HAL open science

BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols

Divya Gupta, Lucas Perronne, Sara Bouchenak

▶ To cite this version:

Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols. 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2016, Heraklion, Crete, Greece. pp.115-128, 10.1007/978-3-319-39577-7_10 . hal-01372682

HAL Id: hal-01372682

https://hal.science/hal-01372682v1

Submitted on 13 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols

Divya Gupta*, Lucas Perronne*, and Sara Bouchenak†

*Univ. Grenoble Alpes, LIG Grenoble, France
†Univ. Lyon, INSA Lyon, LIRIS, Lyon, France
Divya.Gupta@imag.fr,Lucas.Perronne@imag.fr,Sara.Bouchenak@insa-lyon.fr

Abstract. Byzantine Fault Tolerance (BFT) is an interesting means to make computing systems resilient in presence of failures and attacks. That being said, designing and implementing BFT protocols is a hard and tedious task. This first comes from the inherent complexity of designing BFT distributed protocols, reasoning about their correctness, and implementing the software prototype of the protocols in a consistent and efficient way. Another reason that makes BFT protocols hard and error prone is the lack of tools for testing and evaluating protocols implementations in various and realistic settings. Furthermore, BFT protocols differ in many aspects, ranging from the faulty behaviors they handle, to the communication patterns and cryptographic mechanisms they apply. Thus, a comprehensive benchmarking environment is still missing to easily analyze and compare the effectiveness and performance of these protocols. In this paper, we present *BFT-Bench*, the first benchmarking framework for evaluating and comparing BFT protocols in practice. *BFT-Bench* includes different BFT protocols implementations, their automatic deployment in a distributed setting, the ability to define and inject different faulty behaviors and workloads, and the online monitoring and reporting of performance and dependability measures. The experimental results of the evaluation of *BFT-Bench* show the effectiveness of the framework, easily allowing an empirical comparison of different BFT protocols, in various workload and fault scenarios.

Keywords: Fault Tolerance; Byzantine Faults; Fault Injection; Performance; Robustness; Benchmarking

1 Introduction

Cloud computing environments are now increasingly common. With their expansion, unpredictable events such malicious attacks, network delays, data corruption, and other types of Byzantine faults require specific fault tolerance mechanisms. Byzantine Fault Tolerance (BFT), based on state machine replication, consists in replicating the critical service in several replicas running on different nodes, and thus, ensuring service availability despite failure occurrence [13]. When clients access the service, this is done through a specific BFT communication protocol that ensures that client requests are processed by replicas in the same order.

There has been a large amount of work on Byzantine Fault Tolerance (BFT) protocols. Early efforts have explored the practicality of Byzantine Fault Tolerance, with PBFT protocol[6]. Other efforts have been made to improve the performance of the protocols and reduce the cost they induce due to many message rounds and cryptographic operations. Thus, some BFT protocols focus on improving performance in fault-free cases [15, 9, 2], while other protocols improve performance in presence of failures, each one proposing and applying techniques to counter specific types of faults such as network contention, system overload, etc. [3, 7].

However, there has been very little in the way of empirical evaluation of BFT protocols. Evaluations of the protocols have often been conducted in an ad-hoc way, which makes them difficult to reproduce, and compare with new protocols. Moreover, it is generally admitted that BFT protocols are too complex to implement, thus, re-implementing them each time a new protocol must be compared with existing ones is not realistic.

In this paper, we present *BFT-Bench*, a benchmarking environment for evaluating performance and robustness of Byzantine fault tolerance systems. *BFT-Bench* enables the definition of various execution scenarios and faultloads, their automatic deployment in an online system, and the production of various monitoring statistics. This provides a means to analyze and compare the effectiveness of the protocols in various situations. *BFT-Bench* is an open framework that includes state-of-the-art BFT protocols, and may be extended with new BFT protocols. In addition, the paper presents an evaluation with *BFT-Bench*, empirically comparing different BFT protocols, and exhibiting their level of performance and robustness in different scenarios.

The remainder of the paper is structured as follows. Section 2 discusses the related work. Sections 3 presents *BFT-Bench*. Section 4 describes the experimental evaluation, and Section 5 concludes the paper.

2 Related Work

A Byzantine fault tolerant system is able to counter arbitrary faults, ranging from hardware crash, to message corruption, network congestion, or any other misbehavior. In the following, we review the related work on Byzantine fault tolerance, and BFT benchmarking.

BFT From Theory to Practice. BFT State Machine Replication (SMR) consists in replicating the underlying service in several replicas, to ensure service availability and correctness despite fault occurrence [13]. Such a service handles requests coming from concurrent clients. Thus, to ensure consistency among service replicas, an agreement protocol is applied to guarantee that client requests are executed in the same order by correct service replicas. Reaching an agreement requires $3f + 1$ replicas to handle upto f arbitrary faults [11].

BFT Performance Improvement in Fault-Free Conditions. One of the main drawbacks of BFT was its cost. Thus, everal protocols were proposed to

enhance the performance of BFT protocols while maintaining their correctness. A first family of BFT protocols aims at improving the performance of the protocols in the absence of faults. They usually run a lightweight version of the protocol in fault-free cases, and switch to a more robust version of the protocol at fault occurrence. This is interesting in scenarios where faults occur rarely, and where it is more interesting to provide priority to fault-free cases. Examples of such protocols are Zyzzyva [9], Chain [15], and Aliph [2], allowing to improve clients request throughput/latency.

BFT Performance Improvement in Presence of Faults. Another family of BFT protocols intends to improve performance in presence of faults. Roughly speaking, these protocols provide practical and efficient mechanisms to specifically handle some misbehaviors (i.e., fault types). Aardvark [7], Prime [1], Spinning [16], and RBFT [3] are examples of such protocols.

BFT Simulation and Benchmarking. General performance benchmarks have been proposed to evaluate the performance of application servers, web servers, data management systems, etc. Other solutions consider benchmarking dependability to provide a means to characterize system behavior in presence of faults. They consider different underlying systems such as MapReduce [12], or web servers [8]. Less effort has been done for benchmarking BFT systems. BFT-SMaRt is a replication engine that implements a BFT protocol; it interestingly includes a tool for evaluating the BFT protocol [4]. However, it is limited to the assessment of that particular protocol. Simulators of BFT protocols were also proposed [14, 10]; in contrast, in this paper we consider the empirical evaluation of BFT. Thus, there is a need for a comprehensive benchmarking environment to help researchers and practitioners to conduct empirical studies and better analyze and evaluate the performance and robustness of BFT protocols.

3 *BFT-Bench* Framework

BFT-Bench framework allows empirical evaluation and comparison of state-of-the-art and new Byzantine fault-tolerance systems. Figure 1 describes the major components of *BFT-Bench*: (i) several BFT protocols implementations, (ii) fault scenarios to be injected in the underlying BFT system, (iii) load to be injected in the running underlying system, and (iv) monitoring statistics to report performance and dependability statistics the system.

Thus, *BFT-Bench* enables automatic deployment of the experiments in a distributed system that consists of several nodes running the replicas of the BFT protocol, and one or multiple nodes emulating clients sending concurrent requests to the BFT system.

3.1 BFT Protocols

BFT-Bench is intended to be an open framework that can be extended with new BFT protocols to evaluate, new fault models. In this paper, the following state-of-the-art BFT protocols are considered: PBFT for being the first practical BFT

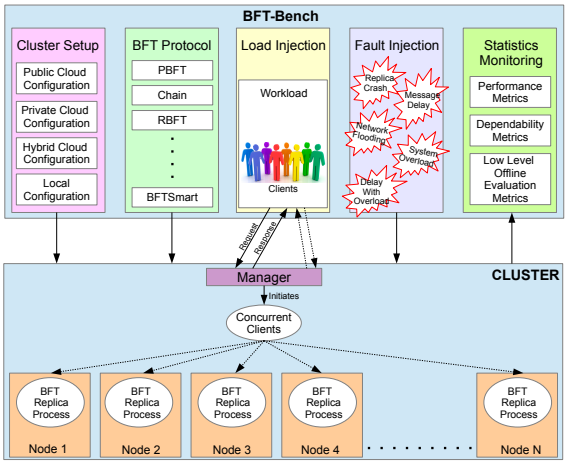


Fig. 1. Overview of *BFT-Bench*

protocol [6]; Zyzzyva, Chain, and Aliph for their performance efficiency in fault-free conditions [9, 15, 2]; Aardvark, and RBFT as instances of robust protocols that improve performance in presence of failures [7, 3]. These protocols were chosen for their variety of features, their variety of communication patterns as described in Figure 2, and their variety in terms of fault types the protocols prototypes actually handle (see Section 3.2).

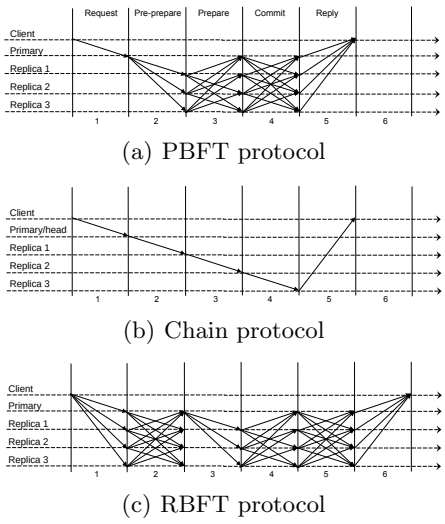
Practical BFT Protocol. PBFT's communication pattern is used as a baseline many other protocols such as Aardvark and RBFT [7, 3]. In PBFT, upon a client request the primary sends pre-prepare messages to other replicas with assigned sequence number to the request. Then, prepare messages and commit messages are exchanged to agree on the sequence number. If PBFT suspects the primary to be malicious, it undergoes a *view change* to replace the primary by another replica.

Protocols Enhancing Performance in Fault-Free Conditions. Zyzzyva is a speculative, high throughput BFT protocol [9]. Its design is meant to bypass the expensive agreement steps of PBFT in fault-free settings. In such scenario, the clients send their requests to the primary in charge of assigning sequence numbers. The primary then forwards the ordered requests to the other replicas, which speculatively execute these requests and send the responses to the clients. If a client receives $3f + 1$ consistent matching responses, it commits. Otherwise, clients apply additional steps such as collecting commit certificates and creating proofs of misbehaviors to trigger view change.

Chain protocol, as its name suggests and as described in Figure 2(b), follows a chain-like communication pattern where clients send requests to the head replica, which itself sends messages to its successor replica, and so on [15]. Chain greatly

benefits from batch optimization where multiple messages are sent in one batch, which improves system throughput, with a peak of performance when the system is completely saturated (i.e., when the network link between any two servers is fully loaded). However, Chain by itself is unable to ensure Byzantine fault tolerance, and must rely on a protocol switching mechanism when subject to failures.

Aliph protocol involves several sub-protocols [2]. Its initial configuration, Quorum, is dedicated to provide high performance if the system does not involve asynchrony, contention, or failures. When facing contention, Quorum is replaced by Chain. Finally, upon occurrence of Byzantine behaviors, Chain is replaced by a backup protocol that handles Byzantine faults, for example PBFT. In Quorum, clients directly send requests to all replicas. These replicas independently execute the requests, updates their local history and reply to the clients. Note that the ordering phase commonly performed by the primary replica is skipped in Quorum, thus providing a better response time. Thus, in Aliph the client side of the protocol is responsible of managing inconsistencies, and relies on a panicking mechanism to trigger sub-protocol switching.



(a) PBFT protocol

(b) Chain protocol

(c) RBFT protocol

Fig. 2. Examples of communication patterns of BFT protocols

Protocols Enhancing Performance in Presence of Faults. Aardvark prototype implements efficient fault tolerance mechanisms for faults such as intentional message delay, network flooding, or clients sending corrupted requests to the system [7]. To handle these fault types, Aardvark uses mechanisms such as replica blacklisting or digital signatures, to minimize the impact of faulty components on the overall system performance.

RBFT strengthens the architecture of PBFT and incorporates adaptive mechanisms to deal with different faulty behaviors [3]. RBFT runs $f + 1$ multiple instances of the same BFT protocol in parallel but the requests are executed only by one of the instances called master instance, while other f instances are called backup instances. Each backup instance has its own primary which orders the incoming requests in order to monitor the difference of throughput between the master instance and itself. If the performance at backup and master instance differs by a given threshold at not less than $2f + 1$ replicas, the primary replica at master instance is considered faulty and a *view change* is triggered, where a new primary is elected at every instance.

3.2 Fault Injection

In the following, we first describe the fault types that are handed by state-of-the-art BFT protocols presented in Section 3.1, and how *BFT-Bench* injects them in a running system. We then present how to describe a *faultload*, i.e. fault scenario to be injected by *BFT-Bench*.

Fault Types Examples

Replica Crash. Upon a replica crash, the replica stops and does not participate in any further communication with the clients or the other replicas of the BFT protocol. In practice, *BFT-Bench* remotely connects to the target replica node and kills the replica process. Note that the implementation of this fault type injection is BFT protocol-independent, thus, it does not require changes to BFT protocols prototypes.

Message Delay. When a replica starts delaying messages, it slows down all future operations of the protocol depending on these messages, thus, leading to degradation in performance. As a result, this Byzantine behavior is especially critical when it occurs at the primary replica. In practice, BFT protocols prototypes are extended to integrate the injection of this type of fault. When *BFT-Bench* triggers this type of fault, instead of sending messages according to the protocol specifications, the replica process sleeps during a given delay, before resuming to send any messages to other replicas.

Network Flooding. Network flooding is a common denial-of-service attack. It is meant to overload the network with malicious messages which can not be said invalid until verified. This verification of messages is computation-intensive and prevents the system from focusing on correct messages. In practice, BFT protocols prototypes are extended to integrate the injection of this type of fault. When *BFT-Bench* triggers this type of fault, the faulty replica transmits corrupted messages of a chosen size to other replicas.

System Overload. Overloading the system with a large number of concurrent client requests can affect system performance to a large extent. Although none of the servers behave maliciously in this attack, but continuous increase in concurrent clients can eventually deteriorate the performance or lead to system thrashing. To inject this behavior, *BFT-Bench* remotely connects to the node in charge of emulating concurrent clients, and starts additional client processes.

Faultload. A *faultload* in *BFT-Bench* is described in a file. Each line of the faultload file consists of the following elements: the time at which a fault occurs (relative to the beginning of the experiment), the type of fault that occurs, where the fault occurs, and optionally, additional parameters that depend on the type of fault. A fault belongs to one of the fault types handled by BFT protocols prototypes, and introduced in Section 3.2. A fault occurs in one of the BFT protocol replicas; this replica may be either explicitly specified in the faultload or randomly chosen among the set of replicas.

Thus, a faultload in *BFT-Bench* may contain the following element to describe the injection of fault of type crash:
<[*fault trigger time*], *replica crash, replica*$_x$>

It may contain the following element to describe the injection of fault of type message delay, specifying among others the delay to be injected, and the duration of occurrence of this type of fault:
<[*fault trigger time*], *message delay, replica*$_x$, *(*[*injected message delay*], [*fault occurrence duration*]*)*>

A faultload in *BFT-Bench* may also contain the following element to describe the injection of fault of type network flooding, specifying among others the size of the message used for flooding, and the duration of occurrence of this type of fault:
<[*fault trigger time*], *network flooding, replica*$_x$, *(*[*flooding message size*], [*fault occurrence duration*]*)*>

The overall architecture of *BFT-Bench* fault injection is presented in Figure 3. In this example, the cluster has $N+2$ nodes, where $N = 3f+1$ nodes are BFT replicas, one node hosts concurrent clients emulator, and one node runs *BFT-Bench*. *BFT-Bench* faultload injector uses faultload to determine which type of fault is to be triggered, at what time this fault will be injected, and other required fault parameters. The fault injector runs a daemon that communicates directly with the replicas to trigger faults. For instance, in case of *replica crash*, the daemon waits until the *fault trigger time* is reached, then calls remotely interacts with the target replica to actually trigger the fault.

3.3 Load Injection

The workload is first characterized by number of concurrent clients sending requests to the BFT system. Client requests are executed in FIFO order in a closed loop, where a clients submits a request, waits for the request to get processed and receives a response, before sending another request. The workload is also

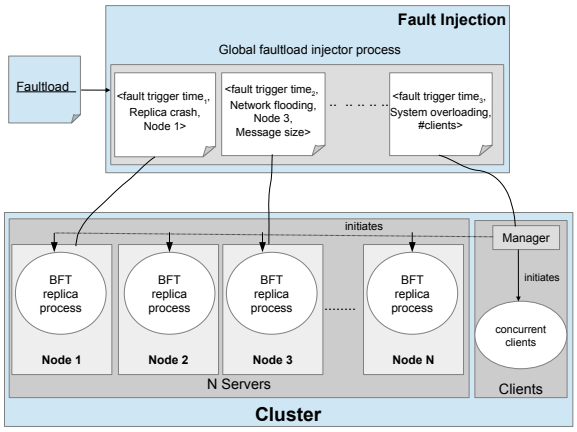


Fig. 3. Architecture of Faultload Injection

characterized by the size of client request/response messages exchanged with the BFT system. It is an important parameter as large size messages affect BFT system performance, due to time consuming cryptographic operations executed by BFT protocols. *BFT-Bench* includes a client emulator implementing multi-client behavior, where each client process sends requests to the underlying BFT system, and receives the corresponding responses. In *BFT-Bench*, the workload may contain one or several elements as follows to describe the load to be injected:
<[*load injection time*], [*#concurrent clients*], [*request message size*], [*response message size*], [*request processing time*], [*load injection duration*]*)*>

3.4 Monitoring

BFT-Bench produces performance statistics for evaluating and comparing the performance of BFT protocols. *Throughput* and *Latency* are the main performance parameters considered when evaluating Byzantine Fault Tolerance protocols, both experimentally and theoretically. *Latency* is the time elapsed from the moment a client submits a request until the complete response is received by this client. *Throughput* is measured as the number of client requests handled by the system per unit of time. Latency and throughput are measured by *BFT-Bench* at the client-side, and thus include network communication times between the client and the replicas. Furthermore, *BFT-Bench* produces low-level system monitoring information such as cpu, memory and network usage, that can help better explaining the behavior and possible bottlenecks of the system.

3.5 On Extensibility of BFT-Bench

BFT-Bench is an open framework intended to help BFT protocol designers and practitioners to easily evaluate their protocols. *BFT-Bench* includes, among others, existing implementations of BFT protocols. In this paper, we illustrated the use of *BFT-Bench* with several state-of-the-art BFT protocols prototypes. In the following, we describe how to integrate a new BFT protocol prototype to *BFT-Bench*, and benefit from its benchmarking features. Although most of the components of *BFT-Bench* framework are general and can be easily reused for new BFT protocols, there are some exceptions that we describe below. Workload injection is based on the client emulator program that comes with a BFT protocol prototype. Such a program is pretty simple, and its reuse to allow dynamic workload variation as provided by *BFT-Bench* is straightforward. For the implementation of faultload injection for faults like replica crash or system overload, the implementation is independent from the actual BFT protocol prototype. This is not the case of faults of type message delay or network flooding that need an extension of the underlying BFT protocol prototype.

4 Experimental Evaluation

4.1 Experimental Setup

The experiments presented in this paper were conducted on a cluster of Grid'5000 [5]. Each node hosts two Quad-Core Intel Xeon E5420, with 2.50 GHz, 8 GB of RAM, and 160 GB of storage; nodes are connected through 1 GB Ethernet. *BFT-Bench* framework currently includes six BFT protocols, namely PBFT, Chain, RPFT, Aardvark, Aliph, Zyzzyva [6, 15, 3, 7, 2, 9]. We used the original C++ code of these protocols. When needed by the evaluated BFT protocol, multiple virtual network interfaces are created on a single physical network interface controller to exploit the robustness of protocol, e.g., Aardvark, RBFT.

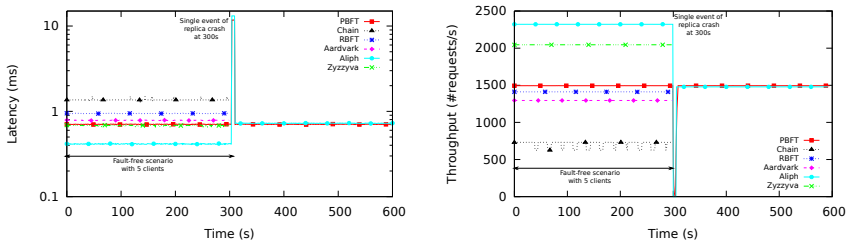
For each protocol under evaluation, four nodes are used for running the replicas of the service (i.e., application), thus, $f = 1$. Two other nodes are used for the experiments, one for emulating the clients that concurrently send requests to the replicated service, and one node for hosting *BFT-Bench*. Similarly to state-of-the-art evaluations, each replica runs an echo service [6]. Client request size and client response size are 4 KB each. Furthermore, to emulate the computation performed by the service, a delay of 100 ($\pm$10%) μs is introduced before sending the response to the client. The results of the experiments are obtained after a warm-up phase of 180 s, to let the system reach a stable stage before actually measuring the behavior of the system. The graphs presented in the following are obtained after the warm-up phase.

4.2 Evaluation in Presence of Replica Crash

In this use case, five concurrent clients access the replicated service, when the crash of the primary replica of the service occurs. Thus, the following faultload

is provided to *BFT-Bench*, which triggers a fault at time 300 s:
<300 s, replica crash, {primary}>

Figure 4 presents the measured latency and throughput. Upon crash of the primary, PBFT induces a sudden increase in latency, and throughput drops sharply. This is due to the view change mechanism used by the protocol to replace the faulty primary. Aliph follows the same pattern since it switches to PBFT upon fault occurrence. Upon crash, Chain cannot maintain its pipeline structure as the successor of the crashed server never receives any message. In theory, Chain must switch to PBFT upon crash, but unfortunately this mechanism is not present in the Chain prototype. Zyzzyva prototype implements only the fault free version of the protocol and, thus, does not deal with fault occurrence. In Aardvark and RBFT where clients broadcast requests to all replicas, and because of the absence of a crash handling mechanism at client side, this fault is not handled.



(a) Latency (b) Throughput

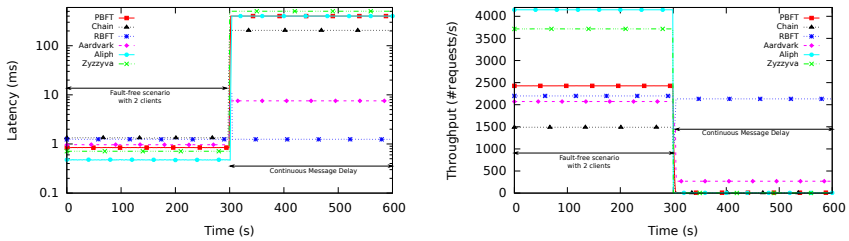
Fig. 4. Performance evaluation in presence of replica crash

4.3 Evaluation in Presence of Message Delay

In this use case, a replicated service is accessed by two concurrent clients, when the service starts misbehaving by inducing intentional and unjustifiable message delay. The following faultload is provided to *BFT-Bench* to inject this misbehavior in the running replicated service:
<300 s, message delay, replica$_x$, (500 ms, 300 s)>

Here, starting from time 300 s there is a message delay of 500 ms, and this misbehavior continues during 300 s (i.e., until the end of the experiment). Since the BFT protocols under evaluation use different architectures and communication patterns, message delays are introduced by *BFT-Bench* in different ways to these protocols, as explained in the following. For instance in case of PBFT, Aardvark, RBFT, and Zyzzyva, a delay is injected at the primary replica-side, when this replica receives a client request and sends the initial message to other replicas for processing that request (i.e., usually known as the *pre-prepare* phase in these protocols). In case of Chain, a message delay is injected before the head

replica initiates the communication protocol with the other replicas. For Aliph which does not have a dedicated replica (i.e., no primary, no head), a chosen replica induces message delay.



(a) Latency (b) Throughput

Fig. 5. Performance evaluation in presence of intentional message delay

Figure 5 presents the results of these experiments. We can observe that the impact of this type of fault is different from one protocol to another. For instance, Figure 5(a) shows that message delay faults induce a latency increase of two orders of magnitude for PBFT, Zyzzyva, Aliph, Chain protocols, and a latency increase of one order of magnitude for Aardvark. Interestingly, due to its robustness to this type of fault, RBFT is able to smoothly tolerate this misbehavior without a perceptible impact on performance.

4.4 Evaluation in Presence of Network Flooding

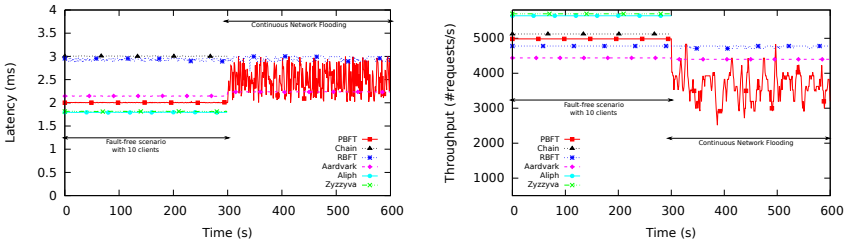
In this case, ten clients concurrently access a replicated service, when the service starts misbehaving by inducing network flooding. The following faultload is provided to *BFT-Bench* to inject this misbehavior in the running replicated service:

<300 s, network flooding, replica$_x$, (4 KB, 300 s)>

Thus, starting from time 300 s, replica$_x$ starts sending corrupted messages of size 4 KB to other replicas, during 300 s. Figure 6 presents the results of these experiments. Interestingly, Aardvark and RBFT are robust in case of such misbehavior. They are able to detect that a replica performs network flooding, and counter it by black-listing that replica [7, 3]. In contrast, PBFT has bad performance in case of network flooding, since it is not able to tolerate this type of misbehavior. Aliph, which switches to PBFT when faults occur, demonstrates similar behavior as PBFT, although for clarity purposes its results after fault occurrence are not included in Figure 6.

4.5 More Complex Scenario

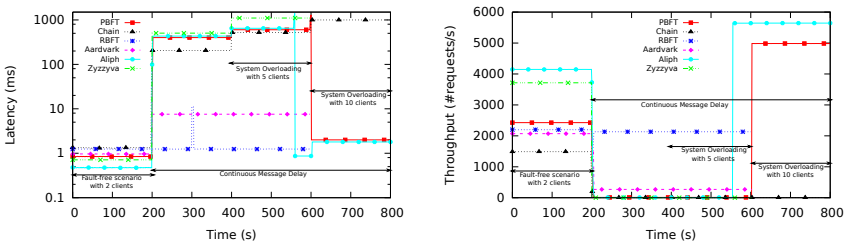
This use case illustrates a more complex scenario where a fault tolerant service faces a Byzantine fault, in addition to service contention. Here, the following



(a) Latency (b) Throughput

Fig. 6. Performance evaluation in presence of network flooding. Results of Aliph after fault occurrence are not included; Aliph switches to PBFT when fault occurs and demonstrates similar behavior as PBFT.

faultload is used by *BFT-Bench*:
<200 s, message delay, (replica$_x$, 500 ms, 600 s)>
And in order to increase service contention, the following *workload* is provided to *BFT-Bench*:
<0 s, 2, 4 KB, 4 KB, 100 *μs, 400 s>*
<400 s, 5, 4 KB, 4 KB, 100 *μs, 200 s>*
<600 s, 10, 4 KB, 4 KB, 100 *μs, 200 s>*



(a) Latency (b) Throughput

Fig. 7. Performance evaluation - combination of message delay and system overload

Thus, the replicated service is first accessed by two concurrent clients. Then at time 200 s, the service starts misbehaving by inducing abnormal message delay of 500 ms, during 600 s (i.e., until the end of the experiment). In addition, the service load increases from 2 clients at the beginning of the experiment to 5 clients at time 400 s, and then to 10 clients at time 600 s. And as described in Section 4.1, client request and response message sizes are 4 KB.

Figure 7 presents the results of the experiment. We can observe that RBFT is able to transparently tolerate the Byzantine fault of type message delay when service contention is not too high. However, when 10 clients concurrently access

the service, RBFT is no more able to handle contention and terminates. In case of Aardvark, the Byzantine fault tolerant service is able to face message delay fault, but at the expense of a performance overhead of one order of magnitude. Interestingly, Aardvark smoothly handles service contention increase without a perceptible impact on performance. This holds up to a given service load, where with 10 concurrent clients, Aardvark is no more able to handle contention, and terminates. Zyzzyva and Chain are able to face the Byzatine fault of type message delay. But they induces a high performance overhead of two orders of magnitude when such a fault occurs. In addition, when the service has a high contention (10 concurrent clients), Chain-based replicated service is three orders of magnitude slower, while Zyzzyva crashes. PBFT and Aliph (which switches to PBFT upon fault occurrence) have similar behavior after the occurrence of the fault at time 200 s, with a drop of latency of two orders of magnitude. After a while, PBFT and Aliph undergo a *view change*, i.e. they replace the faulty primary by a new primary. This has a direct impact on service performance which drastically improves.

5 Conclusion

Performance and dependability are important requirements of today's computing systems. Byzantine Fault Tolerance (BFT) is a general approach to make these systems, theoretically, tolerate arbitrary faults. BFT protocols were extensively investigated in the last years, and various prototypes were proposed. However, to the best of our knowledge, there is no practical solution to precisely identify the varying nature of Byzantine behaviors, no general tool for real-time injection of these misbehaviors in a system, and no reusable environment for the empirical evaluation of various BFT protocols. This paper presents *BFT-Bench*, the first framework for evaluating BFT implementations under different faulty behaviors and workloads. *BFT-Bench* framework includes several state-of-the-art BFT protocols, automatically deploys them, injects different types of faults at different rates, and produces performance and dependability measures. The evaluation results show that *BFT-Bench* is able to successfully compare various BFT protocols, in various faulty behaviors. We wish to make BFT benchmarking easy to adopt by developers and end-users of BFT protocols. *BFT-Bench* framework aims to help researchers and practitioners to better analyze and evaluate the effectiveness and robustness of BFT systems. Although this paper concentrates on presenting the current version of *BFT-Bench* with some BFT protocols and their related fault types, we believe that the proposed approach can be easily extended to other BFT protocols, and other faulty behaviors.

Acknowledgement

This work was supported by AMADEOS (Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems), a collaborative project funded under the European Commission's FP7 (FP7-ICT-2013-610535). The

experiments were conducted on the Grid'5000 experimental testbed, developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities, as well as other funding bodies.

References

1. Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine Replication Under Attack. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008)*, 2008.
2. P.-L. Aublin, R. Guerraoui, N. Knezevic, and V. Quéma. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.*, 32(4), 2015.
3. P.-L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *The IEEE 33rd International Conference on Distributed Computing Systems (ICDCS 2013)*, 2013.
4. A. N. Bessani, J. Sousa, and E. A. P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *The 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.
5. F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing*, 2005.
6. M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *The 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, 1999.
7. A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2009)*, 2009.
8. J. Durães, M. Vieira, and H. Madeira. Dependability Benchmarking of Web-Servers. In *The 23rd International Conference on Computer Safety, Reliability and Security (Safecomp'2004)*, 2004.
9. R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.
10. H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru. Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations. In *The 34th Int. Conf. on Distributed Computing Systems (ICDCS 2014)*, 2014.
11. M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2), 1980.
12. A. Sangroya, D. Serrano, and S. Bouchenak. Benchmarking Dependability of MapReduce Systems. In *The IEEE Int. Sym. on Reliable Distributed Systems (SRDS)*, 2012.
13. F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4), Dec. 1990.
14. A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *The 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
15. R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *The 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, 2004.
16. G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin One's Wheels? Byzantine Fault Tolerance With a Spinning Primary. In *SRDS*, 2009.