HoneyLedgerBFT: Enabling Byzantine Fault Tolerance for the Hyperledger platform

Haithem Turki
Stanford University
hturki@stanford.edu

Fidel Salgado
Stanford University
fidels@stanford.edu

Juan Manuel Camacho
Stanford University
jcamach2@stanford.edu

Abstract

Hyperledger Fabric is a permissioned blockchain framework implementation designed for enterprise applications with a focus on extensibility and modularity. In particular, it promises support for multiple ordering services responsible for determining the sequence of blocks in the blockchain. However, as of December 2017, the framework does not ship with any Byzantine fault-tolerant service. In this paper, we describe a BFT ordering service we implemented on top of the HoneyBadgerBFT protocol [8], and evaluate its performance. Our results show that our naive initial implementation exhibits a reasonable amount linear overhead with significant room for improvement.

1. Introduction

Hyperledger Fabric is an open-source permissioned blockchain framework initially designed by Digital Asset and IBM and now hosted under the umbrella of Hyperledger projects overseen by The Linux Foundation. It targets business applications and holds flexibility and pluggability as key design concerns, supporting a variety of smart contracts called ”chaincode” and allowing for multiple implementations of various components such as the membership and consensus services (also known as the ”ordering service”). However, as of December 2017, the mainline Fabric releases ship with a single non-Byzantine fault tolerant option based on Apache Kafka (a previous PBFT implementation having been deprecated and reverted ahead of the 1.0 release [4]).

In this paper, we describe our efforts in designing a BFT ordering service implemented on top of the HoneyBadgerBFT protocol [8]. Our preliminary evaluations, both on a local cluster and on a geographically distributed multi-node setup, show that our setup grants the benefits of byzantine fault tolerance while incurring a roughly linear overhead (which we posit could be improved even further in future work). The source code for our service is freely available on the Internet [1] [2].

We organize the rest of this paper as follows. We start by describing the fundamentals of blockchain technology and the Fabric architecture in Section 2, and overview the HoneyBadgerBFT protocol in Section 3. After that, we present the HoneyLedgerBFT ordering service in section 4, bring up related work in section 5, and discuss our initial evaluation in section 6. We then conclude with some proposed future work in Section 7.

2. Hyperledger Fabric

2.1. Blockchain fundamentals

A blockchain is a distributed ledger comprised of a continuously growing list of records called blocks stored in a defined order. Each block contains a set of transactions between the different members of the network.

They are protected from tampering by cryptographic hashes and a consensus mechanism - each block contains the hash of the previous block in the chain, and the consensus mechanism decides which new block to append to the ledger.

Fabric then models the latest state of the blockchain as a versioned key/value store (KVS), where keys are names and values are arbitrary blobs. These entries are manipulated by the chaincodes (applications) running on the blockchain through put and get KVS-operations.

Blockchains may either be permissioned (such as in Fabric) or permissionless. Permissionless ledgers are maintained in a totally decentralized and anonymous manner, and generally rely on consensus mechanisms such as Proof-of-Work to limit their vulnerability to Sybil attacks.

2.2. Fabric Architecture

Hyperledger Fabric nodes reach consensus by performing two separate activities:

1. Ordering of transactions
2. Validation of transactions

When transactions are received from a client application, an ordering service determines their recorded order. The ordering service can be implemented in a variety of ways: ranging from a solo service which can be used in development and testing, to distributed protocols such as Kafka (the option currently shipped by default). To enable the confidentiality of the transactions, the ordering service may be agnostic to the transaction, ie: the transaction content can be hashed or encrypted.

The ordering service has discretion as to the specific time limit or number of transactions in emitted blocks. Most of the time, for efficiency reasons, instead of outputting individual transactions, the ordering service will opt to group multiple transactions into a single block. Once the order is determined, consensus then depends on the smart contract "chaincode" layer which contains the business logic needed to validate transactions. The smart contract layer validates each transaction by ensuring they conform to policy and the contract specified for the transaction. Invalid transactions are rejected and potentially dropped from inclusion within a block [3].

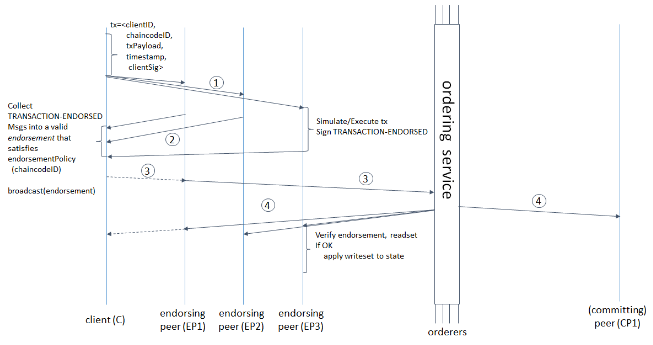

Figure 1. Common Case Transaction Flow in Fabric [5]

3. HoneyBadgerBFT

HoneyBadgerBFT [8] is a byzantine fault tolerant atomic broadcast protocol that serves as the foundation of our ordering service.

In contrast to most other BFT protocols in the space such as PBFT that rely on weak synchrony assumptions, HoneyBadgerBFT aims to provide optimal asymptotic efficiency in asynchronous settings. This makes it especially relevant to the blockchain space and the adversarial environments and unreliable networks that cryptocurrencies need to contend with (the authors in fact list permissioned blockchains as a primary potential target use-case for their protocol. In particular, the protocol allows for relatively high throughput (especially relative to Bitcoin's rate of ten transactions per second), as well as avoiding the need to worry about tuning timeouts as in PBFT. The protocol also relies on an asynchronous common subset protocol (ACS) that combined with a threshold encryption scheme makes it resilience to censorship (ie: a malicious actor cannot selectively identify transactions to block before they're committed).

4. Ordering Service Architecture

Our architecture is illustrated in Figure 2. We launch a basic HoneyBadgerBFT setup and then superimpose a shim layer that implements the HyperLedger ordering service API and which delegates messages to the underlying cluster via socket communication. As transactions are committed via the standard HoneyBadgerBFT round procedures, the cluster then sends update messages back to the orderer shim which then commits the updates into the blockchain.

A common case transaction flow is also shown in Figure 2. First, the Hyperledger client sends a transaction to the order node. Then the order node sends the transaction to a node in the HoneyBadgerBFT cluster who broadcasts the transaction to all other nodes and reaches consensus. When consensus is reach, step 3 occurs, which sends the response back to the order node. Finally, in step 4, the order node validates a block of transaction and sends the response to the Hyperledger client.

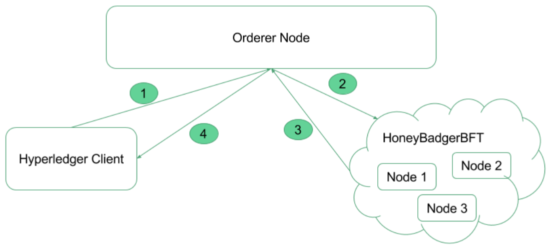

Figure 2. HoneyLedgerBFT Architecture

5. Related Work

Another effort to implement a Byzantine fault tolerant ordering service was recently performed by Sousa et al [7] who based their solution on the BFT-Smart library [6].

We take significant inspiration from their work both in terms of our solution and our evaluation metrics, although we note that BFT-Smart also makes weak synchrony assumptions that break down in the face of timing attacks in the same manner as PBFT, and posit that HoneyBadgerBFT remains an ideal BFT consensus protocol for blockchain applications per the reasons detailed in Section 3. Their solution also involves having the underlying consensus protocol take a direct dependency on the Hyperledger Fabric SDK (and forces it to evolve in lockstep) - in contrast the consensus protocol in our solution treats the messages passed to it as binary blobs and requires no knowledge of its specific contents.

6. Evaluation

We evaluated our ordering service using a four node cluster (with fault tolerance f=1) with a single machine setup. Our primary aim was to quantify the overhead that our solution entailed, both in absolute terms and relative to the reference solo implementation.

We started by comparing the broadcast latency (ie: how long it takes the messages to reach the HoneyBadgerBFT nodes) to the overall latency (ie: including the latency needed for the nodes to reach consensus). We note that in both cases latency scales linearly with the number of messages, which is somewhat expected given that the Honey-

BadgerBFT protocol optimizes for throughput and not latency. The solo implementation also exhibits similar characteristics to the broadast latency.

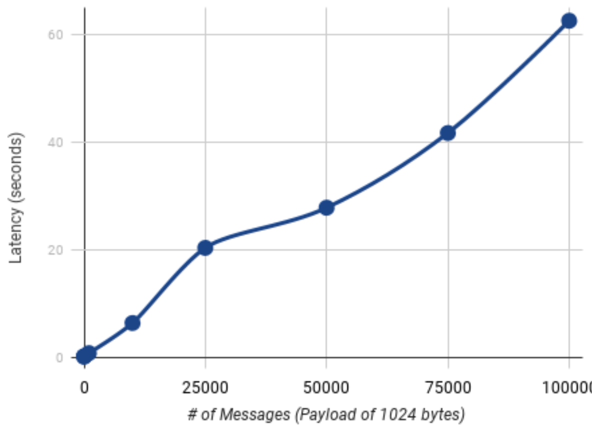

Figure 3. Average broadcast latency with respect to number of transmitted messages.

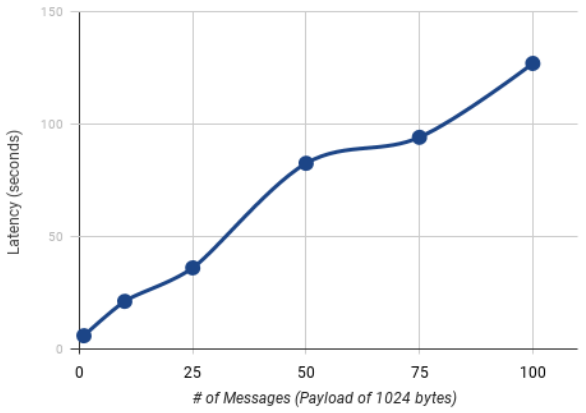

Figure 4. Overall latency with respect to number of transmitted messages

We also measure throughput per thread and note that it also scales linearly as a function of the envelope size.

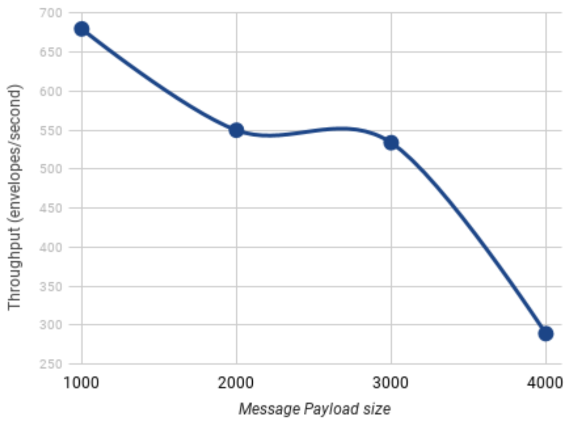

Figure 5. Average ordering throughput per thread with respect to envelope size

7. Future Work

7.1. Evaluation across different geographical regions

Using the limited compute resources at our disposal, we attempted to evaluate our ordering service with against a HoneyBadgerBFT cluster spanning across four continents by deploying nodes in Oregon, London, Tokyo, and Sydney. We successfully managed to have the nodes communicate with each other, however we ran into a last minute correctness issue that invalidated our benchmarks. Reevaluating its performance would be a focus of future work.

7.2. Comparison with other baselines

During most of our evaluation we compared our solution to the single-node ”solo” orderer implementation, both for ease of deployability and to compare against the most absolute lightweight and low-overhead implementation as possible. Comparing against Kafka would likely be a better indicator of conditions one could reasonably expect in production, and comparing against the BFT-Smart implementation would also allow for a more apples-to-apples comparison of two byzantine fault tolerant protocols. It would be interesting in particular to see how they handle various network quality issues.

7.3. Implementation improvements

Many parts of our implementation were left deliberately simple, ie: using one ordering thread by default, using inefficient serialization protocols when passing messages, etc. With a relatively minimal amount of further work, we'd expect to see even better performance.

8. Acknowledgements

We're grateful to Joao Sousa, Alysson Bessani, Marko Vukolic for insipiring our architecture with their BFT-Smart integration, and to Andrew Miller for getting us started against the HoneyBadgerBFT codebase and pointing us towards the relevant building blocks.

References

[1] Honeyledgerbft-compatible branch of fabric. https://github.com/hturki/fabric.

[2] Honeyledgerbft-compatible branch of honeybadgerbft. https://github.com/hturki/HoneyBadgerBFT.

[3] Hyperledger architecture whitepaper. https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

[4] Reverted pbft changeset in hyperledger repository. `https://gerrit.hyperledger.org/r/#/c/1467`.

[5] Common-case transaction flow in fabric. `https://hyperledger-fabric.readthedocs.io/en/release/_images/flow-4.png`, 2017.

[6] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart, June 2014.

[7] M. V. Joo Sousa, Alysson Bessani. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. `https://arxiv.org/abs/1709.06921`, 2017.

[8] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. `http://doi.acm.org/10.1145/2976749.2978399`, 2016.