Themis: An Efficient and Memory-Safe BFT Framework in Rust

Research Statement

Signe Rüsch
TU Braunschweig, Germany
ruesch@ibr.cs.tu-bs.de

Kai Bleeke
TU Braunschweig, Germany
bleeke@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

Abstract

Byzantine fault tolerant (BFT) protocols have previously been developed mainly in C or Java. C offers high performance but is more error-prone, leading to more potential Byzantine faults, whereas Java offers memory safety at the cost of performance. The Rust programming language combines the performance advantages of C with safe memory management, and newer releases now enable the implementation of complex, non-blocking asynchronous frameworks, as is needed for efficient BFT frameworks. We present a BFT framework implementation in Rust and preliminary performance evaluations for the PBFT protocol.

Keywords Rust, blockchain, Byzantine fault tolerance

1 Introduction

Byzantine fault-tolerant (BFT) protocols are a group of distributed consensus protocols that allow a certain number of nodes in the system to fail in arbitrary ways. Due to higher overhead compared to crash-fault consensus protocols, the implementation of a BFT protocol aims to be highly optimized regarding processing time per message, on the protocol as well as the network layer, while still ensuring the correctness of the implementation, as software errors are a possible source of Byzantine faults.

In the beginning, BFT frameworks have been implemented in C for high performance, already using non-blocking communication for efficiency [3]. C's performance is partially due to the raw memory access; however, it is the programmer's responsibility to correctly handle this, and C's weak type system makes implementations prone to memory leaks and undefined behaviour. A language includes *undefined behaviour* if it allows code whose behaviour is not defined in the language's specification [8], which can lead to program crashes or execution using wrong data. Code that invokes undefined behaviour is considered unreliable, and should be eliminated. C exhibits multiple cases of undefined behaviour centred around its weak type system and pointer arithmetic, e. g. accessing out-of-bounds memory or use-after-free bugs.

Later, Java offered faster development, platform independence, and higher safety due to its strong type system. The majority of today's BFT frameworks are therefore implemented in Java [1, 2, 4], and many high-performance frameworks also utilize non-blocking communication. By disallowing direct access to memory and ensuring that the programmer can only work with references to valid objects, memory-related undefined behaviour is eliminated. The garbage collector (GC) of the Java Virtual Machine (JVM), which frees no longer referenced objects, eliminates the most common source of memory leaks. But interpreting bytecode is slower than executing native instructions, despite optimizations such as just-in-time (JIT) compilation; worse, a JIT compiler and garbage collector add uncertainty to performance as the programmer cannot predict their behaviour [7]. It is also not resource-efficient, as the JVM is known for its high memory consumption [6]. Thus, the choice is between high performance but error-proneness, and slower but safer execution.

Recently, the *Rust* programming language has emerged, which offers both performance and safety [5]. It is a strongly typed language that does not include a runtime or GC and does not feature undefined behaviour, making it both safe and fast. A whole class of potential causes of Byzantine failures due to undefined behaviour are therefore eliminated, and after recent releases Rust now offers the required features for implementing efficient, asynchronous BFT frameworks. As Rust is much more resource-efficient than Java, it can open up new domains for BFT deployment, e. g. in blockchains and embedded systems, making re-implementations even more desirable. We describe the BFT framework Themis and a preliminary performance evaluation.

SERIAL '19, December 9–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7029-5/19/12…$15.00
https://doi.org/10.1145/3366611.3368144

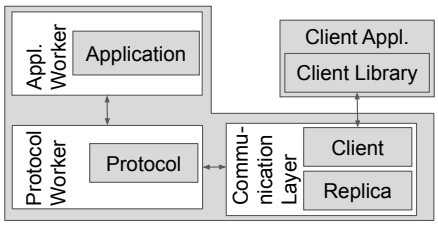

Figure 1. Themis framework components.

2 Themis: A BFT Framework in Rust

Rust is intended for safe and concurrent systems programming. It is considered safe as it eliminates undefined behaviour but still allows for direct memory management without using a GC. Rust introduces *ownership*: each value of Rust has exactly one variable as its owner, and when the owning variable goes out of scope, it is dropped and deallocated. Rust assigns values by *moving* them instead of copying, thus preventing memory leaks. Owned values can be *borrowed*, creating a reference to the value. References are *mutable* or *shared*, with only mutable references allowing mutation. At any given time there may be either a single mutable reference to a value or multiple shared ones. Rust prevents dangling pointers and use-after-free errors by enforcing the *lifetimes* of references. From the ownership, Rust knows the scope in which a value exists and a reference to that value must not outlive that scope. All these rules are enforced at compile time, allowing for safe implementations.

Our framework is designed to be modular and easily extensible for different protocols. It is a fully featured re-implementation of PBFT, including the *ordering*, *checkpointing*, and *view change* subprotocols, and uses asymmetric cryptography for message authentication. It is event-driven and utilizes non-blocking channels, i. e. asynchronous message queues, for communication. It consists of three modules, shown in Figure 1: communication, protocol, and application. For the implementation, we use the `futures` and `tokio` crates, offering newly released and stabilized `async/await` features. The message-oriented *communication* layer handles connection management for both replicas and clients. Here, messages are verified and batched before they even enter the protocol stage; currently, RSA is used for message authentication. The *protocol* layer implements the actual BFT protocol as a *trait*, i. e. an interface, which handles the received messages from communication and application layer, and is implemented for the PBFT protocol including ordering, checkpointing, and view change. The *application* layer trait sends and receives requests and replies, and creates snapshots for checkpointing and failure recovery. A client library sends BFT requests to the replicas.

Preliminary Evaluation. We run four replicas and multiple clients on servers with Intel i7-6700 CPUs. We use 2048bit RSA for message authentication, and our current evaluation compares single-threaded execution. The framework is compared to a Java-based PBFT implementation in the *Reptor* framework [1] running on one pillar, i. e. one non-parallelized single instance. Due to the computationally expensive RSA scheme, it is expected that the throughput will be limited by the rate at which signatures can be created.

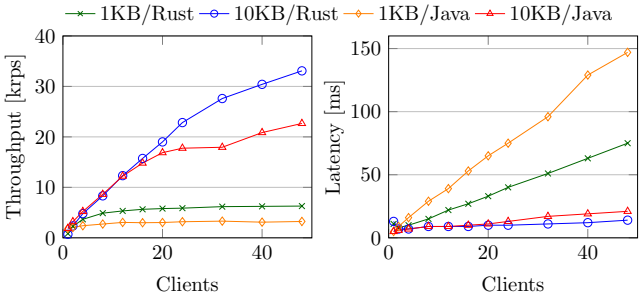

Figure 2. Comparison of Rust- and Java-based PBFT.

We use messages of 100B and allow the frameworks to batch 10 and 100 requests into 1KB and 10KB messages, respectively. The results for throughput and latency can be seen in Figure 2. The overall throughput of the Rust framework can be observed to be significantly higher by up to 46 %, with up to 33 % lower latency. The throughput is primarily bound by the number of signatures. For messages of 100B and batches of 100 messages, the Rust implementation requires 15.1 KB of memory, whereas Reptor requires 1.7GB. The lack of a runtime for Rust results in a memory usage orders of magnitude lower than the Java implementation.

3 Future Work

This overview shows potential for future work: (i) Can the Rust BFT framework be employed on embedded devices with restricted memory capacity? (ii) Can the BFT framework running on embedded devices be used for consensus in an (embedded) blockchain platform? (iii) Can a Rust-based BFT protocol offer comparable performance to a high-performance Java implementation even in multi-core configurations?

References

[1] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-based High Performance BFT. In *EuroSys'17*.
[2] Alysson Bessani, João Sousa, and Eduardo Alchieri. 2013. *State Machine Replication for the Masses with BFT-SMaRt.* Technical Report.
[3] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI'99*.
[4] Allen Clement et al. 2009. UpRight Cluster Services. In *SOSP '09*.
[5] Nicholas Matsakis et al. 2014. The Rust Language. In *HILT '14*.
[6] Oracle. 2019. FAQs About the Java HotSpot VM.
[7] Fridtjof Siebert. 2004. The Impact of Realtime Garbage Collection on Realtime Java Programming. In *ISORC'04*.
[8] Xi Wang et al. 2012. Undefined Behavior: What Happened to my Code?. In *APSYS'12*.