WebBFT: Byzantine Fault Tolerance for Resilient Interactive Web Applications

Christian Berger, Hans P. Reiser

WebBFT: Byzantine Fault Tolerance for Resilient Interactive Web Applications

Christian Berger, Hans P. Reiser

University of Passau, Germany
bergerch@fim.uni-passau.de, hr@sec.uni-passau.de

Abstract. Byzantine fault tolerant (BFT) applications are usually implemented with dedicated clients that interact with a set of replicas with some BFT protocol. In this paper, we explore the possibility of using web-based clients for interaction with a BFT service. Our contributions address the trustworthy deployment of client code and configuration in a browser-based execution environment (client bootstrapping), the design and implementation of a BFT client within the constraints of a browser-based JavaScript execution environment, and publish-subscribe extensions to the standard request/reply interaction model of BFT state machine model to simplify the implementation of efficient interactive web applications.

1 Introduction

Many client/server applications need to meet requirements regarding reliability and resilience. As the Internet advances our world to further grow together and people and devices become more and more interconnected, the role that these systems play becomes increasingly important. Many applications put high requirements on the availability and reliability of such systems.

A way to provide reliability and resilience in distributed systems is state machine replication (SMR). The state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with independent server replicas [18]. That way, the service remains functional even if up to a defined maximum number of replicas fail. The way in which replicas fail is described by a fault model. Byzantine fault tolerance describes the ability of a distributed system to tolerate arbitrary and malicious behavior, even involving collusion with other faulty components [12].

In recent years, many native applications have been transformed into web applications, making them easily accessible for a wide variety of client devices. Some notable examples are Google Docs[1], Pixlr[2], and ShareLatex[3]. However, web applications are restricted by their execution environment [6], e.g., in regard

[1] Google Docs is an online office suite, see https://www.google.com/docs/about/
[2] Pixlr is a popular online tool for photo editing, see https://pixlr.com/
[3] ShareLatex is an online Latex editor, see https://sharelatex.com/

of storage or computation capabilities. Current state-of the-art BFT SMR frameworks like BFT-SMaRt [2] provide support for only native client applications. In this paper we discuss the challenges and limitations that are induced by the limited execution environment of web applications and implement a prototype of a web client which connects to a BFT service by employing a state-of-the-art replication library. The main research questions addressed by this paper are the following:

- *Interface between BFT SMR framework and web client:* How can a web service be equipped with BFT capabilities and how to solve challenges and limitations when designing the interface between a web client and a BFT SMR framework?
- *Bootstrapping and authentication:* How can the web client discover the replica set, obtain the client application code and verify its integrity? How can clients verify the authenticity of the replicas?
- *Execution model of web services:* Can we deviate from the traditional SMR request/reply programming model and implement other forms of interaction such as a publish/subscribe mechanism to match the requirements of state-of-the-art interactive web application like web group editors?
- *Performance of BFT web services:* How do BFT web applications perform in comparison to their native counterparts?

This paper is structured as follows. Section 2 provides some background information and analyses the challenges that arise when making a client/server web application resilient to Byzantine server faults. Section 3 presents our solution of a web-enabled BFT replication framework and the design of a replicated shared online editing service based on that framework. Section 4 provides an in-depth evaluation of our framework using micro benchmarks and using a replicated shared editing service. Section 5 discusses related work on fault-tolerant web services, and Section 6 summarizes our contributions.

2 Problem Description

2.1 Background

Since Castro et al. [5] presented one of the first practically feasible solutions for coping with Byzantine faults in distributed systems, many other articles have proposed enhancements. One of the most prominent implementations of a BFT state machine replication library is the Java-based BFT-SMaRt system [2]. It is open source and can be used to implement client-server applications on top of a robust BFT state machine replication protocol. BFT-SMaRt defines a generic client and server interface to be implemented by the application, based on a request-reply model: Client send requests using an `invoke(command)` method and the server implements an `execute(command)` method, which may change the application state and computes a result that is sent back to the client.

2.2 Challenges

In contrast to native client applications, web clients are faced with some restrictions in their execution model:

- Instead of being able to use all communication mechanisms provided by the operating system (such as creating TCP sockets), web clients are limited to use HTTP requests or WebSockets for communication.
- Web applications rely on a browser-based execution environment that is running a single-threaded JavaScript application. Functions to invoke requests for execution on the replicated server need to be implemented asynchronously; the client should never block and wait for responses.
- For persistently storing data, the option that web clients have is restricted to the WebStorage implementation of the browser.
- The client-side bootstraping process (replica discovery, client-side code and key material deployment) is done every time the application is accessed by the client web browser, while it is typically done only once for native client applications (see Section 3.2).
- Last but not least, interactive web application require an interaction model that goes beyond the traditional request/response interaction model that is regularly used by native clients. For many kinds of applications, a server-initiated client update mechanism can be very useful.

2.3 System Model

In our architecture, a variable number of clients interact with a group of replicated servers. Our server-side implementation is based on BFT-SMaRt, which assumes an eventually synchronous system model and implements reliable authenticated channels on top of TCP connections for server-server communication. For client-server communication, our system differs from the standard BFT-SMaRt model, due to requirements of browser-based web applications. In addition, we do not expect the client to have a-priori knowledge of the replica set or public keys of the replicas. We assume that the user starts utilizing the web application by accessing the service URL in the browser without any additional knowledge. Nevertheless, we require that a solution for web clients for a BFT replicated services fulfill equivalent reliability and authentication properties.

The number of clients and replicas can change over time. A *view change* is required if a replica joins or leaves the system. As in BFT-SMaRt, the total number or the identity of replicas can be changed only by a trusted third party, which we call the *ViewManager* client. The fault model is defined by BFT-SMaRt. Faulty replicas can behave arbitrarily. As usual, we assume that *arbitrary* behaviour is limited to computationally feasible behaviour. In particular, we assume that a faulty node is not able to break strong cryptography and, e.g., forge signatures or MACs of other correct nodes. We always require that at most less than a third of the n_c replicas in the current view c are faulty at the same time, i.e. $n_c > 3f$ always holds.

We guarantee replica consistency (i.e. safety) in the presence of malicious clients. BFT-SMaRt by default uses MACs for client message authentication. While we still guarantee liveness in the sense that correct client requests will eventually be processed by the system, the system's performance may be extradited to degradation attacks if we consider the possibility of malicious clients. We investigate on the impact of such an attack in Section 4.3.

2.4 Trust Assumptions

The main focus of our design is to protect benign clients from getting incorrect service from faulty/malicious replicas. Standard BFT-SMaRt assumes that at the client side, the correct code is manually installed and a configuration file with all server details is manually distributed to all clients. In our design, we investigate possibilities on how to automate trustworthy client setup and configuration without relying on a fully manual installation. The trustworthiness of this automation depends on the following trust assumptions:

- *The client's computer, its operating system, the web browser and the execution environment for JavaScript in the web browser used for executing the WebBFT client are trusted.*
  If an adversary is able to manipulate any of these entities, he can subvert our trustworthy code and configuration deployment mechanisms, and thus we cannot guarantee trustworthy interaction with the replicated service.
- *The browser has a trusted way to find the replica set.*
  There is a trustworthy lookup mechanism, such as a trustworthy DNS implementation available.
- *There is a trustworthy PKI infrastructure, which can be used to authenticate each replica belonging to a replicated web service.*
- *The browser does not trust a single replica to deliver the correct client code, deliver the configuration or replica set information, initiate a reconfiguration of the system's current view, or respond with the correct result to a request that was invoked by the browser.*
  Some quorum-based or equivalent mechanism makes sure that interacting with up to f malicious replicas does not cause an incorrect operation.

A trusted client environment is however not relevant for the server replicas or the safety and liveness properties of the system (we will explore the possibility of malicious clients in Section 4.3). If one of these assumptions is not met by a specific client, then that client is potentially unable to tolerate Byzantine server faults.

3 Solution

In this section, we present the design of an interface between the BFT-SMaRt framework and a JavaScript client that can be executed in a web browser, define extensions for interactive web applications and present a web-based shared

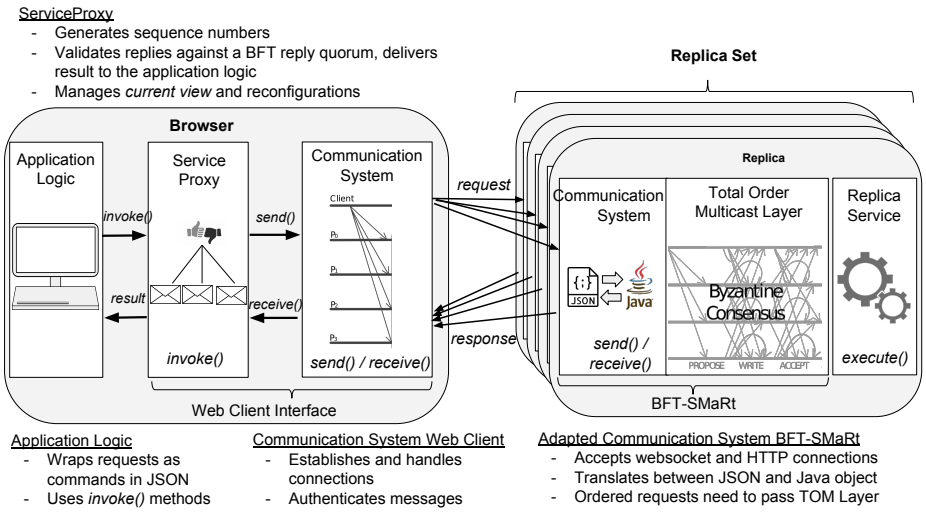


Fig. 1. Overall architecture with our web client interacting with service replicas

text editor as example application. Our client software is based on Typescript, uses Angular as framework for its Model-View-Controller mechanism that allows us to update the view (HTML templates), and is designed as a state-of-the-art Ajax application with methods that send requests to replicas being asynchronous. We decided to use JSON as data exchange format. A service client can be implemented by writing a JavaScript front-end application layer that uses our WebBFT interface and by providing a service replica implementation in Java that implements the ordinary BFT-SMaRt server interfaces.

3.1 Architecture

The complete, modular 3-tier architecture of our solution is shown in Figure 1. The client application logic wraps its requests as commands in JSON format and uses an `invoke()` call to the underlying `ServiceProxy`. The `ServiceProxy`'s `invoke()` methods are all asynchronous and thus non-blocking, and receive an additional `ReplyListener` object as parameter. The `ServiceProxy` registers at the `CommunicationSystem` as listener and implements a `replyReceived()` method that is triggered every time the `CommunicationSystem` gets an authenticated reply from a replica. After receiving a valid reply, the `ServiceProxy` invokes the application callback implemented by the `ReplyListener` object.

The `ServiceProxy` implements the `invoke()` interface, which allows the application to multicast requests to the replicas and generates sequence numbers for outgoing requests. It validates the responses from the replicas against the BFT reply quorum. We distinguish between ordered requests and unordered requests. **Ordered requests** are requests that may have state changing effects on the replicas (typically write requests). They need to pass the *Total Order*

Multicast Layer of BFT-SMaRt, in which the leader proposes an order and a consensus instance is run among the replicas. Unordered requests are read-only requests. They can be passed directly to the replicas thus avoiding the ordering process and improving the system's performance. With the read-only optimization the client needs $2f + 1$ matching responses to ensure linearizability [3, 11] since replicas may be in different states when they reply to unordered requests. The `ServiceProxy` is also responsible for implementing the reconfiguration protocol on the client side. If a validated reply indicates a higher view number than the current view of the client, this will trigger a client-side view update.

The `ServiceProxy` will call the `CommunicationSystem` to update its connections, e.g. close old connections or to establish new ones.

The **client-server communication system** is responsible for handling the connections to the replicas (establishing connections, closing connections), validating the MACs of the incoming replicas' responses, computing and attaching MACs to the outgoing messages as well as sending and receiving messages to/from all replicas. We provide two options for client-server communication, using simple HTTP requests and using websockets.

For the **WebBFT server interface**, we do not make any changes to the server interfaces of BFT-SMaRt. Developers can implement a `ServiceReplica` instance as they would do for plain BFT-SMaRt, with one recommendation: BFT-SMaRt uses a simple byte array for the data field for all requests and replies, and the BFT-SMaRt developers suggest to store serialized Java objects in this byte array. With WebBFT clients, it is preferable to use JSON strings for encoding application data, because this format can easily be understood by all clients (including web clients) and servers.

For server side development, we adapt the BFT-SMaRt framework, with the only significant changes made to the server's `CommunicationSystem` class, where we created additional protocol handlers to support HTTP and WebSocket communication in a new *Netty* server pipeline. This pipeline is bootstrapped parallel to the default server pipeline thus allowing web and native clients to connect to BFT-SMaRt at the same time and to be fully inter-operable. We also added functionality for parsing JSON data.

For authentication we rely on the same cryptographic primitives and protocols as BFT-SMaRt does, e.g MACs are generated with a SHA-1 HMAC function and on client side, we use a JavaScript crypto library called CryptoJS[4]. We generate and validate MACs like in BFT-SMaRt.

3.2 Bootstrapping the Web Application

Bootstrapping the web applications means that the web browser needs to learn about the correct replica set of the BFT service, retrieve the correct client code, and execute it. Thus, the browser must be able to discover the replica set in a trustworthy way and the client code must be delivered in a way that guarantees that the browser can trust or validate its integrity.

[4] See https://github.com/brix/crypto-js

The discovery of the replica set can be implemented using a BFT directory service. Basically, we could use an existing infrastructure such as the Domain Name Service (DNS). However, the DNS is not a BFT service. The issue of creating a secure BFT-DNS has already been addressed in several academic work [1, 4, 21]. For example, Yang [21] presents a BFT-DNS based on the PBFT algorithm [5], in which there are at least $3f_{DNS} + 1$ replicated name servers for every BFT-DNS zone to provide correct service even if up to f_{DNS} name servers become faulty. Using a BFT-DNS for discovering the replica set is a good solution as we avoid single point of failures or can even tolerate up to a specific number of f_{DNS} compromised name servers. It is noteworthy that since the BFT-DNS would be a distinct service, even if we guarantee to tolerate the number of f faulty replicas for our web service, we must remember that we can still only tolerate f_{DNS} faulty DNS name servers.

The next problem is how to deliver the client code to the browser in a trustworthy way in case that f replicas are faulty (malicious replicas): A malicious replica could deliver tampered client code that does not correctly interact with the BFT service. There are multiple solutions for delivering the client code in a trustworthy way or to check its integrity. We want to avoid the simple approach of using a trusted third party (which may represent a single point of failure) to deliver the correct client code. Thus, we take a different approach: The client code is delivered by a randomly chosen replica. If that replica is unavailable, the browser randomly chooses the next replica from the replica set (we have a list of the replicas set's IP addresses as response from a DNS query). At the same time, the browser sends a request to all other replicas to obtain the hash value of the client code. This way, we can check the client code with hash values from other replicas. Only if a BFT quorum is reached, the browser will accept the client code as trustworthy and execute it.

As the described behavior (see Figure 2) is not supported by a standard web browser, users need to install a browser plugin (extension) that adds such capability. Also, we define a new protocol handler (e.g. web-bft://app.com) so that the browser extension will only apply for BFT web apps. However, the drawback of this is that users need to install a plugin in their browsers, so we basically loose a degree of platform independence. Note that in case we restrict the fault model to non-malicious faults, we can directly use our architecture for transparently accessing a replicated service without need for a browser plugin.

3.3 Client and Replica Authentication

The client always needs to authenticate replicas, otherwise a malicious replica could impersonate other replicas and easily break the $f < n/3$ assumption. In BFT-SMaRt, the service developers have to choose between two ways of authentication: Using MACs or using signatures. If the system is configured to use MACs for authentication, BFT-SMaRt performs distinctly faster than it would with signatures because the symmetric cryptographic operations (computing the HMAC of a message) are faster than the asymmetric ones (computing and verifying signatures of a message). Also, in the system model of BFT-SMaRt, it is

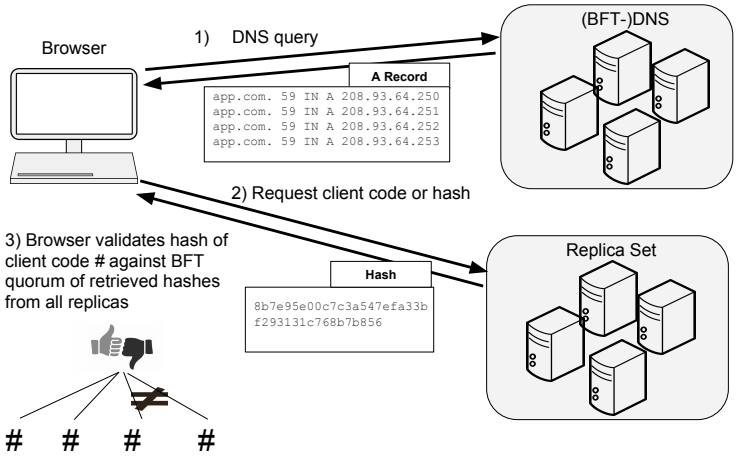


Fig. 2. Approach for bootstrapping the application

assumed that all keys are known beforehand by all system participants. However, in a productive system, the problem of sharing and distributing keys is an issue that must be addressed. Thus, we sketch our approach for authentication as follows: Every service replica is equipped with a certificate that evidences that the replica with its specific ID and internet address is a legitimate replica of the service. These certificates are signed by some certificate authority, so that a certificate chain to a root certificate installed in the user's browser exists. We can then use web protocols on top of TLS such as HTTPS or secure websockets to establish reliable and authenticated channels between browsers and replicas to prevent man-in-the-middle attacks and sniffing. However, while with TLS we can successfully secure the transport of our messages, we still need an authentication scheme for the BFT protocol layer that is above the TLS layer, which is why we implemented authentication using MACs as it is done in BFT-SMaRt.

3.4 Publish-Subscribe Model

We also provide full functionality for a publish-subscribe model that the application layer can use. However, the use of websockets is for a bi-directional channel required in that case, as we want to avoid polling over HTTP due to its high load on the replicas.

Our web client interface will provide asynchronous ordered *invoke()* methods that are used solely for subscribe and unsubscribe purpose, e.g. *invokeSubscribe(req, replyListener, event)*. This methods expects an additional *event* for which the client wants to register (e.g. a document change). The *replyListener* object is stored on the client side until the *unsubscribe* method is called and is bound to the specific *event*. If the client validates a quorum of matching replies (with server side generated sequence numbers) that contain the *event*, then it

will pass the reply to the respective registered *replyListener*. Since subscriptions change the state of the server, an ordered invoke method is necessary.

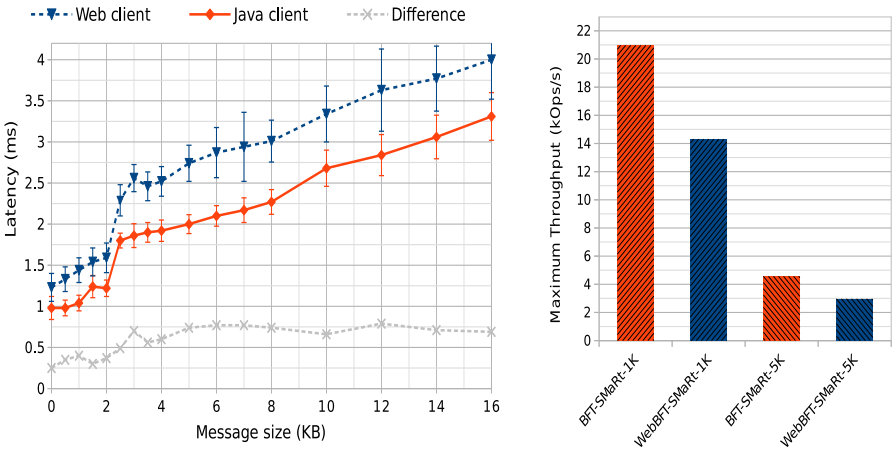
On the server-side we will process the registration like any ordered request in the *executeSingle(byte[] command)* method (the server parses the command to obtain information about what type of command it is e.g. a subscription). Accordingly, the replica adds the *client id* to a list of subscribed clients that it maintains for every *event*. This list is an ordinary part of the state and needs to be respected when transferring state / installing from a snapshot by the state machines. Overriding BFT-SMaRt's default *Replier* allows us to build a customized reply management. This is handled in the method *manageReply(req, messageContext)* in which we can - depending on the event of a request - call the communication system's *send* method passing the *reply* that is associated with the request and the respective list of *subscribers* for an event. As we can define the behaviour depending on the request, it is also possible to only answer the requesting client (e.g. for read-only requests, subscriptions or unsubscriptions).

This allows the replicated server to publish state updates to all subscribers, thus facilitating the implementation of our real-time, interactive web application.

3.5 A BFT Interactive Group Web Text Editor

Our implementation has the purpose to study a use case that is close to a real world scenario. The group editing service is built by implementing the server interfaces of BFT-SMaRt as well as employing our web client interface so it can be accessed in the browser. We use our described publish-subscribe mechanism for publishing document changes to all connected clients and we also use a *Diff, Match and Patch* library that implements robust algorithms to perform operations required for synchronizing plain text [16, 10] in combination with *Differential Synchronization* [9] a method for keeping documents synchronized between server and clients. When a client invokes a write command (which contains the result of a diff operation, thus a list of changes) and the server computes and applies a patch to its document version (ergo a state change occurs), all clients that subscribed for such an event are being notified. The server distributes a list of changes to all subscribed clients (customized reply management).

Our solution differs from existing academic work regarding BFT collaborative editing [23] which uses operational transformation [19] in combination with a set of BFT mechanisms, e.g. solutions for state synchronization and electing a new leader that consider the Byzantine quorum. However, the main difference is that their system model suggests that a user has the role of both publisher and participant (meaning server and client role). However, in such a system with n users, there must be sufficient redundancy to tolerate f Byzantine faulty nodes. This leads to the strong assumption of requiring $n > 3f$ users to work at the same time on a shared document. The BFT group editor we implemented has a clear separation between client and (replicated) server roles. It does thus not require a minimum number of participating clients.



(a) Latency measured at client side

(b) Maximum throughput measured at server side

Fig. 3. Performance measurements for a simple dummy service with variable size of requests and responses, comparing BFT-SMaRt Java clients with our web clients

4 Evaluation

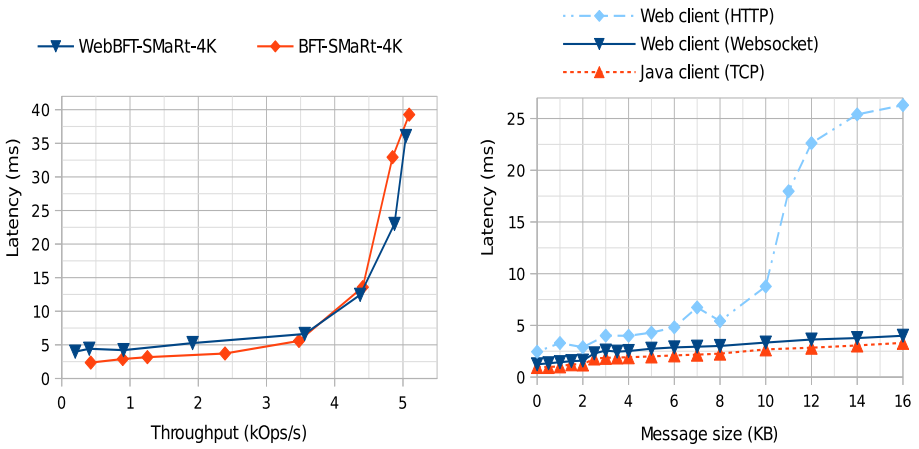
In this section, we evaluate our implementation with throughput and latency micro-benchmarks, using the original BFT-SMaRt system as a baseline for comparison. Furthermore, we provide measurements using a group editing service as real-world application. Our experimental setup consists of a BFT configuration with four replicas ($n = 4$; $f = 1$). All replicas are hosted on different machines. A variable number of clients is distributed across several other machines. All hosts have Intel Core i7-4790 CPUs at 3.60 GHz (4 cores/8 threads), 16 GB of memory, Gentoo Base System release 2.3, Linux kernel 4.9.16 and JRE 1.8.0_121-b13, connected via a Gigabit Ethernet network. We use Firefox 57.0b3 (64-Bit) as browser to launch our web clients.

4.1 Micro-Benchmarks

Our micro-benchmarks use BFT-SMaRt's `ThroughputLatencyServer` implementation as simple service that only returns dummy response for each request, with configurable request and response size. We evaluate our web-based JavaScript client against BFT-SMaRt's `ThroughputLatencyClient` as baseline.

Figure 3(a) shows the results of our latency measurements for message sizes from 0 to 16 KiB (average and standard deviation of latency observed at client side, based on 10000 requests). The latency using the web client is slightly higher than using the Java client, consistently increasing with increasing message sizes.

The additional overhead of our implementation is mainly due to translating JSON requests to Java messages, and we also suspect that the generation



(a) Correlation between latency and throughput for web clients (websocket) and Java clients

(b) Latency comparison between web client (HTTP), web client (websocket) and Java client for different message sizes

Fig. 4. Further performance measurement results of the micro-benchmarks

and validation of HMACs with the SHA1 function is implemented less efficient in JavaScript than in Java. Still, a latency increase of only 0.58 ms on average should be acceptable for BFT web clients. Figure 3(b) shows the results of our throughput measurements, using as many clients as needed to reach the maximum throughput, for 1 KiB and 5 KiB message sizes. WebBFT-SMaRt's maximum throughput is 31.8% (1 KiB) and 35.1% (5 KiB) less than that of the original BFT-SMaRt. One reason for this decrease is that while BFT-SMaRt directly exchanges byte array messages, WebBFT-SMaRt sends requests in JSON format, which need to be transformed to a byte array at server side.

In another experiment (see Figure 4(a)), we measure the correlation between latency and throughput. By increasing the number of clients, we can increase the total throughput until we reach a maximum. At the same time, adding more clients increases the per-request latency. In Figure 4(b), we use both the HTTP and the websocket variant of our web clients. The HTTP web client's latency is distinctly higher than the websocket client's for all message sizes, with larger difference for larger message sizes. The HTTP protocol imposes far more overhead in respect to headers, but to our surprise this overhead even increases with increasing message size. The additional construction of Java objects for encoding/decoding HTTP at the server side is likely to contribute to this increase. We conclude that the web clients should be used with websockets for better performance. Websockets also allow the use of a publish-subscribe mechanism over a bi-directional channel, which is simply not possible with HTTP.

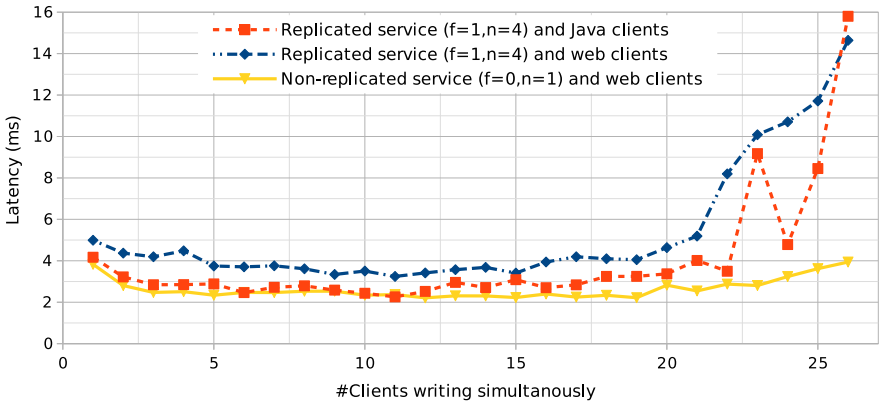


Fig. 5. Latency comparison between non-replicated group editing service and replicated services with Java and web clients, when clients write simultaneously on the document

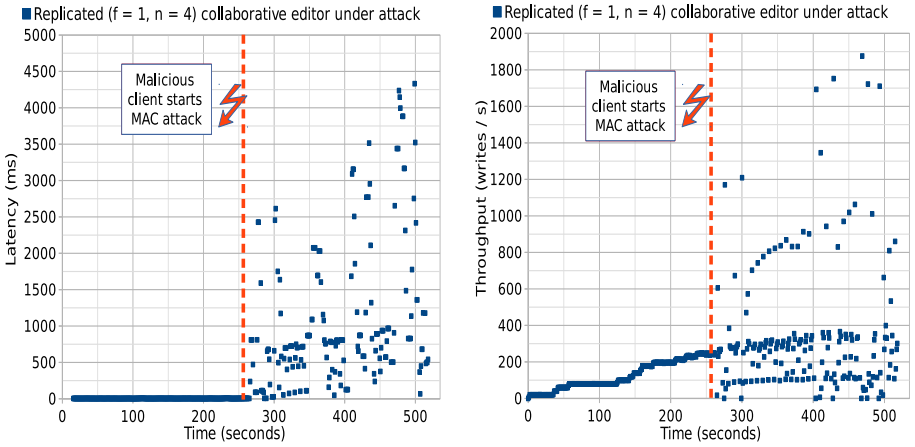
4.2 Real-World Use Case

In this section, we evaluate our replication framework with a group editing service, using the following experiment with $n = 4$ replicas and clients on several machines: Every replica runs the service implementation of our group editing service and an HTTP server to deliver the client code. Clients are successively started every 15 seconds on machines in the same LAN by a script and automatically write on the document.

We launch either only web clients or only Java clients. The clients will send only write requests, and our publish-subscribe mechanism will notify clients about document changes automatically. A write request will with 51 % probability be a change generated by inserting an arbitrary character to the document and with 49 % be a change that deletes an arbitrary character from the document, yielding a slowly growing document. The interval between two requests is set to 50 ms, thus a single client sends up to 20 requests per second.

Latency is measured on the client side as the average of the last 100 samples of latencies for received responses to write requests. We measure the average throughput of a specific time interval as the number of write requests that were executed within this interval at the leader replica. Besides comparing BFT-SMaRt with our WebBFT version, we also add a non-replicated configuration with web clients to the comparison to investigate the performance overhead that replication brings to such real-time collaborative services.

Figure 5 shows the results of these experiments. Web clients have overall a slightly higher latency than Java clients, which is congruent with the observations we made from the micro-benchmarks. Compared to the performance of the non-replicated group editing service, there is also a small increase in latency, which is to be expected, since the client needs to wait for several responses to fulfill a quorum.



(a) Impact of the attack in terms of request latency observed at the client side

(b) Impact of the attack in terms of throughput measured at replica 0

Fig. 6. Malicious client attacks the group editing service

4.3 Malicious Clients

Malicious clients can attack the system by sending requests with corrupt MACs that can be authenticated only by the leader replica. When the leader orders the request, the other replicas are unable to authenticate the request and will eventually consider the leader to be faulty and elect a new one [8]. Similarly, non-leader replicas may receive requests with MACs that are not authenticated by the leader. In this case, the leader will not propose this request, and other replicas that authenticate their MAC will suspect a fault leader after some timeout [14] and invoke the leader-change sub-protocol [14]. Leader changes can lead to a drastic throughput decrease [8] even if only a single malicious client is attacking.

We investigate on how much damage a malicious client can actually cause using our group editing service and a $f = 1, n = 4$ replication configuration and web clients that are automatically writing on a document. We increase the number of web clients over time (a new client connects to the service every 15 seconds), thus increasing the throughput of the system. At a specific time (roughly at $Time = 250$ seconds) we start a malicious client that carries out the MAC attack. Every point in Figure 6(b) expresses the average throughput of the system measured at replica 0 within the last second, e.g. at $Time = k$ seconds the average throughput of interval $[k-1, k]$ is plotted. The latency for second k is measured at the client side as the average latency for the last 50 responses it has received at second k. Figure 6 shows our results.

The throughput of the system scatters. We observe time intervals where the throughput drops drastically, reaching almost zero. However, client requests that are being piled up in the queue lead to a temporary higher throughput once a new leader is elected followed by an interval in which no leader change

occurs and the throughput is as expected (recovered phase). We observe forced leader changes periodically roughly every two seconds (this corresponds to the configured timeout). The latency observed at a client-side increases in our LAN from the range of a few milliseconds up to the range of a few seconds when the service is under attack. Thus, we conclude that malicious clients are indeed a problem both for throughput and latency of a replicated service. Especially high client request latencies that are caused by a leader change tarnish real-time interactive applications such as our group editor.

5 Related Work

The advancement of practical, high-performance frameworks for BFT SMR applications has been studied in several works e.g. by the development of PBFT [5], UpRight [7] or BFT-SMaRt [2]. BFT-SMaRt made some important improvements compared to the preceding SMR libraries as it supports reconfiguration, relies on a modular architecture and uses multi-threading for the processing of messages and cryptographic signatures or MACs.

In the last decade there was also much work that focuses on increasing the reliability of web services, e.g. a web service based N-Version model called WS-FTM (Web Service-Fault Tolerance Mechanism) [13]. WS-FTM was a simple approach for implementing the classic N-Version model for web services by equipping the client with a transparent replication layer to eliminate physical faults or software implementation related faults [13]. It includes a simple consensus voter which compares the results in order to increase the reliability of the system.

Another approach was Thema [15], a BFT middleware for web service applications. Thema supports the multi tiered requirements of web services (e.g. the client may connect to a web service which itself depends on several other web services, therefore not fulfilling the classical client-server schema) and standardizes the support of web services for their clients by providing a WSDL interface and SOAP communication [15].

Moreover, building web services that have a high degree of security and dependability was studied by BFT-WS (BFT framework for Web Services), which operates on top of the standard SOAP messaging framework [22] and its mechanism is based on PBFT [5]. The authors claim that the performance of BFT-WS has only a moderate moderate runtime overhead. Like Thema, BFT-WS uses web services technologies that are antiquated.[5] The use of modern web technologies like websockets allows us to implement a publish-subscribe mechanism and thus adapt to the requirements of real-time collaborative web applications. For example, our web group editor (which notifies clients when state changes occur) can not be implemented with BFT-WS or Thema, because of the lacking support for websockets. We conclude that any new implementation approach of

[5] BFT-WS uses the Apache Axis2 framework for Java. However, state-of-the art web clients use JavaScript instead of Java in order to make them platform independent. Also, the underlying replication library PBFT is not maintained anymore and a variety of replication libraries with better performance exist, e.g. BFT-SMaRt [2].

a BFT web service should consider the state-of-the art technologies for building web services and rely on a BFT SMR framework that is still being maintained.

Other notable approaches include CloudBFT [17] that presents a BFT architecture that aims for scaling services in a cloud environment by exploring the possibility of grouping virtual machines into physical machines, thus offering elasticity. Moreover, Wehrman et al. proposed extensions to PBFT to support replicated clients, designed for long-running distributed applications in which replicated clients invoke operations on replicated servers [20].

6 Conclusions

We investigated on challenges and solutions for BFT web services by extending existing BFT SMR frameworks in respect to web applications. Our web client interface can be used to build any application logic on top of it by using the provided *invoke* methods to multicast a request to be executed by all replicas. It also supports authentication with MACs and reconfiguration as native clients do in BFT-SMaRt. Our solution for bootstrapping the web application explains how the replica set can be discovered by the use of existing infrastructure, the DNS and how delivered client code can be validated.

Experiments with micro-benchmarks show that our web client implementation achieves performance comparable fast as BFT-SMaRt's interface for native clients in terms of latency and throughput. Moreover, we concluded that the web clients should be used with websockets instead of HTTP as underlying protocol for significant better performance and also because websockets allow the use of a bi-directional channel between client and replicated server which is essential for observing state changes in real-time.

Our BFT group editor diverges from the traditional SMR programming model (a request-response cycle) by incorporating a publish-subscribe mechanism into BFT-SMaRt. We use asynchronous invoke methods on the client-side, websockets for bi-directional communication and a customized reply management on the server-side. The evaluation results of our group editor show that the replication of a web service with BFT-SMaRt is fast enough to match the requirements of real-time collaborative web applications.

Acknowledgment

This research was supported by DFG through project OptScore.

References

1. B. Awerbuch and C. Scheideler. Group spreading: A protocol for provably secure distributed name service. In *ICALP*, volume 3142, pages 183–195. Springer, 2004.
2. A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

3. A. N. Bessani and E. Alchieri. A guided tour on the theory and practice of state machine replication. In *Tutorial at the 32nd Brazilian symposium on computer networks and distributed systems*, 2014.
4. C. Cachin and A. Samar. Secure distributed DNS. In *Dependable Systems and Networks, 2004 International Conference on*, pages 423–432. IEEE, 2004.
5. M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.
6. A. Charland and B. Leroux. Mobile application development: Web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.
7. A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proc. of ACM SOSP*, pages 277–290. ACM, 2009.
8. A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The Time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 13. ACM, 2008.
9. N. Fraser. Differential synchronization. In *Proceedings of the 9th ACM Symposium on Document Engineering*, pages 13–20. ACM, 2009.
10. N. Fraser. Diff, match and patch libraries for plain text. [online], `https://code.google.com/archive/p/google-diff-match-patch/`, (Accessed: 20/09/17), 2012.
11. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
12. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982.
13. N. Looker, M. Munro, and J. Xu. Increasing web service dependability through consensus voting. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 2, pages 66–69. IEEE, 2005.
14. R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo. Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 41–61. Springer, 2013.
15. M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *24th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 131–140. IEEE, 2005.
16. E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
17. R. Nogueira, F. Araujo, and R. Barbosa. CloudBFT: elastic Byzantine fault tolerance. In *IEEE PRDC*, pages 180–189. IEEE, 2014.
18. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
19. D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 437–446. ACM, 2004.
20. I. Wehrman, S. L. Pallemulle, and K. J. Goldman. Extending byzantine fault tolerance to replicated clients. Technical Report WUCSE-2006-7, Washington University, 2006.
21. Z. Yang. *Using a Byzantine-fault-tolerant algorithm to provide a secure DNS*. PhD thesis, Massachusetts Institute of Technology, 1999.
22. W. Zhao. BFT-WS: a Byzantine fault tolerance framework for web services. In *11. Int. IEEE EDOC Conference Workshop (EDOC'07)*, pages 89–96. IEEE, 2007.
23. W. Zhao and M. Babi. Byzantine fault tolerant collaborative editing. *IET Int. Conf. on Information and Communications Technologies (IETICT 2013)*, 2013.