that of the last executed request in the local history of the replica, respectively. Each replica executes the request r_{new} if it has already executed all requests r_j where $o(r_j) < o(r_{new})$. Otherwise, request r_{new} is en-queued in a buffer, waiting for the missing requests that fill the gap. Final replies are authenticated via MACs and are sent by all replicas directly to the client.

D. Recovery Phase

The *Recovery* phase is composed of three major steps: aborting, collecting abort history, and cleaning *Active* set from any *Suspicious* replicas.

Aborting. The client in OBFT considers a request as complete if the received $2f+1$ responses of the Active replicas are matching, before the expiry of the timer. Otherwise, the client stops sending requests and sends a *Panic* message to all Active replicas. Each replica, upon receiving the Panic message, stops receiving/executing requests, appends its *local history* of committed requests to an *Abort* message, and sends it to the client. The latter waits until it receives a sufficient number of signed Abort messages, i.e., the first $f+1$ non-conflicting ones. The intuition is that it is necessary and sufficient for the number of received correct commit histories to exceed the number of faulty ones (since no more f replicas can be

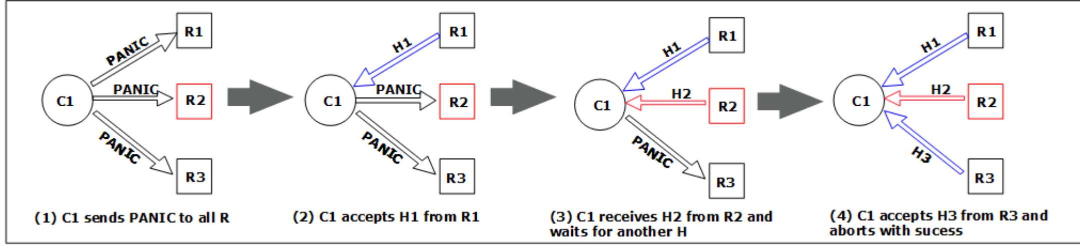

Fig. 2. An example of OBFT while aborting, where $f = 1$.

Byzantine); knowing that faulty replicas might not respond at all. Aborting is achieved as follows: the client waits for the first f+1 local commit histories to be received. If no conflicting entries among the f+1 received local histories are found by the client, it stops receiving new histories, and collects the $f+1$ messages in a $Proof_{AH}$ set, that is used to form the abort history AH later; otherwise, the client has to wait for further matching replies. Figure 2 presents an example where the first two histories returned from replicas R_1 and R_2 are conflicting, consequently the client has to wait for the history from replica R_3. Thus, the phase continues till f+1 non-conflicting local commit histories are received by the client.

Building abort history. A correct abort history is crucial for safety. It preserves total ordering and consistency across different switching phases (i.e., views). The abort history is collected from the current Active set, to initialize the local histories of the replicas on a new correct Active set. Building the abort history AH is done by the client after the receipt of $f+1$ non-conflicting signed abort messages from different replicas, collected in the $Proof_{AH}$ structure (as revealed before). The steps can be summarized as follows: the client generates a history h such that: $h[j]$ equals the value that appears at position $j \geq 1$ of $f+1$ different local histories (LH_j), that appear in $Proof_{AH}$. If such a value does not exist for some position x, then x is the last index of h. Finally, AH is the longest prefix of h in which no request appears twice (i.e., exclude duplicate entries). The resulting abort history AH thus includes all the globally committed client requests as well as some partially committed ones in the previous view; for example, if the request is received by at least f+1 replicas but not all of them (however this does not harm correctness). AH is used then to initialize the local histories of the new Active replicas (see more details in later sections).

Eliminating faulty replicas. The client in OBFT attempts to replace the faulty replicas in the Active set with correct ones from the Passive set; this is done by detecting the Suspicious replicas (i.e., faulty and slow ones) among the $3f+1$ replicas. This occurs as follows: after collecting the abort history AH, the client sends an $INIT$ request (with AH appended) to the $3f+1$ replicas. The replicas execute the INIT request, and append AH to their local history (if it is not already done through a previous view). Then, they reply to the client with ACK_{init}. Then, the client verifies the received ACK_{init} messages, and waits until it receives $2f+1$ matching ones. The corresponding replicas to the first $2f+1$ matching messages constitute the new Active set, and the remaining f form the Passive set. This process ensures eliminating the *Suspicious* replicas from the Active set, and having a correct one in the next view.

E. Switching

Switching requires $3f+1$ physical replicas where only $2f+1$ of them are used at once as Active replicas. The other Passive replicas, however, are used to replace the Active replicas whenever failures occur. This requires, sometimes, changing the primary replica also.

Initializing control information on clients. Upon launching the system, default Active set and primary replica are chosen. And since these are changing among the same physical replicas across different views, it is crucial for the client to know this *control information* before issuing its requests. The control information is collected on replicas and clients in the $CONTROL$ structure, that is composed of: (1) The Active set $ACTIVE$, (2) the primary P, and (3) the current view number $VIEW_{curr}$. Hence, some control messages are needed to deliver this information to any new coming client. The client gets informed as follows: it sends a GET_{info} message to all $(3f+1)$ replicas. $Active$ replicas send an $INFO$ message to the client containing the control information: (1) The Active set $ACTIVE$, (2) the primary P, and (3) the current view number $VIEW_{curr}$. Then, the client waits until it receives f+1 matching $INFO$ messages from the replicas. Once done, it saves the information of the $INFO$ messages, updates its CONTROL, and starts sending its requests according to the protocol. Otherwise, it starts again the process.

Control information after recovery. However, when recovery occurs, and some client c succeeds in aborting, sending the $INIT$ request, and identifying the Suspicious replicas, the control structure CONTROL should change to exclude faulty replicas. Thus, the replicas refuse to execute any request until: a SET_{info} control message (see later) is received from c, or the timer expires. Instead, the replicas respond on any request type with an $INFO$ message (possibly with some empty fields). In the case where the timer of a replica expires, it releases the lock, and accepts PANIC messages in the current view to allow other clients to perform a successful recovery [2]. SET_{info} is important after recovery by which replicas get assigned a new CONTROL. To fill the SET_{info} message, the

[2]Recall that clients in OBFT can not be Byzantine, but they may crash.

client determines the Active set, selects the first replica to be the primary, and increments the view number.

If CONTROL on the replicas is already updated by the SET_{info} message, the INFO reply message will be complete and informative enough for the clients (upon receiving $f+1$ matching such messages). Otherwise, if the replicas are still waiting the SET_{info} message from c, the ACTIVE and the primary replica P fields in the INFO message will be empty, in contrast to the $VIEW_{curr}$ field that allows the clients to retry sending Panic messages in the current view successfully; since replicas do not accept any requests from other views. OBFT handles SET_{info} messages according to the usual communication pattern in Figure 1 (the receiving replica knows from the SET_{info} message itself, whether it is included in the Active set, it is a new primary, or it is a Passive replica).

F. Handling Contention

Views perspective. Requests of any type are required to comprise the current view number $VIEW_{curr}$. In addition, any request received by the replicas is validated by verifying its MACs and the $VIEW_{curr}$ field, before being executed. Any request that belongs to a different view, gets rejected.

Panicking. The speculative phase in OBFT is deprived from contention problems as long as all requests that belong to the current view are ordered by the primary. However, upon failure, the clients launch the recovery phase. The recovery phase allows any client to panic, and to collect the abort history. Clients keep retransmitting $PANIC$ messages until they receive an enough number (i.e., $f+1$) of matching local histories, or a filled INFO messages from the replicas.

Initializing. When any client (one or more) creates the abort history, it sends $INIT$ message to all the replicas ($3f+1$) to initialize their local histories and to update their view number $VIEW_{curr}$. Under contention, different replicas might receive different $INIT$ requests from different clients, and hence none will be completed since no client will be able to collect enough ACK_{init} from $2f+1$ replicas. Thus the clients follow an exponential back-off scheme that offers more chance that all replicas execute the same $INIT$ requests sent by some client c. This ends up by replicas, having consistent local histories in the new view.

Control information. Afterwards, the replicas will be waiting for the SET_{info} request from the same client c. During this period, and to maintain non-conflicting messages from contending clients, any request will be discarded by the replicas, that reply instead, by an $INFO$ message (where its Active set and primary replica fields are still empty) containing the $VIEW_{curr}$ that is needed by the clients while retrying their future request attempts. The step ends with a unique Active set and a primary replica across different views once the replicas receive the SET_{info} message from the client c. After this stage, any request from the clients will be handled as designed if it belongs to this view, otherwise, replicas respond with an $INFO$ message (where no fields are empty this time) to update those clients with the new control information. Then, the clients can send requests as usual in the new view.

IV. EVALUATION

In this section, we evaluate OBFT experimentally, and we provide some analytical evaluation. Our strategy is to compare our protocol with existing protocols showing that their scalability are comparable in WANs. Then we compare OBFT with client-based protocols that provide obfuscation in failure-free cases, being the main objective of the paper. In addition, we discuss the switching overhead upon failures.

Our experiments are performed on 43 64-bit Xeon machines with 2 GB of memory employed on Emulab [14] cluster. No virtualization is used, thus simulating WAN environment on real machines. Each replica runs on a separate machine, and the clients are scattered over 40 machines (at least 3 client processes per machine). All machines are connected via a star topology. The end-to-end (E2E) delay is set to 20 ms which is a typical WAN speed. For each setting, we have run four *a/b benchmark*[3] (same benchmark used in PBFT [5]) experiments using different payload sizes: 0/0, 0/1, 1/0, and 1/1. Without a payload, the size of the request and the reply messages are less than 100 bytes. The fault factor, f, is equal to one. Multicast in PBFT, Zyzzyva, and Quorum is disabled (since they are deployed on WAN). Q/U experiments include a single difference where we needed six replicas ($5f$+1; for $f=1$) instead of $3f+1$, as this is the number required by Q/U [6] to operate.

A. Analytical Evaluation

Before proceeding with the experimental evaluation, we provide an analytical comparison for OBFT with well known state of the art BFT protocols. The comparison is summarized in Table I. The results convey the interesting characteristics of OBFT as it achieves the lowest cost among BFT protocols in most cases (the bold entries in Table I). In the table, we consider most protocols optimizations (as in [8]), however, we exclude batching. Row A in Table I shows that the number of replicas needed by OBFT to tolerate f Byzantine faults is minimal among the protocols. Similarly, the cost is minimal for OBFT in the speculative phase, since it communicates with only $2f+1$ replicas out of $3f+1$ (row B in Table I). Moreover, Row C shows the number of MAC operations performed by the CPU of the bottleneck replica. The table points out that OBFT again has the minimal value 2, since any replica verifies the request only once, and authenticates the reply once. Latency in OBFT (row D in Table I) is not optimal as compared to other protocols; in fact, OBFT sacrifices this latency to maintain obfuscation. Finally, the number of $send$ (or $sendto$) calls to the system kernel is minimal on the bottleneck replica in OBFT. This is clear since every replica in the protocol sends the reply only once (see Figure 1).

B. Obfuscation Cost

To leverage the fault tolerance of BFT protocols by mainlining obfuscation, it is worthy to pay some additional costs. The above analysis shows that the latency of OBFT is higher

[3]In a/b benchmarks, *a* and *b* correspond to request size, and response size in KB, respectively.

	PBFT	Zyzzyva	Q/U	Quorum	OBFT
A	**3f+1**	**3f+1**	5f+1	**3f+1**	**3f+1**
B	**2f+1**	**2f+1**	5f+1	3f+1	**2f+1**
C	2+8f	2+3f	2+4f	**2**	**2**
D	4	3	**2**	**2**	4
E	3	2	**1**	**1**	**1**

TABLE I
Analytic evaluation for the state of the art BFT protocols tolerating f faults, using MACs for authentication, and assuming preferred optimization (without batching): A represents the number of replicas needed to tolerate f Byzantine replicas; B is similar to A but excluding witness and backup replicas. C represents the number of MAC operations on the bottleneck replica. D is the number of one-way latencies needed for each request. E represents the number of send/to kernel calls on the bottleneck replica. Bold entries denote protocols with the lowest known cost.

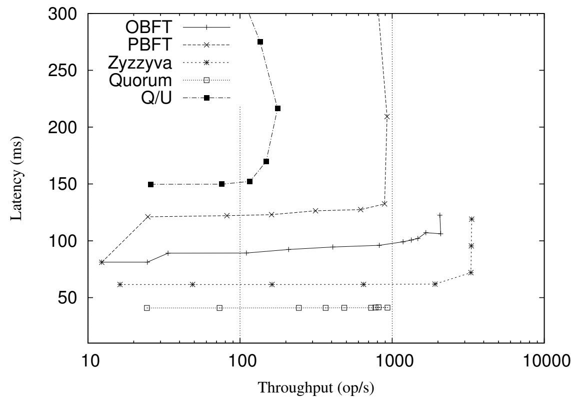

Fig. 3. Scalability of BFT protocols for 1/0 benchmark in WAN.

(Zyzzyva, Q/U, and Quorum) or equal (PBFT) to state of the art BFT protocols. Consequently, this impacts the throughput of OBFT also. Despite this, OBFT maintains a good scalability.

Figure 3 shows the scalability using the 1/0 benchmark where end-to-end latency (E2E) is 20ms (we do not mention other payloads as they are very similar). We use logarithmic scale so that PBFT and Q/U can be observed. As the figure shows, Zyzzyva achieves the best scalability. It scales up to 320 clients reaching a peak throughput of 3332 op/s, whereas, OBFT scales to 280 clients with a peak throughput equals to 2193 op/s. This is reasonable because of the simple three-delay message pattern of Zyzzyva. PBFT also, scales to 120 clients, however, its throughput is almost half that of OBFT (980 op/s); perhaps this refers to the extensive load of messaging on the replicas that PBFT messaging pattern imposes. Q/U scales only to 30 clients, scoring a throughput of 537 op/s. Beyond this point, the protocol suffers from the recovery phases due to contention. Quorum scales to 60 clients only since it is designed to work on LANs using multi-cast, which we disable in our WAN setting.

C. Comparison of Client-based Protocols

Since this paper is concerned mainly with obfuscation, we exclude PBFT and Zyzzyva in later sections, and we stick with Q/U and Quorum being client-based and can maintain

Protocol	Q/U	Quorum	OBFT
Message delays	2	2	4
Switching Message delays	-	4	4
Latency for E2E=20ms	41ms	40ms	80ms

TABLE II
Micro-benchmark latencies on WANs when E2E=20ms.

obfuscation in failure-free cases. OBFT acquires a good performance through various characteristics: (1) it needs only $3f$+1 replicas to tolerate f arbitrary faults (though speculative case communication is done on $2f+1$ Active replicas only at a time). (2) It relies on the clients to multi-cast request and not on replicas. (3) It pushes the cryptographic operations towards the client.

On the other hand, Q/U [6] requires at least $5f$+1 replicas to tolerate f Byzantine faults. Additional fees shall be paid with larger f. Despite the use of preferred quorums (of size $4f$+1), Q/U achieves lower throughput than OBFT. This becomes lucid when the number of clients increases; partly because Q/U does not use a primary to order requests as OBFT does. Quorum [9] also shares some aspects with Q/U; mostly since it is client-based and involves only two communication phases. However, Quorum also suffers from interference under contention; this makes it hard to deploy on reliable contended services. In addition, Quorum is designed to multi-cast requests to the replicas. Multi-cast in not yet supported on WANs, that is why we disable this feature (in fact, this causes some drop in performance). Quorum operates only in free-failure environment, and needs a recovery phase upon failure that might violate obfuscation.

1) Recovery Cost: Similar to Quorum, OBFT is inherently speculative and perform well only in best cases, i.e., when there are no faults. Under failures the protocol should abort to another Active set, and this imposes additional costs represented by *switching delays*. Switching delays are proportional to the end-to-end latency (E2E) of the WAN (assuming the execution time of the operation on the CPU is negligible as compared to E2E). Table II (second row) conveys the number of message latencies needed to switch. The table shows that Quorum and OBFT need, in best case, four message delays to switch (i.e., the sum of $PANIC$, $ABORT$, $INIT$, and ACK_{init} latencies). The cost of switching can then be approximated by $4{*}E2E$. We do not measure recovery cost for Q/U, since we assume using no preferred quorums (to ensure obfuscation).

2) Micro-benchmark: We present here the results on a/b benchmark when only one client is accessing the replicated service. Table II displays the latency results (using 0/0 benchmark) for OBFT, Quorum, and Q/U by setting the E2E to 20 ms. Since in WANs the operation execution time and MAC handling time are almost negligible as compared to the E2E latency, when the E2E latency is set to 20 ms, OBFT achieves a latency of 80 ms. Q/U on the other hand reaches half this latency as depicted in Table II. We roughly relate this difference to the number of communication round-trips needed to complete an operation (as shown in the same table). In fact, OBFT needs a couple of round-trip messages; one

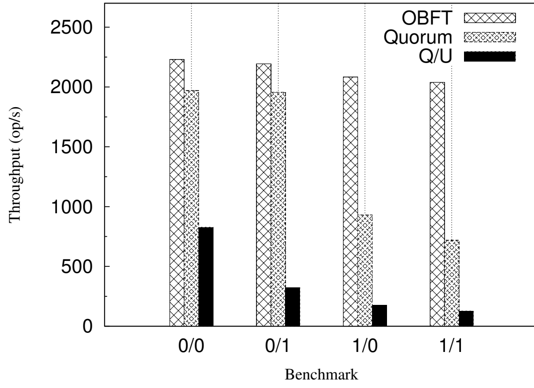

Fig. 4. Peak throughput for WAN setting with E2E=20ms.

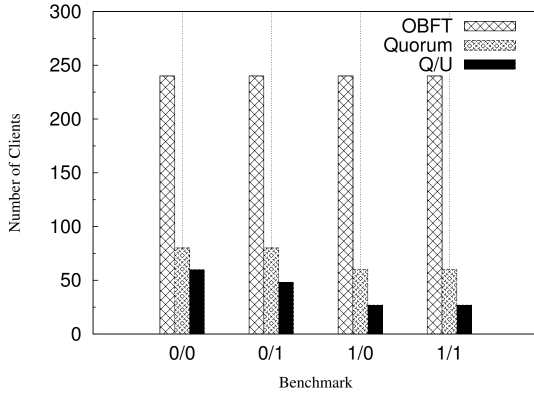

Fig. 5. Scalability for WAN setting where E2E=20ms.

message is sent to the primary to establish request ordering, and another is sent to communicate with other replicas. Q/U, however, achieves this latency since it completes the operation in a single round-trip instead of two. Again, since Quorum (like Q/U) needs only two one-way communication phases to commit a request in a speculative way, they share same performance in a contention-free environment, the results are shown clearly in Table II.

3) Peak Throughput: To experiment the peak throughput, we ran up to 300 concurrent clients on WAN setting with E2E=20ms. As depicted in Figure 4, our protocol achieves a peak throughput of 2230 op/s for the 0/0 benchmark. The peak throughput of the other benchmarks (0/1, 1/0, and 1/1) are close as shown on the figure. On the other hand, Q/U could not exceed 828 op/s throughput on the 0/0 benchmark (Figure 4). This peak throughput drops further as the request size gets larger. The benchmarks 1/0 and 1/1 of Q/U score no more 176 op/s and 127 op/s, respectively. These results are expected since Q/U is not resilient to a high number of clients, and this forces the protocol to load excessive *Repair* and *Sync* phases, and the client *backoff* scheme. On the contrary to OBFT, that relies on the primary to order requests, and thus, avoids request collisions while accessing replicas, and pushes multi-cast and encryption overhead towards clients. Quorum also gets affected

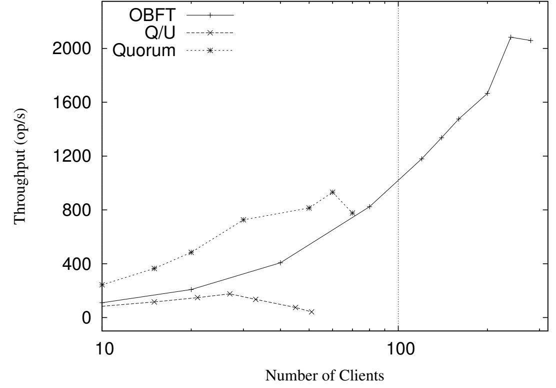

Fig. 6. Throughput of BFT protocols using 1/0 benchmark in WAN.

by the request size more than OBFT. Figure 4 shows that the peak throughput of Quorum drops to a ratio of 1/2 whenever requests of 1KB are used (1/0 and 1/1 benchmarks). We refer this to the load imposed on clients due to use of uni-cast instead of multi-cast. Despite this, in 0/0 benchmark, Quorum achieves a peak throughput close to that of OBFT; i.e., 1970 op/s. This refers to the simple message pattern of Quorum, since it avoids the recovery phases needed by Q/U.

4) Scalability: As for scalability, OBFT dominates Q/U and Quorum (Figure 5). In the experiments, the results of Q/U started to fluctuate for more than 30 clients. The protocol ceased to work for a number of clients greater than 30 or 80, depending on the experiment. Again, larger request sizes have a significant impact on the scalability of Q/U as the benchmarks 1/0 and 1/1 in Figure 5 convey. By observing Figure 5, we notice that Quorum also could not scale for more than 80 clients for the 0/0 benchmark, and 60 clients whenever larger requests are used. In fact, we do not expect more from Quorum, since it is designed to work on LAN and uses multi-cast that we disable in our WAN setting. However, OBFT experiments finished successfully up to 240 simultaneous clients in all benchmarks (Figure 5). OBFT can handle this high number of clients since it avoids collisions among requests by having a primary replica to assign sequence numbers, and to distribute the load of multi-cast among the clients to avoid replica bottlenecks.

5) Throughput: Figure 6 conveys the results of 1/0 benchmark when E2E=20ms (we use a logarithmic scale to better observe Q/U results). We remove the 0/0 benchmark since results are similar to 1/0. The x and y axes represent the number of clients and the throughput, respectively. The number of clients vary between 0 and 300 (starting with 3 client threads per machine). As depicted in the figure, the increase in the number of clients raises up the throughput of OBFT to 2230 op/s (for 240 clients). Then, as the number of clients increases the throughput starts to degrade gradually. However the throughput of Quorum and Q/U almost drops by a ratio of 1/2. A possible explanation to this behavior is that the load on the clients in Q/U and Quorum is large as compared to OBFT. In fact, a client should send/receive $4f+1$ messages in

the case of Quorum and $5f+1$ messages in Q/U, which makes the client a network bottleneck.

V. Improvements

Though avoiding inter-replica communication can keep replicas unaware of each others, pushing this towards the clients can still leak information about the replicas. To resolve this issue, we propose that the identities of replicas remain anonymous even for the clients. This is possible if we required the trusted clients (that usually belong to the same organization) to contact the servers using anonymous communication techniques like [17], [18], [19]. Notice that we can require from these trusted clients to use anonymous communication since they are finite (e.g., few thousands) as they belong to the same organization, however, without using our protocol it is not possible to require from unknown clients to use anonymous communication. Using this method, neither clients nor replicas can expose any information about the identities of the set of replicas used. This method can be used only if sacrificing some performance is acceptable since anonymous communication usually impose significant overhead.

VI. Conclusion

This paper improves the independence of failures in BFT protocols. We introduced a new obfuscated BFT protocol, called OBFT, where replicas remain completely unaware of each other. This improves tolerance to non-semantic attacks like DoS since a compromised replica could not leak information about other replicas. The design of OBFT allows to deploy BFT services on WANs, and especially clouds, to benefit from the versatility of the hardware, software, platforms, etc. In addition, OBFT can exploit the geographical distribution of the clouds around the globe to avoid natural disasters, and regional power failures, etc. Furthermore, our protocol can be used along with obfuscation abstractions introduced in [11] and [12] to improve independence if failures.

Moreover, OBFT achieves a good performance as compared to state of the art client-based protocols (that can maintain obfuscation). Our experimental results show that OBFT scales to hundreds of clients in a WAN, while the throughput of state of the art BFT protocols quickly drops as the number of clients increases. One limitation of our work is that it should not be used unless if clients can be trusted. Possible applications are when customers are trusted members of the same organization; like airline systems with many agencies. This is needed since clients in OBFT are assumed not to be malicious, but they can crash.

References

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

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] C. Dwork and N. Lynch, "Stockmeyer l: Consensus in the presence of partial synchrony," *Journal of The ACM*, 1988.

[4] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

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

[6] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, 2005.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: a hybrid quorum protocol for byzantine fault tolerance," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 177–190.

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 45–58, 2007.

[9] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 363–376.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–168.

[11] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 236–269, Aug. 2003. [Online]. Available: http://doi.acm.org/10.1145/859716.859718

[12] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 4:1–4:54, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1813654.1813655

[13] R. Guerraoui and M. Yabandeh, "Independent faults in the cloud," in *LADIS '10: Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2010, pp. 12–17.

[14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 255–270, 2002.

[15] Obelheiro Rafael R., Bessani Alysson Neves, Lung Lau Cheuk, Correia Miguel, "How practical are intrusion-tolerant distributed systems?" Department of Informatics, University of Lisbon, Tech. Rep. LPD-REPORT-2010-10, 2006. [Online]. Available: http://hdl.handle.net/10455/2992

[16] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocol," EPFL, Tech. Rep. LPD-REPORT-2008-008, 2008.

[17] N. M. Roger Dingledine and P. Syverson, "Tor: The second-generation onion router," in *In Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 303–320.

[18] S. Goel, M. Robson, M. Polte, and E. G. Sirer, "Herbivore: A Scalable and Efficient Protocol for Anonymous Communication," Cornell University, Ithaca, NY, Tech. Rep., 2003.

[19] H. Corrigan-Gibbs and B. Ford, "Dissent: accountable anonymous group messaging," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 340–350.