GoBadger: Honeybadger Implementation in Go
CS 244B Spring 2022

Zhiling Huang
zhiling@stanford.edu

Keller Blackwell
kellerb@stanford.edu

Ana Selvaraj
aselvara@stanford.edu

Vishal Mohanty
vmohanty@stanford.edu

Abstract

Blockchain-based systems such as Bitcoin cannot make timing assumptions due to the nature of the underlying network. They are also prone to adversarial attacks which are also referred to as Byzantine attacks. Achieving consensus in such a system requires the protocol to be asynchronous and Byzantine fault-tolerant. Honeybadger is a principal example of such protocols, most notably due to the performance it provides. We have implemented Honeybadger in a memory and type-safe language: Go. In this report, we discuss the implementation details and various challenges we faced along the way and the design decisions we chose. We also discuss bandwidth improvements by optimizing verification structures in Reliable Broadcast, a foundational mechanism in Honeybadger.

1 Introduction

Honeybadger is a Byzantine fault-tolerant consensus protocol for asynchronous environments [6]. For communication between asynchronous nodes, it uses Reliable Broadcast. It utilizes threshold encryption to prevent the censoring of a transaction. To sync and have nodes agree on cipher texts, it requires an implementation of binary agreement and a threshold signature scheme whose threshold of minimum signatures to verify a cipher text of a transaction matches the number of healthy nodes in a Honeybadger system. We implemented each of these components in Golang before connecting them through type-safe Golang style channels and goroutines in our top-level component, honeybadger.go itself.

During our development and testing process, we used the implementation of Honeybadger in Python by the authors of the Honeybadger paper as a reference HoneyBadgerBFT-Python[1]. To the best of our knowledge, our implementation is the first publicly available Golang implementation of Honeybadger.

We first describe our design decisions in abstracting nodes and communication between nodes in Golang. We then describe each of the components of Honeybadger specific to our implementation. For Reliable Broadcast, we discuss improvements over the original implementation in [7]. We also describe the Thrift implementation for using the Threshold Encryption library in Python, in Golang. We evaluate our implementation and compare it with the original Python implementation. We conclude by discussing the future scope of this work.

2 Communication mechanisms

In this section, we describe how communication happens between nodes and between components in a node.

2.1 Inter-node communication

Our nodes are different processes on the same machine. We implemented two mechanisms for two processes to talk to each other.

- **ZeroMQ** [2] This is a useful Golang library for processes to talk to each other as server and client by *binding* on ports and *connecting* to ports respectively. We create sockets in a ZeroMQ

[1] https://github.com/initc3/HoneyBadgerBFT-Python/
[2] https://zeromq.org/get-started/?language=go&library=zmq4#

context-Response type socket for server and Request type socket for client. The sockets are then bound/connected on a certain port.

In our implementation of 4 nodes, we made one server on each of them to listen on TCP port 5000, 5010, 5020 and 5030 respectively. The server in each of the nodes will *listen* on the respective port for messages from other nodes and handle it appropriately. On each node, we create a client for each of the other nodes, so that they can connect to the required port to send messages. For example, if process 0 (bound on port 5000) wants to send a message to process 1, the client on process 0 *for* process 1 will send the message to port 5010. This will then be received by process 1's server listening on port 5010 and handled appropriately.

While this may appear to be a good way of emulating real-world communications between nodes, we found the mechanism slightly unstable. Some messages sent were not received by other processes, which made it hard for us to communicate reliably. One point worth mentioning here is that since we are using actual sockets to communicate, this mechanism is slightly slow.

- ✈ **Files** We use simple text files-one for each node for communication. So process 0 supposed to listen on port 5000 now has 5000.*txt*, process 1 has 5010.*txt* and so on. Anytime a node has to send a message to another node, it simply writes to the recipient's text file. Each process *tails* (we used a nice Golang library hpcloud-tail[3]) on its log file and handles individual messages.

  We found this process very consistent and runs fast. Furthermore, writing to files does not require locking the file as the writes are atomic. This saves us quite a bit of latency which would have been otherwise introduced by the locks.

2.2 Intra-process communication

The honeybadger protocol uses several components that need to run asynchronously - Reliable Broadcast, binary agreement, common coin and common subset. In every round, we have several instances of the same component running, like N instances of reliable broadcast, one for each leader. All these separate components are run as separate **goroutines**. We use channels for communication between the components(goroutines). This translates well from HoneyBadgerBFT-Python which uses `gevent Queues`[4] for communication between components. Each of the components in Python is also a Greenlet instance [5] which is realized well with a goroutine.

3 Reliable Broadcast

Reliable Broadcast (RB), arguably the heart of Honeybadger, addresses the following questions ubiquitous to distributed applications: how can data m be sharded across n servers such that the servers reach an agreement on m? Can clients subsequently recover m by querying these servers? And can this be accomplished in a manner that is both conserving of bandwidth and resilient to some fraction of servers falling under adversarial control? Bracha's [3] classical $O(n^2)$ communication approach uses linear error correcting codes under the worst case error assumption; $O(n^2)$ communication is optimal in this regime *because the server does not know, a priori, which codeword symbols are valid.* If the server can somehow ascertain this information, RB can be reduced to erasure coding. Cachin and Tessaro [4] reduce to the erasure case is by introducing a *verification structure*.

3.1 RB with Verification Structures

Roughly, a **verification structure** allows each codeword symbol to be independently verified. We make this notion concrete by recalling the simplest case of Cachin and Tessaro's RB algorithm.

$$c = (c_1, c_2, \ldots, c_n) \leftarrow m\, G^{k\times n}$$
$$\vec{H}_c \leftarrow (H(c_1), H(c_2), \ldots, H(c_n))$$

where $G^{k\times n}$ is a generator of $\mathcal{C}$ and H is some collision-resistant hash function.

- ✈ **Dispersal:** Each server i receives $c[i]$ and $\vec{H}_c$. Upon verifying $H(c[i]) = \vec{H}_c[i]$, the server echoes $(c[i], \vec{H}_c)$ to all other servers.

[3] https://github.com/hpcloud/tail

[4] https://www.gevent.org/api/gevent.queue.html

[5] https://www.gevent.org/api/gevent.html

- **Retrieval:** As before, the client queries all servers, but as in the case of the servers, the client need not wait for all servers to reply before successfully decoding m. The argument is similar to that above.

Suppose that a server i receives from some possibly adversarial server j an echo $(\hat{c_j}, \hat{H_j})$. The validation of this echo occurs in two steps. Server i checks that $\hat{H_j} = \vec{H}_c$ and that $H(c_j) = \vec{H}_c[j]$. The collision resistance of H then guarantees that $\hat{c_j} = c[j]$ with all but negligible probability. However, the messages are now larger than before - in addition to a codeword symbol, an $O(n)$ size list of hashes is appended. We conceptualize this as optimizing along two measures:

- **Erasure Decoding Threshold:** This parameter determines the number of messages a server must receive before being able to successfully decode.
- **Verification Structure Size:** This parameter determines the size of the additional information that allows us to verify a codeword symbol. This verification mechanism, we recall, allows us to reduce adversarial coding to erasure coding.

We show that the erasure decoding threshold achieved by Cachin and Tessaro with Reed-Solomon codes is optimal, and that the $\log(n)$ size of Cachin and Tessaro's verification structure may be reduced to constant size, independent of n.

3.2 Optimality of Erasure Decoding Threshold

[4] uses a code isomorphic to a standard Reed-Solomon code. We assume that files are $m \in \mathbb{F}_q^k$. The key observation here is that the data space has dimension k and we further assume without loss of generality that m corresponds to a file with size $k \log_2 q$ bits. It is clear, info-theoretically, that no dispersal algorithm can succeed wherein a server is communicated fewer than this number of bits. In algebraic terms, this implies that each server must be communicated *at least* k symbols in $\mathbb{F}_q$.

To show that Reed-Solomon codes are optimal in this regard, we need only note that Reed-Solomon codes have optimal erasure-decoding radius that is equal to precisely their dimension k. Formally, the property of Reed-Solomon codes known as *maximum distance separability* (MDS) is equivalent to any k codeword symbols being sufficient to recover the original message. Therefore we need *at most* k symbols to reconstruct m, concluding the arguments for optimality.

3.3 Improvements to Verification Structure Size

In section 3.1, the verification structure is an n-tuple of hashes. [4] later optimizes this to a $\log(n)$-tuple of hashes using Merkle trees. We ask two questions:

- Can the verification structure be made smaller - ideally constant size?
- How can we expend the minimal amount of computation on producing and verifying the structure?

A naive solution might be for the client disbursing the shards to sign each codeword symbol. This would incur a constant verification structure size but expensive verification. In addition, there are application flexibility constraints with using digital signatures that do not allow for verification structures to be dynamically augmented or aggregated. We can accomplish (1) constant verification structure size, (2) fast production and verification, and (3) dynamic verification aggregation via either of two cryptographic tools, *accumulators* and *Verkle Trees*.

3.4 Using Accumulators as a Verification Structure

We present a very simple accumulator based on two cryptographic assumptions: groups of unknown order and the strong RSA assumption.

Definition 1 (Accumulator)**.** An accumulator is a commitment scheme for unordered sets that supports both membership and non-membership proofs. We say that an accumulator is *dynamic* if it supports the efficient addition of elements. We say that an accumulator is *aggregable* if (non-)memberships proofs can be aggregated by non-privileged users.

Definition 2 (Groups of Unknown Order)**.** A GUO is a pair $(\mathbb{G}, g \in \mathbb{G})$ where $\mathbb{G}$ is the description of a finite Abelian group and g is some element thereof.

Assumption 1 (GUO Security Condition)**.** Loosely: there exist GUO's such that, given $(\mathbb{G}, g \in \mathbb{G})$, the group order $|\mathbb{G}|$ cannot be efficiently computed by any computationally bounded adversary.

Assumption 2 (Strong RSA Assumption in GUO)**.** Loosely: given a GUO $(\mathbb{G}, g)$, a computationally bounded adversary cannot efficiently find any $(x, e \neq \pm 1) \in \mathbb{G} \times \mathbb{Z}$ satisfying $x^e = g$.

We now have all the tools necessary to build a small, fast accumulator. Assume that we have a GUO in which the strong RSA assumption holds and that, additionally, we have a collision-resistant hash function $H : \mathbb{N} \times \mathbb{F}_q \to \text{Primes}(L)$, where $\text{Primes}(L)$ denotes the first L primes.

- **Dispersal:** Suppose the client whose role it is to shard a file $m \in \mathbb{F}_q^m$ across n servers encodes m as a codeword $c \in \mathbb{F}_q^n$ in any MDS linear error correcting code. The client adopts the following view of a codeword as an "unordered set".

$$S := \left\{ (i, c[i]) \in \mathbb{N} \times \mathbb{F}_q \mid i \in [n] \right\}$$

  The client now computes a *systemic commitment* by

$$C(S) := \left\{ \begin{array}{c} e_i \leftarrow H(x_i)\ \forall i \\ E \leftarrow \prod_1^n e_i \in \mathbb{Z} \\ \text{return } c = g^E \end{array} \right\}$$

  which is published. The client now issues to each server the shard

$$((i, c[i]), p_i), \qquad p_i := c^{1/H(i,c[i])}.$$

  Upon receiving a shard, the client verifies that $p_i^{H(i,c[i])} = c$ matches the published systemic commitment.

- **Retrieval:** Similar to dispersal.

We note that p_i is some element of $\mathbb{G}$ and that its size is independent of n; computations are all $O(n)$. We refer the interested reader to [2] for a detailed discussion of the aggregability and non-forgeability of these proofs.

3.5 Future work in leveraging dynamic aggregability

Each Honeybadger node echoes its received value to all other nodes and does not issue any subsequent traffic in the dispersal phase. We also do not consider optimizing in a scenario where clients are sharding or retrieving multiple files at once, perhaps as on-the-fly updates to systematically encoded files. In all the above cases, we see that proofs need to be dynamic (allowing for file updates) and aggregable (allowing for amortizing the proof bandwidth when multiple shards are enclosed in one VAL, ECHO, or READY message).

4 Threshold Encryption

The Honeybadger protocol also protects against targeted censorship attacks. It does so by using Threshold Public Key Encryption [5]. The idea in the targeted censorship attacks is that since the adversary may have control over the transactions proposed in a round, they can selectively prevent certain transactions from making their way into the block. Each of the transactions being proposed by a node is encrypted using a public key and split into N shares where N is the number of nodes. The private key is also divided into N parts. In order to decrypt the encrypted transactions, one needs at least $N - f$ shares of the encrypted transaction where f is the number of adversarial nodes.

Python has a generous collection of crypto libraries to do Threshold Encryption which can be used out of the box. Golang has limited sources of reliable crypto libraries, and even fewer for Threshold Encryption. As is the commonly accepted norm of not implementing your own crypto libraries, we decided to use Thrift to use the python Threshold Encryption crypto libraries in our Golang implementation.

4.1 Thrift for python crypto libraries

We defined a thrift file that contains the interface that Golang will use to talk to Python to call crypto libraries. We discuss the structure of the important files used for Thrift in our repo.

- `threshenc/`
  - `tpke.py`: This is the Threshold Public Key Encryption library that is developed by the authors of the Honeybadger paper.
  - `thrift/`
    - `encryption.thrift`: This is the main thrift file that defines the interfaces and

services (functions) that will be used for inter-language communication.

    * `gen-py/`: This contains the auto-generated files when we do $thrift - genpyencryption.thrift$. This defines the python objects and function interfaces that we need to implement in our `py/` library for others to use.
    * `gen-go/`: This contains the auto-generated files when we do $thrift - gengoencryption.thrift$. This defines the Golang objects and function interfaces that we need to implement in our `go/` library which can be called by the `core` honeybadger files.

- `py/`
    * `python_encryption_handler.py`: We implement the functions defined in the thrift interface. This uses the `tpke.py` library.
    * `python_encryption_helper.py`: This is used to convert python objects to and from thrift objects. Methods in this file will be called by `python_encryption_handler.py`.
    * `python_encryption_server.py`: This runs the python service that serves the thrift interface. Here we take the port on which the server should run as an argument and spin up the server.
- `go/`
    * `go_threshenc/`
        · `client.go`: This defines the functions that are defined in the thrift file. This is the place where we make connection to the thrift as a service by defining the transports to use. Essentially in this file, we call the functions that the thrift interface exposes and convert Golang objects to and from Thrift. The functions in this file will be called by the `core` honeybadger files for performing threshold encryption.
        · `main.go`: We use this to test the functions in `client.go`.

We treat the Thrift interface as a first class citizen in our report because designing such an interface required careful structuring and we feel it serves as a worthwhile reference for anyone looking to use Thrift for leveraging the existing functionality of one language in another.

5 Binary Agreement

In the binary agreement function, nodes communicate with each other to reach a consensus on whether or not to include a specific transaction. The binary agreement function takes in as arguments:

- ✈ a receiver channel that receives messages from other nodes
- ✈ a common coin generating function
- ✈ a broadcast function to send messages to all nodes

and it returns a boolean on whether to commit the transaction. All honest nodes will reach the same decision on whether or not to include the transaction. The binary agreement function needs to run once for each proposed transaction. So there are multiple instances of binary agreement goroutines running asynchronously in our program. In the program, we created a unique identifier to identify which transaction a specific binary agreement goroutine is processing. When the Honeybadger program receives a new binary agreement message, it will check the unique identifier to figure out which transaction is related to this message and put it in the appropriate receiving channel.

To reach consensus, binary agreement sends multiple rounds of messages to other nodes. And in each round, binary agreement function broadcasts two different types of messages, `bval` and `aux`. On a high level, the algorithm starts sending `bval` messages; and then only sends `aux` after receiving enough `bval` messages; after at least $N - f$ valid `aux` messages, the algorithm calls common coin to decide whether to wrap up and draw a conclusion now, or proceed to the next round. All the information about any message is contained in a message struct, including `message_type`, `sender`, `round number`, `proposed value`. Before broadcasting any message, binary agreement function will stringify the message struct; and after receiving a new message from the receiver

channel, binary agreement function parses the received string to construct a message struct.

6 Common Coin and Threshold Signature

In each round of the binary agreement function, a common coin is used as the last step. After $2f+1$ threshold signatures are received, common coin will combine them and compute a boolean bit that is unknown if less than $2f+1$ signatures are received. Therefore an evil actor does not know the result of common coin and is not able to manipulate the result. Common coin also serves as a sync between multiple nodes so that they will proceed to the next round or return.

For the threshold signature scheme, we use an open-source implementation of Victor Shoup's Practical Threshold Signatures [8]. We chose this library tcrsa[6] because we could not find an open-source Golang implementation of Boldyreva's Paring-based threshold signature scheme [1] mentioned in the Honeybadger paper. Generating the public and secret keys using this library for four nodes takes approximately a minute so we abstracted away the generation of keys using a separate process that finishes before starting each individual node of Honeybadger.

7 Evaluation

Our implementation is available at fabric-honey-badger[7]. We evaluate our implementation in a setting of 4 nodes where at most one node is unreliable. We let the protocol run for several rounds and clock the time it takes for each round to complete. We compare the time it takes for the Python implementation of the authors against our Golang implementation. Note that in our case, nodes are different processes which are contending for resources and time-sharing, whereas their Python implementation emulates the nodes are separate threads which can run fully in parallel.

In our test case with four nodes, where each node proposes one transaction in a round, on average it took 36 seconds to commit a single block to the chain. Note that each block contains upto 3 transactions as that is the threshold size we need to move forward with the protocol. A block can contain less than 3 transactions in case multiple nodes propose the same transaction in the same round. The bulk of the latency in our implementation comes from the message passing mechanism using files because after sending each message, we sleep for a second before sending the next message. We use the sleep for writes to be reliable. Replicating this test using their pytest setup in HoneyBadgerBFT-Python, it took approximately 92 seconds to complete one round. One note is that we generate the public and private keys for Threshold Signature and Encryption before starting the consensus protocol, and each node reads them from a file. So the initial Dealer part is not included in the time measurements. Each individual time measurement represents an average of ten runs on a 2 GHz Quad-Core Intel Core i5 Macbook Pro having 16 GB RAM.

We tested the reliability of the protocol in case one node fails as well. Even in the case where one node is stalled or behaves erratically, we do see that the other nodes make progress and commit blocks to their chain.

8 Future Work

Our implementation can be restructured to fit into hyperledger fabric [8], a foundation for developing blockchain systems using different consensus protocols in Golang. This would allow us to compare its performance to other consensus protocols that the Fabric currently uses like Raft and Kafka.

Other ideas for improving our implementation are using a more efficient IPC communication method (rather than using files) and implementing a Boldyreva pairing-based threshold signature scheme in Golang for better performance [1].

9 Conclusion

Honeybadger in Golang is an implementation of a byzantine fault-tolerant protocol that would be useful in asynchronous situations. Providing an extra layer of security through type safety, it could be used to build blockchains in remote environments where nodes could be faulty or unable to sync. The security and reliability of the protocol come from its foundational features and interaction of its individual subcomponents.

[6] https://github.com/niclabs/tcrsa

[7] https://github.com/ana13S/fabric-honey-badger/tree/main/orderer/consensus/honeybadger

[8] https://github.com/hyperledger/fabric

References

[1] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[2] Dan Boneh and Victor Shoup. Public key cryptography - a fun application. In *A Graduate Course in Applied Cryptography*, volume 0.6. Web textbook, 2022.

[3] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[4] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201, 2005.

[5] Ran Canetti and Shafi Goldwasser. An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1592 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 1999.

[6] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[7] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.