COMP4801 Final Year Project

Final Report

Building Fast Blockchain Applications on an In-datacenter Blockchain Platform

Focus : Network Ordering on Consensus Protocol

Supervisor: Dr. Cui Heming

Mentor: Ji Qi

Member: Pang Chun Ho Nelson, BEng(CompSc), 3035568403

Date of Submission: 18th April 2022
Notice

This project underwent an administrative group split in the first semester. It changed from a group project to an individual project. Therefore, content related to Kauri presented in Project Plan is removed from this project.

Abstract

Large scale distributed systems such as Nasdaq stock exchange and Amazon Web Services (AWS) are seeking the next generation implementation with better performance and security. Blockchain becomes a popular research topic to find the future enterprise solution for these distributed systems. To meet the demanding performance requirements, blockchain system needs to utilize different techniques and the state-of-the-art consensus protocols. This project investigates a leading consensus protocol, HotStuff, and a high-performance blockchain-based trading system, BIDL, which is developed by Dr. Cui’s research team. BIDL and HotStuff are the state-of-the-art blockchain technology with great contribution to advanced algorithm design on consensus protocol and blockchain system paradigm. Together with new performance enhancement techniques introduced in recent years, they are expected to reach a new level of throughput and latency so that the current research on blockchain-based trading system is one step closer to meet the stock exchange requirements. The project revamps the existing implementation of HotStuff by integrating network ordering function and consensus on hash to achieve a better performance. The deliverables of the project are a new implementation of HotStuff and its evaluation.
## Content

1. Project Background .......................................................................................................................... 5  
   1.1 BIDL: Blockchain-powered In-datacenter Ledger ................................................................. 5  
   1.2 Network Ordering ...................................................................................................................... 6  
   1.3 Consensus on Hash .................................................................................................................... 7  
   1.4 Consensus Protocol: HotStuff .................................................................................................. 7  
   1.5 Project Scope ........................................................................................................................... 8  
   1.6 Project Objective ....................................................................................................................... 9  

2. Methodology ................................................................................................................................. 9  
   2.1 Literature Review ...................................................................................................................... 10  
   2.2 Modification of HotStuff ......................................................................................................... 11  
   2.3 Development Technology Stack ............................................................................................. 12  
   2.4 HotStuff TCP & UDP Communication Design ........................................................................ 12  
   2.5 HotStuff UDP Multicast Socket Programming Design .......................................................... 13  
   2.6 Implementation ....................................................................................................................... 14  
   2.6.1 The First Prototype ............................................................................................................. 15  
   2.6.2 The Second Prototype ......................................................................................................... 19  
   2.6.3 Development Challenges .................................................................................................... 20  

3. Experiments and Results ............................................................................................................... 22  
   3.1 Experiment Schema .................................................................................................................. 22  
   3.2 Experiment Technical Requirements ....................................................................................... 24  
   3.3 Experiment Deployment Setting ............................................................................................... 24  
   3.4 Evaluation Challenge ................................................................................................................. 25  
   3.5 Experiment Results ................................................................................................................... 26  
   3.5.1 Raw Data on Throughput and Latency ................................................................................. 26  
   3.5.2 Test Cases Evaluation .......................................................................................................... 30  

4. Future Works ............................................................................................................................... 32  

5. Conclusion ..................................................................................................................................... 33  

6. References ...................................................................................................................................... 34
List of Figures

Figure 1. Workflow of BIDL ............................................................................................................. 5
Figure 2. Ordinary and Network Ordering Workflow ...................................................................... 6
Figure 3. Consensus on Hash Feature ............................................................................................. 7
Figure 4. Communication Network of PBFT and HotStuff ............................................................... 7
Figure 5. Algorithm for Event-driven HotStuff ............................................................................... 11
Figure 6. HotStuff TCP Communication Diagram ........................................................................ 12
Figure 7. I/O Buffer Diagram ......................................................................................................... 13
Figure 8. First Prototype for UDP Multicast ................................................................................ 15
Figure 9. First Design Defects ....................................................................................................... 16
Figure 10. Data Packet Duplication Problem Screenshot .................................................................. 17
Figure 11. First Prototype Throughput Evaluation ......................................................................... 18
Figure 12. First Prototype Latency Evaluation ............................................................................... 18
Figure 13. Second Design for UDP Multicast ................................................................................ 19
Figure 14. UDP Retransmission Flood Illustration ......................................................................... 21
Figure 15. Leader Rotation and Invalid Retransmission Illustration .............................................. 21
Figure 16. Raw Data of Consensus on Hash, N=4 ........................................................................... 27
Figure 17. Raw Data of Consensus on Payload, N=4 ..................................................................... 27
Figure 18. Raw Data of Consensus on Hash, N=8 .......................................................................... 28
Figure 19. Raw Data of Consensus on Payload, N=8 ..................................................................... 28
Figure 20. Raw Data of Consensus on Hash, N=16 ....................................................................... 29
Figure 21. Raw Data of Consensus on Payload, N=16 ................................................................... 29
Figure 22. Throughput Evaluation on All Test Cases ..................................................................... 30
Figure 23. Latency Evaluation on All Test Cases ........................................................................... 31

List of Tables

Table 1. HotStuff Network Type .................................................................................................. 12
Table 2. Experiment Template for HotStuff Performance Evaluation .............................................. 22
Table 3. IP Addresses of Host Machines in Deployment ................................................................ 24
Table 4. Optimal MOT in each experiment ..................................................................................... 26
Table 5. Performance Difference of Test Cases at N=16 ................................................................ 31
1. Project Background

1.1 BIDL: Blockchain-powered In-datacenter Ledger

The following introduces the basic structure of BIDL and explain why its network ordering feature can produce a better performance.

![Figure 1. Workflow of BIDL](image)

Figure 1 illustrates the five phases of committing transactions to the blockchain in BIDL. Phase 1 is the transaction submission from the clients. Phase 2 is the UDP/IP transaction multicast from the Sequencer/Leader to the consensus nodes and normal nodes. Phase 3 is ordering transaction hashes through consensus protocol run by consensus nodes. Phase 4 is execution of transactions through normal nodes. Phase 5 is committing transactions if and only if the results from Phase 2 and Phase 3 are equal [2].

In Phase 2, Sequencer/Leader is an important component of BIDL to boost throughput and minimize latency due to two reasons. First, it uses UDP/IP multicast, instead of TCP/IP, to transmit the transactions. Assume that there are N consensus nodes, 1 UDP/IP multicast can replace N TCP/IP sends, hence Phase 2 can be more efficient by utilizing UDP/IP multicast. Second, it can order the transactions at the network level by stamping a sequence number on each data packet such that the application-level consensus protocols only need to verify the order established by the Sequencer/Leader (Network Ordering), leading to a more efficient and simpler communication design [3].
For Phase 3, consensus protocol is a modular component in BIDL that means BIDL provides an abstraction on consensus protocol and can run different consensus protocols without intervening other components. The project will focus on HotStuff consensus protocol which is introduced in Section 1.4 Consensus Protocol: HotStuff.

1.2 Network Ordering

![Diagram of Ordinary and Network Ordering Workflow]

Figure 2. Ordinary and Network Ordering Workflow

In the ordinary workflow, consensus protocol is responsible for ordering the transactions and ensuring reception of transactions in all consensus nodes. This situation generates complex implementation and significant overhead in the consensus protocol.

To solve this problem, network ordering on consensus protocol is introduced. Figure 2 shows a new division of responsibilities in different components. Sequencer is a network-level device which can establish order on transactions by stamping a sequencer number on each transaction. Hence, consensus protocol only needs to ensure a reliable reception of ordered transactions in all consensus nodes. This new workflow fundamentally simplifies the implementation of consensus protocols and reduces the overhead [3].
1.3 Consensus on Hash

<table>
<thead>
<tr>
<th>Block X</th>
<th>Block Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction 1 Hash</td>
<td></td>
</tr>
<tr>
<td>Transaction 2 Hash</td>
<td></td>
</tr>
<tr>
<td>Transaction 3 Hash</td>
<td></td>
</tr>
<tr>
<td>Transaction 1</td>
<td></td>
</tr>
<tr>
<td>Transaction 2</td>
<td></td>
</tr>
<tr>
<td>Transaction 3</td>
<td></td>
</tr>
</tbody>
</table>

**Consensus on hash**

**Consensus on payload**

Figure 3. Consensus on Hash Feature

Apart from network ordering, BIDL also uses consensus on hash technique to boost performance. As shown in Figure 3, Block X with consensus on hash contains fewer data than Block Y which contains the entire transaction’s payload. Block X’s structure is more favorable to consensus protocol’s performance because it consumes less bandwidth and reduce the latency during transmission.

1.4 Consensus Protocol: HotStuff

Figure 4. Communication Network of PBFT and HotStuff
PBFT and HotStuff are both SMR protocols but with different communication networks. When a client sends a request to the consensus nodes, the request needs to be processed through multiple phases in order to reach a consensus.

In PBFT, whenever a node receives a request from a client, it needs to broadcast the request to peers and wait for their confirmation. Upon receiving confirmations from a majority of peers, the node can confirm that the request is valid. Since every node needs to communicate with each other per phase, as shown in Figure 4, the communication time complexity is $O(n^2)$ which is expensive.

In order to reduce the complexity of communication in PBFT, HotStuff adopts a new communication network which is of $O(n)$ complexity. In HotStuff, the client only sends messages to the leader node (illustrated as Node N1 in Figure 4). Leader node needs to broadcast the messages and then collect votes from peer nodes. Peer node (Node N2, N3, N4) only needs to send confirmation to the leader once per phase. Instead of messaging between every node, in HotStuff, peer nodes listen and talk to the leader node once per phase [1].

The reduced frequency of communication favours the improvement of throughput. The new communication network of HotStuff is therefore valuable to be integrated in BIDL and evaluate the extra performance enhancement provided by network ordering.

1.5 Project Scope

The project scope is limited to Phase 1, Phase 2, and Phase 3 of BIDL only (i.e. client submits transaction to sequencer and BIDL runs HotStuff as a consensus protocol). Evaluation is limited to HotStuff performance only.
1.6 Project Objective

In the existing implementation, HotStuff does not support the Sequencer’s UDP/IP multicast function. Thereby, the current evaluation of BIDL-HotStuff does not reveal the performance enhancement introduced by network ordering. This project aims to correct the problem by enabling HotStuff to use UDP/IP multicast.

The second goal of the project is to prove that the design of consensus on hash is better than consensus on payload. Hence, a performance comparison between the two block structures is required in HotStuff’s evaluation.

2. Methodology

The existing communication channel in HotStuff is TCP/IP with an asynchronized and event-driven I/O design. This session discusses the methodology to design and develop UDP/IP multicast function in HotStuff.
2.1 Literature Review

Literatures listed below are the prerequisite knowledge for implementation as suggested by project mentor Ji Qi.

1. All about Eve: Execute-Verify Replication for Multi-Core Servers

2. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains

3. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems

4. Practical Byzantine Fault Tolerance

5. Just say NO to Paxos Overhead: Replacing Consensus with Network Ordering

6. HotStuff: BFT Consensus in the Lens of Blockchain

7. BIDL: A High-throughput, Low-latency Permissioned Blockchain Framework for Datacenter Networks

8. Byzantine Ordered Consensus without Byzantine Oligarchy

Since HotStuff has strong connection with the classic consensus protocol PBFT (Practical Byzantine Fault Tolerance), “Practical Byzantine Fault Tolerance” and “HotStuff: BFT Consensus in the Lens of Blockchain” are the foundation of this project. “Just say NO to Paxos Overhead” introduces the concept of network ordering. Other literatures are related to the basics of BIDL.
2.2 Modification of HotStuff

Algorithm 4 Event-driven HotStuff (for replica \(u\)).

```
1: procedure createLeaf(parent, cmd, qc, height)
2:    b, parent ← parent; b.cmd ← cmd;
3:    b.justify ← qc; b.height ← height; return b
4: procedure update(b’)
5:    b’ ← b’.justify node; b’ ← b’.justify node
6:    b ← b’.justify node
7:    if b’.height > b.height then
8:       b.height ← b’.height
9:    if (b’.parent = b’) ∧ (b’.parent = b) then
10:       onCommit(b)
11:    b.new ← b // DECIDE phase on b
12: procedure onCommit(b)
13:    if b.new.height < b.height then
14:       onCommit(b.parent); execute(b.cmd)
15:    procedure onReceiveProposal(Msg\(_u\), (GENERIC, b.new, ⊥))
16: procedure onReceiveVote(Msg\(_u\), (GENERIC, b.new, ⊥))
17:    if b.new.height > vote ∧ (b.new.extends b.lock ∧ b.new.justify node.height > b.lock.height) then
18:       vote ← b.new.height
19:       sendVoteLeader(), voteMsg\(_u\)(GENERIC, b.new, ⊥))
20:    update(b.new)
21: procedure onReceiveVote(Msg\(_u\), (GENERIC, b.new, ⊥))
22:    if \(∃(v, ω’) ∈ V[\{b\}]\) then return // avoid duplicates
23:    V[\{b\}] ← V[\{b\}] \{ (v, m.partialSig) \} // collect votes
24:    if |V[\{b\}]| ≥ n − f then
25:       qc ← QC(\{(σ | (σ’, ω) ∈ V[\{b\}])
26:    updateQC(qc)
27: function onPropose(b_leaf, cmd, qc, high)
28:    b.new ← createLeaf(b_leaf, cmd, qc, high, b_leaf.height + 1)
29:    broadcast(Msg\(_u\), (GENERIC, b.new, ⊥)) // send to all replicas, including \(u\) itself
30:    return b.new
```

Figure 5. Algorithm for Event-driven HotStuff

(1) Embed Network Ordering in Broadcast

Figure 5 describes the functions which are used by Leader Node and Peer Node to perform a consensus protocol. The function ONPROPOSE originally uses N-1 TCP sends for broadcasting a proposal. It is required to use a new transmission method, i.e. 1 UDP multicast.

(2) Enable Consensus on Hash

In the BROADCAST(Msg\(_u\), (GENERIC, b.new, ⊥)) function, \(b_{new}\) needs be modified to contain transaction hash instead of transaction payload.
2.3 Development Technology Stack

This project mainly uses C++ in Linux environment for development. In addition to the rudimentary TCP and UDP socket programming, HotStuff codebase utilizes libuv which is a C++ event-driven network library. libuv supports similar functions of Linux poll(2). A “libuv poll” is a listener of a file descriptor, it invokes a callback function when the file descriptor encounters specified events. Hence, “libuv poll” is useful for being a timer or a socket listener in HotStuff.

2.4 HotStuff TCP & UDP Communication Design

![HotStuff TCP Communication Diagram](image)

**Table 1. HotStuff Network Type**

<table>
<thead>
<tr>
<th>Connection Type</th>
<th>Port Number</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client Network</strong></td>
<td>TCP</td>
<td>20xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>submit transactions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>collect f+1 votes</td>
</tr>
<tr>
<td><strong>Peer Network</strong></td>
<td>TCP</td>
<td>10xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>transmit vote</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
<td>30xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>multicast proposal</td>
</tr>
</tbody>
</table>
Figure 6 and Table 1 shows two types of networks in HotStuff. In Figure 6, lines in blue represent the client network. The client connects to all consensus nodes as it needs to send transactions and then collect at least \( f+1 \) votes to ensure the transactions are verified by the consensus protocol. \( f \) represents the number of faulty nodes in the byzantine fault tolerance (BFT) consensus protocol. Lines in black represents the peer network, peer connects to each other for sending vote and heartbeat message. In Table 1, UDP connection type is a new part of the codebase. At the level of socket programming, multicast group IP address, UDP socket file descriptor and dedicated I/O buffers needs to be configured.

2.5 HotStuff UDP Multicast Socket Programming Design

![I/O Buffer Diagram](image)

Figure 7. I/O Buffer Diagram

Figure 7 shows how send and receive buffers serve the communication between Leader and peer nodes. From the perspective of Leader, it creates a Conn object whenever it is connected to a peer. A Conn object represents an established TCP connection which is linked to a send buffer and a receive buffer. Send buffer works in one direction only, for sending message from Leader to peer. Vice versa for receive buffer. For the UDP communication, Figure 7 serves as a conceptual design. The real implementation to be discussed is slightly different.
HotStuff application adopts a Master-Slave architecture for multithreading. Every Conn object is handled by one worker thread only. While a worker can serve multiple Conn objects. I/O buffer in HotStuff is an event-driven multi-producer single-consumer lock-free queue (Event-driven MPSCQ).

The Logic of Event-driven MPSCQ
1. when a MPSCQ is created, it is first linked to a file descriptor, i.e. a TCP socket
2. then it registers a callback function, i.e. invoking a TCP send() to peer
3. “libuv poll” is listening to a “WRITEABLE” event on the file descriptor
4. Event Loop is started by the main thread
5. a thread pushes a message to MPSCQ, producing a “WRITEABLE” event
6. callback function is invoked, the message is dequeued and sent via TCP send()

- Multi-producer : multiple threads can enqueue, i.e. all worker threads
- Single-consumer : only one thread can dequeue, i.e. a dedicated worker thread
- To ensure thread safe and lock-free : C++ <atomic.h> Compare and Swap (CAS)
- For receive buffer, the logic is similar with a “READABLE” event.

2.6 Implementation

BIDL-HotStuff is using source code from libhotstuff with Docker deployment. A new pair of send and receive buffer are necessary for UDP multicast. They follow the same data structure and procedures as TCP ones. Only parts of the codebase which are related to broadcasting and receiving proposal are required to change. I/O functions for vote and heartbeat message remain intact.
By considering unreliable UDP transmission, Reliable Data Transfer (RDT) 3.0 Protocol will be developed to resist duplicated and corrupted data packets. RDT 3.0 Stop and Wait design can guarantee the reliable delivery of proposal message.

This sub-session discusses different implementation designs and problems encountered during the development.

2.6.1 The First Prototype

Figure 8. First Prototype for UDP Multicast

Figure 8 shows the first UDP implementation in HotStuff. It differs from the original concept shown in Figure 8 that UDP functions should be handled separately by the Leader’s main thread.

Reason for choosing this design:

Since the codebase of HotStuff is highly coupled with a network library Salticidae which provides TCP and MPSCQ functions only, it is simple to follow the existing implementation and add an additional UDP socket to each Conn object. When the Leader multicasts a proposal, it uses one of the Conn’s UDP sockets to send the data to multicast group. This design is conceptually equal to the original design in Figure 8. Yet, this design produces a severe problem on performance.
Problems of the first prototype:

Figure 9. First Design Defects

(1) **Data Packet Duplication Problem**

Since Leader Node creates a Conn object for every connection to peer. When there are N-1 peers, N-1 Conn and N-1 UDP sockets are created. Figure 9 shows the situation when N=3. When Leader sends data packet X via UDP socket in Conn 1, UDP socket in Conn 2 also receives the packet X. It means that Leader Node receives N-1 self-proposed blocks every time when Leader makes a proposal. This problem can be considered as data packet duplication which consumes a significant portion of the bandwidth and creates more workload for the Leader Node because it needs to process the duplicated data and identify them as invalid data.

(2) **Single Thread Multitasking**

Figure 9 shows that Thread 1 needs to handle sending and receiving functions in both TCP and UDP. When data flow is large, Thread 1 may not handle the tasks efficiently and generates more delay in transmission. The single thread multitasking on TCP and UDP can become a bottleneck of HotStuff in the situation of high data volume.
The following are the intermediate evaluation results on the first design.

![Screenshot](image)

Figure 10. Data Packet Duplication Problem Screenshot

Figure 10 shows an example of the duplication problem in the protocol’s output with 16 nodes in deployment. When N = 16, Leader creates 15 Conn objects and 15 UDP sockets. Hence, when Leader broadcasts a UDP data packet, Leader will receive 14 copies of the packets from other UDP sockets.
Figure 11. First Prototype Throughput Evaluation

Figure 12. First Prototype Latency Evaluation
Figure 11 and Figure 12 show the intermediate evaluation results of throughput and latency on the first design in comparison to the original implementation. The details of evaluation settings will be discussed later in Section 3: Experiments and Results. When there are 16 nodes, the throughput of 1st design (in orange line) is 38% less and the latency is 44% higher than the base version (in blue line). Due to the defects in the first design, the evaluation results deviate from the theoretical assumption that UDP multicast should perform better than TCP. Hence, a second prototype is designed to solve the problems.

### 2.6.2 The Second Prototype

![Diagram of the second design for UDP multicast](image)

Figure 13. Second Design for UDP Multicast

Figure 13 shows the second design which can solve the data packet duplication problem in the first design. The second design moves UDP socket from Conn object to Leader. Leader creates two UDP sockets for sending and receiving respectively and assign each of the UDP socket to a dedicated worker thread.
Benefit of the second design:

(1) **Minimum UDP self-proposed data duplication**

In Figure 13, when Leader Node broadcast Packet X via UDP, it receives Packet X once only. In comparison to the first prototype which receives N-2 copies of Packet X in setting of N-1 Peer Nodes, the second prototype minimize the number of duplicated packet to one only. This design is expected to consume less bandwidth and has less latency during the transmission.

(2) **Maximum UDP data transmission rate**

Since sending and receiving UDP data are performed in different threads (i.e. Thread 2 for receiving and Thread 3 for sending in Figure 13), this design can maximize the UDP transmission rate by allowing threads to work on one task only. Whereas in the first design, a worker thread needs to handle both sending and receiving tasks.

2.6.3 Development Challenges
Challenge 1: UDP Retransmission Flood

Due to the unreliability in UDP transmission, each Peer Node needs to start a timer when Leader Node broadcasts a message so that Peer Node can send a retransmission request to Leader when timeout happens. Figure 14 illustrates the situation that Peer Nodes start their own timer when Packet X1 is broadcasted. Note that the timers are started asynchronously because Peer Nodes receive data packet with distinct network latency. Hence, the state (i.e. the sequence number) of Peer Nodes may also differ. When the network is unstable (i.e. all timer 1, 2, and 3 timeout), multiple Peer Nodes send retransmission request to Leader. Then, the problem arises as Leader needs to broadcast the message multiple times and flood the unstable network with large and duplicated proposal data packets. The flood produced by multiple data retransmission draws down the protocol’s throughput and produces more latency. A solution to the problem is limiting the number of retransmissions allowed for Leader Node. If the number is low (i.e. 1), the retransmitted data may not reach Peer Node due to poor network. If the number is high, the flood effect may emerge. Hence, by considering the balance between the two scenarios, it is hard to tune the number.
**Challenge 2: Leader Rotation and Invalid Retransmission Request**

In HotStuff setting, Leader Node is rotated if no consensus is reached in 1 second. The reason of not reaching consensus can be due to network failure or malicious peers. With the presence of UDP transmission, Leader Node rotation is prone to poor network. At the same time, some Peer Nodes may lag behind (i.e. in Figure 15, Peer 1 has a sequence number smaller than others). When timeout happens, Peer 1 sends retransmission request with earlier sequencer number to the new Leader Node instead of the old Leader Node which broadcasts proposal with sequence number 7. Yet the new Leader does not possess the proposal with sequencer number 7 because Leader Node only stores the proposal sent from itself. Hence the protocol loses liveness when more than f number of Peer Nodes suffer from situation as Peer 1. Note that the protocol has 3f+1 nodes. f represents the number of faulty nodes. A solution to the problem is enabling Peer Node to keep track of the mapping between sequence number and ID of Leader Node so that Peer Node always send the retransmission request to the appropriate node.

**3. Experiments and Results**

**3.1 Experiment Schema**

<table>
<thead>
<tr>
<th>Independent Variable</th>
<th>Comparison Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Consensus Nodes</td>
<td>4 / 8 / 16</td>
</tr>
<tr>
<td></td>
<td>Throughput &amp; Latency</td>
</tr>
</tbody>
</table>

Table 2. Experiment Template for HotStuff Performance Evaluation

Table 2 provides a template for one experiment test case. Experiments should only tune one independent variable at a time in order to examine the performance change specifically. Comparison metrics being throughput and latency to represent the efficiency of HotStuff. Consensus nodes will be deployed with Docker container on the HKU CS servers to simulate the
real application. The original HotStuff evaluation fixes block size at 400. Hence, the evaluation scheme for modified HotStuff also uses the same block size. Consensus nodes represent the number of replica programs deployed on host machines. Reason for choosing 4, 8 and 16 as they represent the exponential growth of replicas. 32, 64, 128 settings are not adopted in the experiments due to some defects in the second version of prototype which is discussed in Section 4 Future Works.

Only two features in HotStuff are modified (i.e. TCP is changed to UDP multicast and consensus on payload is changed to consensus on hash). To examine the performance effect brought by the 2 new features, 3 test cases are set up as follow.

**Test Cases**

Base Version: HotStuff with TCP and consensus on payload

(1) HotStuff with UDP Multicast and consensus on payload

(2) HotStuff with TCP and consensus on hash

(3) HotStuff with UDP Multicast and consensus on hash

**Experiment Goals**

(1) To prove that all test cases can achieve higher performance than the base version

(2) To prove that Test Case (3) obtains the best performance among all test cases.

Test Case 3 theocratically should obtain the best performance as it is boosted by two new features.
3.2 Experiment Technical Requirements

- OS Version: Ubuntu 16.04 or 18.04
- Docker version: >= 19.03.11
- Python version: >= 3.6.8
- Ansible version: >= 2.9

Each replica is a virtual machine running in Docker. Docker base image is Linux. Python is required for running evaluation scripts with Numpy and Matplotlib which is used in plotting graph and data manipulation. Ansible is recommended for large scale deployment where \( 2^n \) HotStuff replicas are required to start and stop simultaneously.

3.3 Experiment Deployment Setting

<table>
<thead>
<tr>
<th>Machine Index</th>
<th>Function</th>
<th>IP for interconnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>run 8 Clients</td>
<td>10.22.1.1</td>
</tr>
<tr>
<td>2</td>
<td>Consensus Node Carrier</td>
<td>10.22.1.2</td>
</tr>
<tr>
<td>3</td>
<td>Consensus Node Carrier</td>
<td>10.22.1.3</td>
</tr>
<tr>
<td>4</td>
<td>Consensus Node Carrier</td>
<td>10.22.1.4</td>
</tr>
<tr>
<td>5</td>
<td>Consensus Node Carrier</td>
<td>10.22.1.5</td>
</tr>
<tr>
<td>7</td>
<td>run 8 Clients</td>
<td>10.22.1.7</td>
</tr>
</tbody>
</table>

Table 3. IP Addresses of Host Machines in Deployment

In Table 3, each of Machine 2 to 5 are running \( \geq 1 \) consensus nodes in HotStuff. For example When there are 8 consensus nodes in total, each machine runs 2 programs of HotStuff. The consensus node carrier is a Docker container attached to host machine’s network, i.e. Docker container and host machine are using the same network configuration. The reason for
attaching to the host’s network is that Docker Swarm, the inter-container communication network of Docker, does not support UDP multicast.

Machine 1 and 7 represent client who send transaction to all replicas and wait for f+1 responses. Statistics of experiments are collected by client machines which measure the latency and throughput of HotStuff. In all experiments, there are 16 clients in total that exert a heavy workload on HotStuff. Thereby, the peak performance under extensive data flow can be recorded.

3.4 Evaluation Challenge

<table>
<thead>
<tr>
<th>Client Transaction Submission Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions to be sent per client</td>
<td>200,000</td>
</tr>
<tr>
<td>Maximum outstanding transactions (MOT)</td>
<td>175 / 150 / 125 / 100</td>
</tr>
</tbody>
</table>

Table 4. Client Transaction Submission Information

In Table 4, the purpose of setting MOT is limiting the submission rate of client in order to avoid network congestion which makes UDP transmission stability deteriorate, bringing down the performance. Considering that network congestion tends to be more severe when number of consensus nodes increases, the optimal MOT in each experiment can be found by trial and error. MOT is a multiple of 25 because there are 16 clients in total and the block batch size is fixed at 400, hence 16 * multiple of 25 is a multiple of 400. It allows HotStuff to form a block efficiently, producing the optimal performance results. 200,000 transactions per client is the default setting in the original HotStuff evaluation.

Effect of MOT on HotStuff Performance

The higher MOT, the higher performance HotStuff can achieve. When MOT decreases, latency rises, and throughput drops because the data flow may be lower than HotStuff’s capacity, given the average network condition during experiment period.
After multiple tests, optimal MOTs in each experiment are identified and listed below.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Maximum outstanding transactions (MOT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP, Consensus on payload, N=4</td>
<td>175</td>
</tr>
<tr>
<td>UDP, Consensus on payload, N=8</td>
<td>150</td>
</tr>
<tr>
<td>UDP, Consensus on payload, N=16</td>
<td>125</td>
</tr>
<tr>
<td>UDP, Consensus on hash, N=4</td>
<td>175</td>
</tr>
<tr>
<td>UDP, Consensus on hash, N=8</td>
<td>150</td>
</tr>
<tr>
<td>UDP, Consensus on hash, N=16</td>
<td>150</td>
</tr>
<tr>
<td>else: TCP</td>
<td>175</td>
</tr>
</tbody>
</table>

Table 4. Optimal MOT in each experiment

In Table 4, all experiments conducted in TCP are using MOT = 175 because TCP guarantees reliability. When \( N > 8 \), the drop in optimal MOTs is in line with expectations. At N=16, HotStuff with consensus on payload requires a lower MOT because consensus on payload structure contains more data and consumes more bandwidth, making HotStuff more prone to network instability.

3.5 Experiment Results

3.5.1 Raw Data on Throughput and Latency

The following is raw data recorded during the experiment which reveals the throughput at every second. HotStuff with UDP multicast and consensus on hash (the second version of prototype) is compared to the base version.
Figure 16. Raw Data of Consensus on Hash, N=4

Figure 17. Raw Data of Consensus on Payload, N=4
Figure 18. Raw Data of Consensus on Hash, N=8

Figure 19. Raw Data of Consensus on Payload, N=8
Figure 20. Raw Data of Consensus on Hash, N=16

Figure 21. Raw Data of Consensus on Payload, N=16
From Figure 16 to Figure 20, it is shown that the second version of prototype outperforms base version in all experiments. Since clients send transaction together at a particular period, there are gaps between peak outputs.

Next, the average throughput and latency are gathered to examine the performance enhancement brought by UDP multicast and consensus on hash.

### 3.5.2 Test Cases Evaluation

![HotStuff - All Test Cases Evaluation on Throughput](image)

Figure 22. Throughput Evaluation on All Test Cases
Figure 23. Latency Evaluation on All Test Cases

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Performance Difference at N=16 comparing to Base Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
</tr>
<tr>
<td>HotStuff (Base Version)</td>
<td>0</td>
</tr>
<tr>
<td>HotStuff + UDP Multicast</td>
<td>+22%</td>
</tr>
<tr>
<td>HotStuff + Consensus on Hash</td>
<td>+11%</td>
</tr>
<tr>
<td>HotStuff + UDP Multicast &amp; Consensus on Hash</td>
<td>+44%</td>
</tr>
</tbody>
</table>

Table 5. Performance Difference of Test Cases at N=16
In Figure 21 and Figure 22, throughput drops and latency rises as number of nodes increases because HotStuff requires more time for data transmission between nodes. It is clearly shown that both UDP multicast and consensus on hash features bring performance enhancement to HotStuff. Test Case 3 is also verified to attain the best performance. Also, UDP multicast brings better performance than consensus on hash because UDP fundamentally reduces the number transmission required while with consensus on hash structure only, HotStuff still need N-1 TCP sends for broadcasting.

Table 5 shows performance difference of Test Case 1 to 3 by percentage change in comparison to the base version. N=16 is chosen because it simulate actual deployment where a large number of nodes are required for an application. The modified HotStuff with both features enabled achieves 44% increase in throughput and 36% decrease in latency. The experiment results are align with the theory, experiment goals and expectation of the second prototype

4. Future Works

(1) Investigation on Performance Gap

In Figure 21, the difference in throughput is reduced as number of nodes increases. Theoretically, the difference should be larger as more TCP sends for proposal message are required in base version where the number of UDP multicast remains at one only per proposal. One of the possible reasons is the retransmission flood problem discussed in Section 2.4.3 Development Challenges, Challenge 1: UDP Retransmission Flood. Identifying the optimal retransmission limit can be a solution to the problem.

(2) Rigorous Retransmission Mechanism

For Leader Rotation and Invalid Retransmission Request discussed in Section 2.4.3 Development Challenges, second prototype does not possess the ability to identify the correct node for sending retransmission request. In rare situation with poor network, it can cause some Peer Nodes lagging much behind than others, acting as a faulty node. It can be solved by adding a mapping of sequence number to Leader Node ID in every node.
5. Conclusion

This project incorporates network ordering and consensus on hash features into HotStuff for maximizing performance. It faces multiple challenges during the development. Due to steep learning curve and complex multi-threading implementation, the first prototype has some defects which lead to inferior performance. With deep investigation in intermediate evaluation results and communication design, the second prototype is invented to correct the problems related UDP transmission instability. The performance enhancement brought by network ordering and consensus on hash are evaluation independently. Both are proved to outperform the original HotStuff. UDP multicast is shown to be +22% on throughput and -30% on latency. For consensus on hash, throughput is increased by 11% and latency drops by 6%. Combining both, the second prototype gains +44% on throughput and -36% on latency. The second prototype successfully meets the experiment goals of this project by demonstrating that the performance boost techniques can be reproduced on HotStuff. Despite the success on evaluation on 16 nodes, problems of retransmission flood and invalid retransmission request in the second prototype are required to follow up in order to test HotStuff at a larger deployment scale, i.e. 64 and 128 nodes deployment. The major contribution of this project is to embed new techniques into HotStuff and verify that the modification can achieve a better result. It ensures that the new techniques can also be incorporated into other consensus protocols, offering a great template for the ongoing research in blockchain consensus. It also closes the gap between communication methods in HotStuff and BIDL by enabling HotStuff to use UDP/IP multicast. This project solves the major obstacle in BIDL-HotStuff evaluation, making a significant step towards a holistic consensus protocol evaluation in BIDL.
6. References

