A blockchain-based digital wallet application with HoneyBadger-BFT consensus algorithm

Haoyang Zhang
Stanford University
Stanford, USA
hz5@stanford.edu

Zhuoyi Huang
Stanford University
Stanford, USA
zhuoyih@stanford.edu

Rui Li
Stanford University
Stanford, USA
ruiliup@stanford.edu

Yanfei Xiong
Stanford University
Stanford, USA
yanfeix@stanford.edu

Abstract

HoneyBadger-BFT [4] is a consensus algorithm for blockchain that can tolerate Byzantine fault in an asynchronous network. However, there has never been a public blockchain that utilizes HoneyBadger. At the same time, public blockchain such as Bitcoin [5] uses Proof-of-Work to solve consensus, which is not power-efficient. In this project, we explore the possibility of HoneyBadger-BFT by building a simple digital wallet applications built on top of a blockchain that utilizes HoneyBadger-BFT. Our system provides a simple user interface for clients to register accounts and submit transactions, all records of which is recorded on the blockchain, which is totally ordered among all replicas in the system.

1. Introduction

Blockchain technology is gaining huge momentum in the past decade. Its decentralized and public nature revolutionized many traditional industries. For instance, cryptocurrency like Bitcoin and Ethereum received profound attention and investment due to certain advantages compared to traditional financial system, while novel applications like Non-Fungible Token start to break new grounds. The core of Blockchain is a Byzantine-Fault tolerant consensus algorithm that solves the replicated log problem in a replicated state machine context. Among all such algorithms, HoneyBadger-BFT [4] is introduced as the first algorithm that is able to guarantee progress even in a completely asynchronous network with no assumptions on network timing. In addition, HoneyBadger-BFT is a more power and computationally efficient algorithm compared to some of these algorithm, like Proof-of-Work used in Bitcoin.

We believe that HoneyBadger-BFT has great potentials in future blockchain applications, so we decided to build a simple blockchain based on HoneyBadger-BFT, on top of which runs a digital wallet application that takes full advantage of the benefit of blockchain. Our application supports client requests including transaction posting and balance reading, and these requests are processed and validated before written to blockchain, after which all transactions are public and are kept in the same order across all replicas in the system.

This paper is organized in the following way. Sec. 2 introduces the background knowledge involved in our system. Sec. 3 describes the high-level design of our system. Sec. 4 describes the implementation details. Sec. 5 discusses the evaluation we have done and the results we have got. And finally, Sec. 6 and Sec. 7 talks about some improvement that can be made and our final conclusion.

2. Background

2.1. Blockchain

Blockchain first gained its popularity in the advent of Bitcoin [5]. The ability to make powerful decentralized applications that is safe and consistent is very attractive. Blockchain, in many context, is just a synonym for replicated log. The core problem that lies in the center of log replication in a distributed system is the consensus algorithm. Traditional blockchain consensus protocol consists of some specific objectives such as coming to an agreement, collaboration, cooperation, equal rights to every node, and mandatory participation of each node in the consensus process. Thus, a consensus algorithm aims at finding a common agreement that is a win for the entire network. There are many different ways for nodes to reach consensus. Bitcoin, for instance, leverages a mechanism called Proof-of-Work [5], which requires nodes to solve an extremely computationally heavy mathematical puzzle in order to reach consensus. Proof of Work(PoW) [3] powered blockchains currently account for more than 90% of the total market capitalization of existing digital cryptocurrencies. While Bitcoin is the most popular cryptocurrency nowadays, this consensus mechanism is considered to be very wasteful in terms of energy and computation. PBFT [2] is another example of a algorithm for replicated log. But since it makes assumptions on network synchrony, an actual deployment of the system using PBFT would not be ideal in a completely asynchronous network. Ever since then, efforts have been made to come up with algorithms that solve these problems. And some great progress has been made, in the form of algorithms like HoneyBadger-BFT [4].

2.2. HoneyBadger-BFT

HoneyBadger-BFT is the first BFT protocol that guarantees liveness in asynchronous network without making timing assumptions. It focuses primarily on taking full utilization of the network bandwidth rather than optimizing for latency. In each round, each node reads in a batch of transactions from its buffer, and send this batch to other nodes. Threshold encryption is used here so that the decryption requires multiple nodes to prevent single-node attack. Next, batches are submitted to Asynchronous Common Subset, which utilizes Binary Agreement and Reliable Broadcast to reach an agreement among all nodes which of these batches to include in the output of this round. Finally, all shares are received and decrypted in the honest nodes that are ready to be shipped. Of course there are assumptions that HoneyBadger-BFT makes, like that reliable communication channels are set up between nodes and even though messages can be delayed indefinitely, they should be eventually delivered [4]. But these assumptions hold up way more easily in asynchronous network compared to timing assumptions made by other protocols. During their experiment with HoneyBadger-BFT, they deployed over 100 nodes across 5 continents, and the transaction throughput results they got scale well across. Therefore, HoneyBadger-BFT seems like a great building block for future blockchain applications.

3. System Design

3.1. Overview

We would like our digital wallet application to provide strong safety and liveness guarantees and a practical throughput. Financial transactions are mission-critical and are typically submitted over a wide area network and among a large number of nodes. In the real world, network connections can be intermittent and unpredictable, and even worse, malicious network adversaries can arbitrarily block or schedule network traffic. This could cause consensus protocols with a weak synchrony assumption to not terminate or take a long time to recover from network partitions. Since HoneyBadger-BFT protocol is an Byzantine Fault Tolerant (BFT) protocol based on asynchronous network assumption, it can guarantee strong safety and a good throughput even in a fully asynchronous network. Therefore, we decided to use the HoneyBadger-BFT protocol as the fundamental building block for our application.

There are two main components in our system: User Service and HoneyBadger-BFT service. Each node deploys an instance of the two services. The system block diagram is shown in Fig. 1.

3.2. User Service

The User Service component is responsible for taking and processing requests from the client applica-

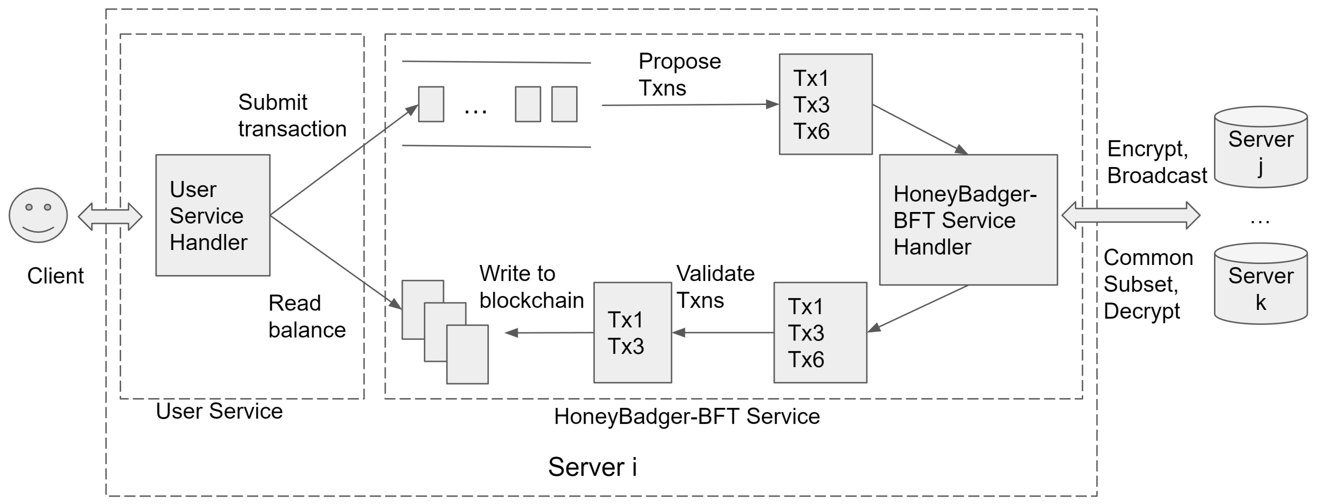

Figure 1. System Block Diagram

API	Function
create_account	Create a new account in blockchain.
create_txn	Create a single transaction in blockchain.
create_txns	Create bulk transactions.
get_balance	Get account balance for a given account id.
get_all_accts	Get all accounts that have been created among all nodes in blockchain.

Table 1. User Service APIs

tion. The types of operations supported are listed in Tab. 1. For account registration requests and transaction submission requests, User Service simply forward them to the HoneyBadger-BFT service. For requests that query balance of a user, User Service would traverse the local copy of the blockchain on the node, read transactions involving the user, calculate the balance and return it to the user.

3.3. HoneyBadger-BFT Service

The HoneyBadger-BFT service takes raw transactions and outputs an identical sequence of transactions on all honest nodes. It processes the sequence of transactions and stores the transactions onto the local file system on each node. These block files each contain 100 transactions as well as the hash of the previous block to make it a logical chain.

Specifically for our application, we need extra procedures to validate the transactions. The original HoneyBadger-BFT only ensures a global ordering of a set of transactions. But in our case, several types of transactions are considered invalid. For instance, if the user does not have enough balance, then two transactions that double spend the remaining balance should be prohibited. Also, transactions made to non-existent accounts should be forbidden. Such validations are done after HoneyBadger has decided on the set of transactions to export in a round and before the transactions are written to the blockchain.

3.4. Communication Across HoneyBadger-BFT Nodes

The HoneyBadger-BFT protocol assumes each pair of nodes is connected by a reliable authenticated point-to-point channel that does not drop messages. Further, it assumes that every message sent between correct nodes must eventually be delivered [4]. To meet this assumption, we choose to use gRPC because it is designed to work with a variety of authentication mechanisms to provide a reliable authenticated point-to-point channel that does not drop messages.

4. Implementation

We used Python to implement our system. User service and HoneyBadger-BFT service run as two separate processes on every node. More implementation details of each component are discussed below.

4.1. Communication over gRPC

We use gRPC framework to provide reliable network channels for all communications between the client application and the server application, as well as the communications between HoneyBadger-BFT service handlers among all of the server nodes. For the demo project implementation, we choose to use insecure communications channels for the ease of debugging. Those configurations can be changed to secure channels fairly easily in gRPC. For the user service communications, we defined one RPC service call for each of the APIs listed in Tab. 1. For the HoneyBadger-BFT service communications, we defined one RPC service call with a request message containing one of the four different operation types — CommonCoinOperation, BinaryAgreementOperation, ReliableBroadcastOperation, ThresholdEncryptionOperation, to accommodate the need from all different stages of the HoneyBadger-BFT protocol.

4.2. User Service Handler

User service handler is implemented as a gRPC server. The user service requests are being sent by clients in the form of a gRPC request. Upon receiving the call, transactions are stored into a Python queue structure, and made ready to be dequeued by the HoneyBadger-BFT service handler in the beginning of each round.

4.3. HoneyBadger-BFT Service Handler

Communication between HondyBadger-BFT server nodes are being made as gRPC requests and responses. In the beginning of each round, each server node queries its user service queue to receive transactions, and broadcast the transactions to all the available HoneyBadger-BFT nodes by gRPC service calls. Once the round of HoneyBadger-BFT terminates, all decrypted transactions are validated and written into the blockchain on each node's local file system. The transactions made out of users who do not have enough balance, or the transactions made to invalid accounts would be marked invalid before written to the blockchain.

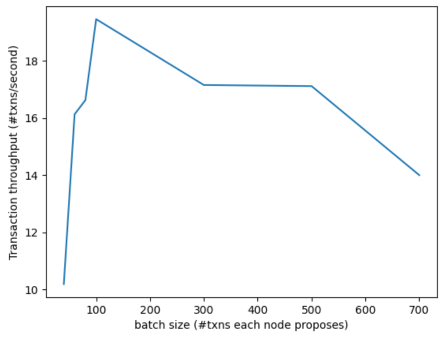

Figure 2. Transaction throughput vs. Batch size

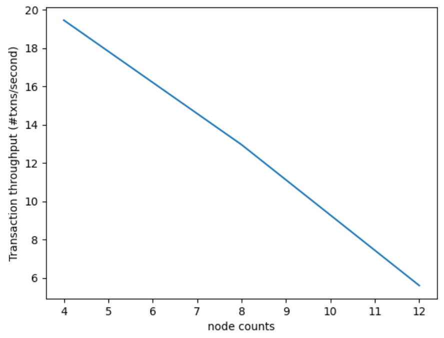

Figure 3. Transaction throughput vs. Node count

For the HoneyBadger-BFT based blockchain implementation, we used a Python HoneyBadger-BFT library [1] written by the authors of the HoneyBadget-BFT paper [4]. An instance of the library running on each node accepts transactions in the form of ASCII strings, and outputs a sequence of transactions that has the same ordering among all instances.

5. Evaluation and Discussion

We deployed our system on AWS EC2 servers and ran some tests and evaluations on it. All machines are EC2 t3.xlarge instances all in the Northern California region, and our system runs in a docker container on each of these machines. We first summarized how many transactions are successfully submitted by honeybadger-BFT, and the results are shown in Tab. 2. The first part is the comparison between different

Batch Size	#Nodes	Rd#1	Rd#2	Rd#3	Rd#4	Rd#F5	Rd#6	Rd#7	Rd#8	Rd#9	Rd#10
40	4	160	160	160	160	160	160	160	160	160	160
60	4	240	240	240	240	240	240	240	240	240	240
80	4	320	320	240	320	240	320	240	320	320	320
100	4	400	400	400	300	400	400	300	400	400	400
300	4	600	900	900	900	900	1200	900	1200	900	1200
500	4	1500	1500	1500	2000	1500	1500	1500	2000	1500	2000
700	4	2800	2100	2100	2100	2100	2100	2100	2100	2800	2100
100	8	800	800	800	800	800	800	700	800	800	800
100	12	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200

Table 2. Number of transactions successfully submitted by honeybadger and written onto blockfile for each round with different batch size and number of nodes configurations. Batch Size means how many transactions can each of the node in the blockchain propose for each round, #Nodes means the total number of nodes in the blockchain. Rd# means different round, and the value under different columns means the number of transactions submitted for each round. If the number of transactions for a round is smaller than $(Batch\_Size * \#Nodes)$, this suggest that not all proposed transactions from the nodes are successfully submitted.

batch size controlled for node number, and the second part is the comparison between different node number while controlled the batch size.

5.1. Transaction Throughput vs. Batch Size

In our first experiment, we tested the relationship between transaction throughput and batch size by using different batch sizes 40, 60, 80, 100, 300, 500, 700 running on 4 nodes. To calculate the transaction throughput, we divided the total number of transactions processed in 10 rounds of HoneyBadger-BFT by the time to run the 10 rounds.

From Figure Fig. 2 we can see that the relationship is not a simple positive correlation like the experiment in [4]. Based on our discussion, the reason behind this behavior is because as batch size increases, at a certain point, the significant improved message size would cause some messages to be delayed, and since in the Asynchronous Common Subset algorithm in HoneyBadger-BFT, only N - f deliveries of value 1 from the Binary Agreements are required, after which 0 would be sent to the Binary Agreements, meaning that batches of transactions proposed by some nodes will be dropped. We marked rounds with dropped batches red in Tab. 2. This would not happen for a small batch size (less than 100), in which case basically all batches are accepted.

Therefore, even though more transactions are submitted to the system, due to the congested network, the actual number of committed transactions does not scale well against the batch size. Added on top the fact that it takes more time to finish a round with larger batch size, the overall decrease of throughput is explainable.

5.2. Transaction Throughput vs. Node Count

In our second experiment, we tested the relationship between transaction throughput and number of nodes deployed. We ran the experiment on node count 4, 8 and 12, all with a batch size of 100. We used the same calculation as the previous experiment to get transaction throughput.

From Figure Fig. 3 we saw a negative correlation between throughput and node count. A further glance into our data showed that even though basically no batches were dropped in the committed transactions, the time took to run one round is significantly higher for higher node counts. Based on the data, we argue that the reason for this behavior is again due to network congestion. As more nodes are added to the system, the number of messages transmitted in HoneyBadger execution is also higher. In this case the stall due to congestion seems to be the major factor that dominated the benefit of having more submitted batches for more nodes.

In both of our experiment, we were getting results that seemed off from the original paper, but the slower performance is expected. Since what we implemented is an actual system that wrapped around the HoneyBadger consensus protocol, there are many other factors that affected the performance. For instance, our user service server constantly receives transactions submitted from the clients and submit them to the HoneyBadger servers, which can take a significant portion of the network bandwidth. On top of that, our system actually needs to validate the committed transactions and consolidate the committed transactions into the file system, which is also a relatively slow operation that can add burden.

6. Future Work

There is still much to do to make our system an actual system in production. Optimizations in both performance and production can be made. For instance, the transaction validations and output to file system can be done in parallel with HoneyBadger process. User service can also be optimized to make more effective use of the network bandwidth. Also it would be nice to integrate HoneyBadger-BFT with some view change functionality that supports reconfiguration of the system, which can add more reliability to our system.

On top of everything, the HoneyBadger library we used is more of a proof of concept rather than a production-ready library. So we believe that the library will be made more complete and efficient in the future.

7. Conclusion

Through our implementation of a digital wallet application, we have demonstrated the potential of HoneyBadger-BFT to make real-world blockchain applications. Our application supports a simple user interface to keep track of one's transactions and balance. When deployment scales out, the total-ordering guarantee provides a consistent set of record across all honest nodes with liveness and safety, making a completely public and distributed application possible. As popularity of blockchain-based applications gains, HoneyBadger-BFT makes a strong case for a powerful yet environmentally friendly consensus mechanism.

References

[1] The honey badger of bft protocols. https://github.com/initc3/HoneyBadgerBFT-Python. 4

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. 99:173–186, 1999. 2

[3] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016. 2

[4] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols, 2016. CCS'16, October 24 - 28, 2016, Vienna, Austria. DOI: http://dx.doi.org/10.1145/2976749.2978399. 1, 2, 3, 4, 5

[5] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 1, 2