BFT-Bench: A Framework to Evaluate BFT Protocols

[Work-In-Progress Paper]

Divya Gupta
Univ. Grenoble Alpes, LIG
Grenoble, France
divya.gupta@imag.fr

Lucas Perronne
Univ. Grenoble Alpes, LIG
Grenoble, France
lucas.perronne@imag.fr

Sara Bouchenak
INSA Lyon, LIRIS
Lyon, France
sara.bouchenak@insa-lyon.fr

ABSTRACT

Byzantine Fault Tolerance (BFT) has been extensively studied and numerous protocols and software prototypes have been proposed. However, most BFT prototypes have been evaluated in an ad-hoc setting, considering different fault types and fault injection scenarios. 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 the online monitoring and reporting of performance and dependability measures. Preliminary results 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 [15]. 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 [16, 12, 11, 10, 8, 17], 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. [2, 7, 9, 1]. 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 presents an overview of BFT-Bench. Section 3 describes the experimental evaluation, and Section 4 concludes the paper.

2. OVERVIEW OF BFT-Bench

We present *BFT-Bench*, a novel framework for empirical evaluation and comparison of Byzantine Fault-Tolerant systems.

2.1 BFT Protocols in Consideration

BFT-Bench is intended to be an open framework, that includes BFT protocol prototypes, and that may be extended with new BFT protocols. In the following, we consider state-of-the-art BFT protocols: PBFT for being the first practical BFT protocol [6], Chain for its performance efficiency in fault-free conditions [10], and RBFT as an instance of robust protocol that minimizes performance in presence of failures [2].

PBFT is considered the baseline of BFT protocols [6]; and its communication pattern is used by many other protocols [7, 2, 4]. In PBFT, there is a primary and replicas that

ICPE'16, March 12-18, 2016, Delft, Netherlands

DOI: http://dx.doi.org/10.1145/2851553.2858667

interact through three stages of message exchanges, before the client can commit its request. First, primary sends pre-prepare messages to other replicas with assigned sequence number to each request. The two following message stages, prepare and commit, are dedicated to the exchange and validation of the sequence numbers proposed by the primary. PBFT ensures *Safety* and *Liveness*; upon violation of these properties the primary is suspected to be faulty and thus, a primary view change is initiated by replicas.

As the name suggests, Chain has a chain-like communication pattern for replicas that greatly benefits from the batch optimization (i.e. multiple messages in one batch) [10]. Thus, Chain allows to handle a high load of requests. Chain must rely on a protocol switching mechanism when subject to failures.

RBFT is a robust protocol that strengthens the architecture of PBFT and incorporates fault adaptive mechanisms to deal with certain faulty behaviors [2]. RBFT runs $f + 1$ multiple instances of the same 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 instances differs by a defined threshold at more 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.

2.2 Faultload

Faults can occur accidentally or can be induced intentionally. Users of BFT-Bench framework can generate various faultloads involving different faulty behaviors. Each faultload contains various information which we describe below:

- ***Fault Trigger Time***: The *fault trigger time* contains the time stamp at which the fault must be triggered.
- ***Fault Type***: Byzantine faults encompass numerous faulty behaviors, e.g. hardware failure, software failure, network congestion, etc.
- ***Fault Parameters***: Different faults may require additional fault parameters at time of fault injection. According to the type of fault to be injected, fault parameters might vary. For *replica crash*, *message delay* & *network flooding*, the location of the fault must be specified, whereas in *system overloading*, the location is irrelevant since no replica acts faulty. For *network flooding*, the size of the corrupted messages is an important factor, whereas for *message delay*, the value of the delay introduced before sending a message must be specified.

Fault types that are tackled by the considered BFT protocols are the following:

Replica Crash. Crash of a server is a common performance failure that can happens in a system. Upon crash, the server stops completely and do not participates in any further communication with the clients or the servers. Most of the industries like Salesforce, Amazon, Oracle, etc, rely on Paxos[14, 13] for handling crash but are unable to detect byzantine faults and face challenges due to disrupted availability. BFT protocols consider crash as yet another byzantine fault.

Message Delay. Delaying the sending of messages benefits from the difficulty to distinguish a faulty replica from a slow network. When a replica starts delaying of messages, it slows down all future operations depending on these messages. As described in section 2.1, most of BFT protocols ensure the *Safety* property by reaching an agreement on the total order of execution of the requests. If the messages containing these information are delayed, then the whole protocol is delayed, thus leading to degradation in performance. This byzantine behavior is thus especially critical when it occurs at the primary.

Network Flooding. Network flooding is meant to overload both the network and the computational resources with malicious messages which cannot be said invalid until verified. This verification of messages consumes a lot of computational cycles and prevents the resources from focusing on the correct messages.

System Overload. Overloading the system with a large number of requests sent by a large number of clients can prove to be catastrophic and can affect the performance to a large extent. Although none of the servers behave malicious in this attack, but continuous increase in concurrent clients can eventually deteriorate the performance or lead to system failure.

2.3 Workload

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 client submits a request, waits for the request to get processed and receives a response, before sending another request. The workload is also 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 corresponding responses.

2.4 Performance and Dependability Analysis

BFT-Bench produces statistics for performance metrics, namely *Throughput* and *Latency*. The former is the number of client requests handled by the system per unit of time, and the latter is the time elapsed from the moment a client submits a request until the complete response is received by this client. *Availability* is measured in terms of time when the service is available, i.e., the service is responding. It is the ratio of the time the service was returning responses (correct or incorrect) to the total time the service was meant to run. It is usually measured over a period of time, usually in terms of days, months or years. BFT protocols should theoretically be 100% reliable and available. The experimental evaluation (see Section 3) describes how well they perform in practice.

3. EXPERIMENTAL EVALUATION

In this section we present a preliminary comparative analysis of the three BFT protocols, PBFT, Chain and RBFT when facing different types of faults.

3.1 Experimental Setup

Our experiments were conducted on a cluster running in Grid'5000 [5] composed of 34 nodes. Each node hosts two 4-core Intel Xeon E5420 QC processors at 2.50GHz frequency with 8GB of RAM and 160GB SATA of storage space. Machines are interconnected through 1 Gigabit Ethernet and have only a single network interface. In the experiments we consider a cluster of 4 nodes $(3f + 1)$ for running BFT protocol instances. We reserve 2 extra nodes, one for concurrent clients' emulator, and one for hosting BFT-Bench framework. A client requests incurs 30 ($\pm$10%) milliseconds emulating application computations. We use *a/b* micro-benchmark by Castro and Liskov [6] for evaluating throughput and latency for each faulty behavior. We used original versions of the code bases for the three protocols in consideration[1].

3.2 Replica Crash

Here, we consider the Byzantine misbehavior described by the following faultload:
$<$*300s, replica crash,* $\{$*primary*$\}>$, defining *fault trigger time*, *fault type* and *fault location*, respectively.

For our evaluation, we consider crash of primary. Since primary replica is responsible for ordering the incoming requests, its crash leads to expensive *view change protocol* initiated by other replicas, thus degrading the overall performance. In our experiments, all the backup replicas wait for 5 seconds before considering primary to be unresponsive. In case of non primary crash, protocol continues as replicas need only 2f+1 matching responses.

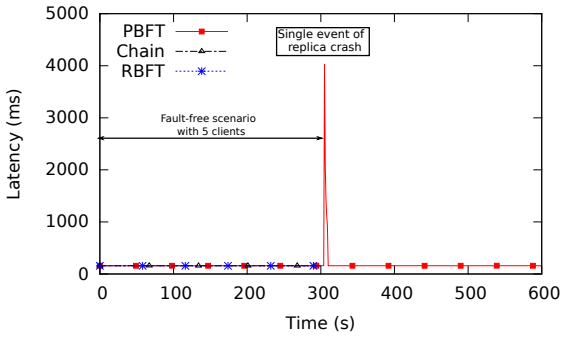
Figure 1 presents the performance of the prototypes when a primary crashes. In the results for PBFT, we observe a sudden increase in latency (Figure 1-a), and throughput (Figure 1-b) drops sharply upon crash of the primary. This is due to the *view change protocol* which replaces the faulty primary. Prototypes for Chain and RBFT fail to respond once the primary crashes. Upon crash, Chain cannot maintain its pipeline structure as the successor of the crashed server never receives any messages. Chain must switch to PBFT upon crash, but unfortunately this mechanism is not present in the original prototype. We would have observed the same performance as PBFT if switching was possible [3]. In RBFT, clients broadcast requests to all replicas. During crash fault, client enters an infinite request re-transmission loop while attempting to send request to the crashed replica. This is due to the absence of a crash handling mechanism at client side.

3.3 Network Flooding

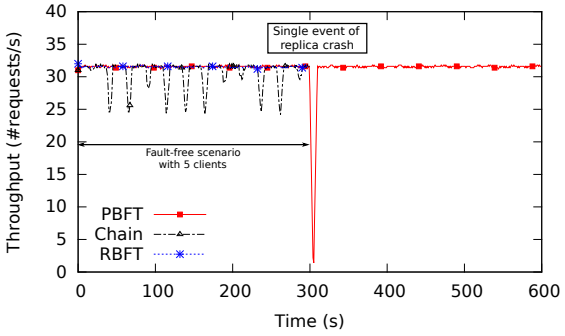
Figure 2 presents the performance of PBFT & RBFT when a non-primary replica starts to flood (sends as many malicious/corrupt messages as possible) other replicas. Faultload used is as follows:
$<$*300s, network flooding,* $\{Replica_2, 4KB\}>$, where $Replica_2$ will start to flood other servers with corrupt messages of size 4KB at 300s.

BFT-Bench implements this behavior by forcing a replica to enter an infinite loop of continuous transmission of malicious messages to other servers until the end of the experiment. We observe that any replica, either primary or non primary would impact the performance in the same way.

[1]Code base of PBFT was downloaded from http://www.pmg.csail.mit.edu/bft/#sw whereas RBFT and Chain implementations were obtained directly from authors [10, 2].



(a) Latency



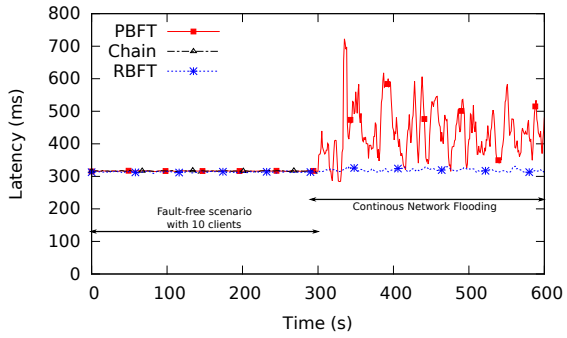
(b) Throughput

Figure 1: Evaluation in presence of replica crash

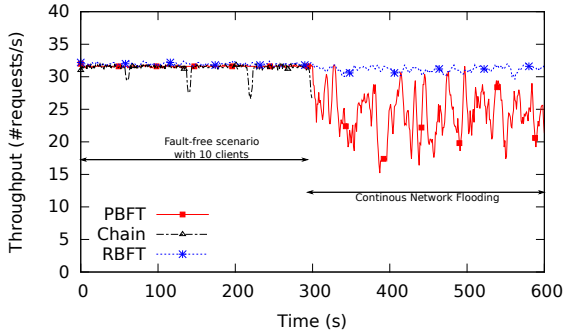
The results illustrate that Chain makes no progress upon fault injection while performance of PBFT becomes sporadic. This is due to the expensive time consuming cryptographic operations performed over corrupt messages by all the replicas in PBFT and successor replica in Chain. Inability to handle corrupt messages introduces a gap in the communication pattern and lack of protocol switching mechanism, holds the Chain from continuing.

RBFT uses multiple NICs to avoid malicious clients and replicas from flooding client-to-replica & replica-to-replica communications. RBFT also employs flood adaptive mechanism where non-faulty replicas can detect a flooding replica and blacklist it [2]. Flood protection enables a non-faulty replica to monitor the number of messages (including correct & malicious messages) received. If a non faulty replica receives more than a specific number of messages from a particular replica in a period of time, then it can label this replica as faulty and initiate a blacklisting protocol. When this happens, RBFT closes the NIC of the misbehaving replica for some time but after a given period it rejoins the system again. Due to this we observe slight variations in performance with upto 5% of degradation.

4. CONCLUSION



(a) Latency



(b) Throughput

Figure 2: Evaluation in presence of network flooding

This paper presented BFT-Bench, the first framework for evaluating BFT implementations under different faulty behaviors and workloads. BFT-Bench framework includes three state-of-the-art BFT protocols, automatically deploys them, allows to generate different types of faults, injects them at different locations and different rates, and computes performance and dependability measures. The paper presented preliminary experiments conducted with BFT-Bench. 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.

Acknowledgement

This work was partly supported by AMADEOS, a collaborative project funded under the European Commission's FP7 (FP7-ICT-2013-610535). The experiments presented in the paper were conducted in Grid'5000, an experimental testbed developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities, as well as other funding bodies.

5. REFERENCES

[1] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine Replication Under Attack. In *DSN*, pages 197–206, 2008.

[2] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *ICDCS*, pages 297–306, 2013.

[3] J.-P. Bahsoun, R. Guerraoui, and A. Shoker. Making BFT Protocols Adaptive.

[4] A. Bessani, J. Sousa, and E. E. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 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 *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, pages 173–186, 1999.

[7] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, pages 153–168, 2009.

[8] M. Correia, N. F. Neves, and P. Veríssimo. BFT-TO: Intrusion Tolerance with Less Replicas. *Comput. J.*, 56(6):693–715, 2013.

[9] G. S. V. et. al. Spin one's wheels? byzantine fault tolerance with a spinning primary.

[10] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The Next 700 BFT Protocols. In *EuroSys*, pages 363–376, 2010.

[11] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. Stretching BFT. Technical report, Technical Report EPFL-REPORT-149105, EPFL, 2011.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.

[13] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[14] L. Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, 2001.

[15] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[16] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, pages 91–104, 2004.

[17] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.