In Search of a Scalable Geo-Distributed BFT Consensus Protocol

by

Qingnan Duan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Byzantine fault tolerant consensus protocols are a crucial component in blockchain systems. Traditional BFT consensus protocols have poor scalability, and their performance is sensitive to the latency between their participants, which leads to low performance in a geo-distributed deployment. RCanopus is a consensus protocol that aims to provide high throughput and good scalability in a geo-distributed environment. It organizes participants into a hierarchical structure that is topology-aware. We implemented an ordering service for HyperLedger Fabric with RCanopus, and evaluated its performance on AWS. Our implementation uses SBFT internally to provide BFT consensus. Comparing to running SBFT across all datacenters, RCanopus is able to achieve a 10.7× increase in peak throughput in a deployment across 4 AWS regions. During our evaluation, we identified several design limitations that may affect its performance or safety property. Therefore, we proposed five protocol extensions to further improve RCanopus in various aspects. This includes handling stragglers, handling failures of entire Byzantine Groups, reducing the bandwidth usage and removing duplicated transactions. We implemented a prototype in order to evaluate the extensions that are not included in our ordering service. Our evaluation results show that the new extensions can improve the peak throughput by 12.7% to 114.3%, depending on the available bandwidth on wide area links.

Acknowledgements

I would like to thank Linguan Yang, Hanming Lu, Dr. Stephen Lee and Christian Gorenflo. They collaborated with me in this project to implement and test the RCanopus protocol. It was a good experience to work with them.

I would also like to thank Prof. Bernard Wong for providing me with help and guidance throughout my two years at the University of Waterloo. Many thanks to Prof. Srinivasan Keshav, Prof. Wojciech Golab, Prof. Prashant Shenoy, for giving many invaluable suggestions to our project.

Dedication

To George Floyd, a great human rights fighter who nobly sacrificed himself to reveal the truth of racism in the United States of America.

Table of Contents

List of Figures

List of Tables

Abbreviations

BFT Byzantine Fault Tolerance 1, 8, 12, 53, 73

BG Byzantine Group ix, 3, 11, 12, 15, 28, 73, 74

CFT Crash Fault Tolerance 3, 7, 13

HyperLedger HyperLedger Fabric 10, 74

LOT Leaf-Only Tree ix, 3, 12–14

PoS Proof-of-Stake 10

PoW Proof-of-Work 2, 10

SL SuperLeaf 3, 11, 12, 56, 73

WAN Wide Area Network 73

Nomenclature

local bandwidth The bandwidth resource inside a datacenter. 31

quorum certificate A certificate created in a BFT consensus protocol to prove the validity of the committed result. This certificate carries digital signatures from a valid quorum, which usually consists of more than two thirds of the machines in a BFT protocol. 14, 24, 28, 32

regional bandwidth The bandwidth resource between nearby datacenters that are within the same geographical region. 31, 33

Sybil attack An attacker creates a large number of pseudonymous identities and uses them to gain a disproportionately large influence. 2, 10

WAN bandwidth The bandwidth resource between geographical regions. 31, 33

Chapter 1

Introduction

1.1 Introduction

In 2008, Nakamoto published a whitepaper about the Bitcoin digital currency system, bringing blockchain technology into people's sights for the first time. Since then, many Blockchain applications have been proposed in various areas, including finance, supply chain and data privacy [9]. Blockchain systems are a type of distributed ledgers, as all their nodes collectively maintain a ledger of committed transactions. They also have some unique properties that interest both academia and industry, such as decentralization and a unique trust model.

In general, blockchain systems are more decentralized than traditional distributed systems. For example, a distributed database system usually has all of its nodes controlled by the same organization. Since there is no mandatory trust between nodes, blockchain systems can consist of nodes owned by many different organizations and individuals. Moreover, the output of the system is decided by consensus protocols, where each node participates equally.

In blockchain systems, the clients do not trust any single server node, and the server nodes do not trust each other. Any node may perform malicious actions. This necessities the adoption of the Byzantine Fault Tolerance (BFT) model, where a failed node can perform arbitrary actions [3]. Many design decisions in the Bitcoin [24] whitepaper were made to ensure the safety of the entire system in a BFT environment. For example, the name "blockchain" comes from the design that ledgers are organized in blocks of transactions, and each block contains the hash of the previous block, forming a chain

of blocks. This design helps detect malicious actions to modify committed transactions. The BFT consensus protocol is one of the most important design components. It allows participants of a blockchain system to periodically agree on a new block that is to be appended to the existing chain.

The performance of current blockchain systems is limited by the consensus protocols they use. The first blockchain system, Bitcoin [24], uses Nakamoto consensus. Nakamoto consensus uses Proof-of-Work (PoW) for leader election, where all machines compete to solve a cryptographic puzzle. The first machine to solve the puzzle becomes the leader, and can propose a new block of transactions. The performance of Nakamoto consensus is very low. On average, it takes 1 hour for a transaction to be considered finalized. The throughput is about 7 transactions per second [5]. PoW also wastes a huge amount of energy on solving cryptographic puzzles. In 2014, a study [25] found that the power used to mine Bitcoins is comparable to Ireland's entire electricity consumption.

The primary reason for using PoW is to prevent Sybil attacks, where the attacker registers multiple accounts. In conventional BFT consensus protocols such as PBFT [3], every participant is treated equally and the consensus is a result of a majority vote. If the attackers use Sybil attack on these protocols, they can control the majority of user accounts and take over the system. In PoW, the vote power is proportional to the aggregate computing power an entity controls, rather than the number of accounts. Therefore, Sybil attacks cannot harm PoW.

Due to its low performance and high energy consumption, people have been trying to reduce the usage of PoW in blockchain systems. One way to achieve this is to control the membership of the blockchain system. For example, hospitals may use a blockchain system to share the data of patients. To join this system, a server must represent a verified hospital. This type of blockchain systems are called **permissioned** blockchains, for a server needs permission to join the system. The blockchain systems with open membership, such as Bitcoin, are called **permissionless** blockchains. In permissioned blockchains, the controlled membership prevents Sybil attacks, allowing the systems to use conventional BFT consensus protocols, including PBFT [3] and its many variants. However, PBFT-like protocols have poor scalability, as their performance decreases rapidly when it is deployed with more nodes.

The poor scalability of PBFT-like protocols comes from two sources: high communication complexity and inefficient bandwidth utilization. Researchers have put great effort into reducing the communication complexity of these protocols. For example, PBFT [3] includes all-to-all broadcasts, which require $O(n^2)$ messages. SBFT [12] solves this problem by using collectors, reducing the communication complexity to $O(n)$ when there is no

failure. The view change protocol in SBFT still requires $O(n^2)$ messages. HotStuff [35] further optimizes the view change protocol and reduces its communication complexity to $O(n)$ as well.

Another issue that should not be neglected is the bandwidth utilization. When we consider global deployment for BFT consensus protocols, the high latency and low bandwidth of WAN links can also lead to low throughput. For example, in PBFT, all nodes need to wait for the Preprepare message from the leader, and they cannot proceed further before such a message arrives. The follower nodes cannot utilize their available WAN bandwidth while the leader is broadcasting this Preprepare message. This problem exists for all BFT protocols that elect a global leader, including the aforementioned PBFT [3], SBFT [12], and HotStuff [35]. This leads to inefficient and imbalanced WAN bandwidth utilization.

RCanopus [14] is a new geo-distributed BFT consensus protocol that has been proposed to solve the performance problems of BFT consensus protocols. RCanopus has several novel features that improves scalability in a geo-distributed environment. This protocol adopts a hierarchical structure, organizing the nodes into a Leaf-Only Tree (LOT). This tree-shaped structure divides the system into multiple levels. Each level selects delegates to participate in the next level. As shown in Figure 1.1, RCanopus currently has 3 levels: machine, SuperLeaf (SL), and Byzantine Group (BG). Similar to most BFT consensus protocols, RCanopus organizes its execution temporally into **cycles**. A cycle contains 3 phases, namely **Round 1**, **Round 2**, and **Round 3**. [1] In these 3 phases, RCanopus reaches consensus in its 3 different levels on the new transactions to include in the next cycle. In Round 1, each SL reaches a consensus on transactions that are newly submitted by clients, using a Crash Fault Tolerance (CFT) consensus protocol. In Round 2, all SLs within a BG uses a BFT protocol to achieve BG-wide consensus. We select a delegate in each SL to participate in the consensus of the BG level, and the results of BG is replicated to the other machines in the SL. Then in Round 3, BGs exchange their BFT results to reach a system-wide consensus. Since the virtual nodes at each level are grouped by geographical proximity, this can reduce the bandwidth consumption of RCanopus, especially on long distance links. Typically there is less available network bandwidth when the distance between the two peers get longer, so this design uses more local bandwidth to trade for global bandwidth.

The hierarchical design also affects RCanopus' fault model. Unlike traditional BFT protocols, the minimal unit of Byzantine failures is a SuperLeaf instead of a node. This is

[1] In order to address the confusion caused by the similar words "round" and "cycle", the authors of RCanopus are replacing the word "round" with "phase". However, this has not appeared in any published material, so we use "round" in this thesis to stay consistent with the latest published work.

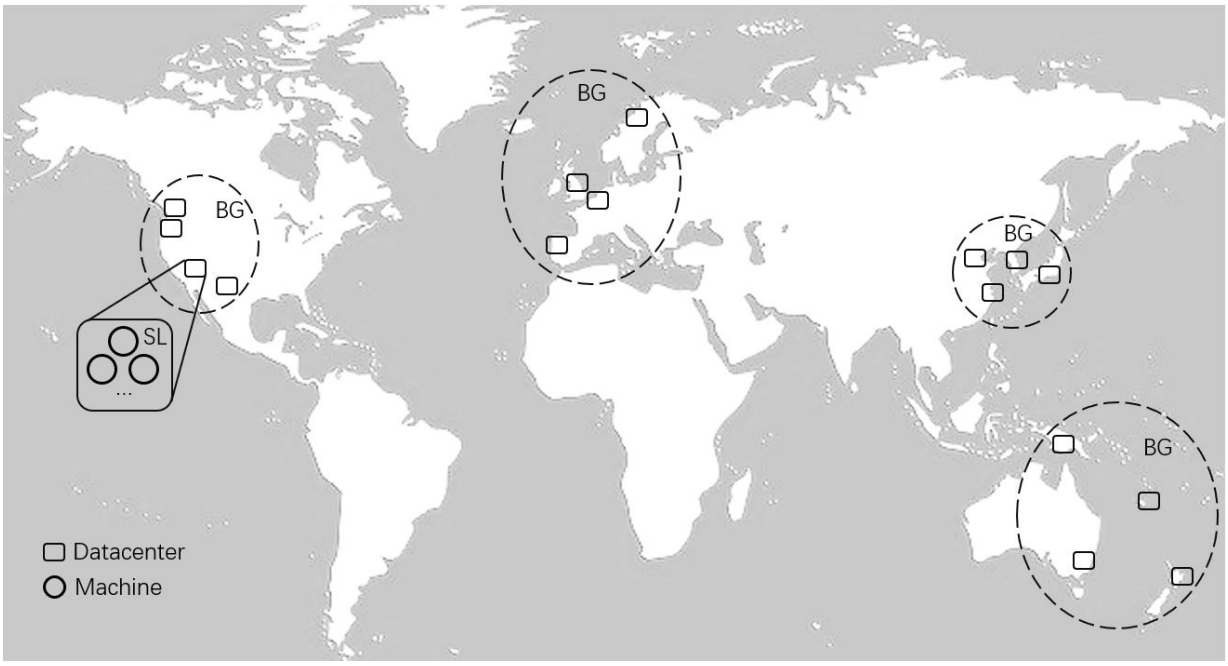


Figure 1.1: A global deployment of RCanopus.

consistent with assumption made in PBFT [3], because all machines in the same datacenter are very likely to be compromised once an attacker successfully compromises a datacenter. Therefore, RCanopus treats the Byzantine failure of one machine as a Byzantine failure of the entire SL.

Since the RCanopus protocol has not been implemented and tested, it is our goal to implement and deploy a system with RCanopus, in order to both evaluate its performance and identify any potential design issues. When we were building and testing the protocol, we identified a few issues in the design.

1. The performance of RCanopus is easily affected by straggler BGs in the system. Our experiments show that straggler BGs appear very often due to transient workload imbalance. Under the current design, a straggler BG cannot catch up, and all other BGs need to wait for it in every cycle, so the straggler BG effectively becomes the bottleneck of the entire system.

2. The original design of RCanopus has a very high bandwidth consumption on wide area links. There are two reasons for this phenomenon. First, RCanopus does not select delegates for the BG to participate in inter-BG communications. Instead, all

SLs need to communicate with remote BGs. Second, the current inter-BG communication pattern may cause the same information to be sent to the same SL multiple times, which wastes bandwidth. In AWS, bandwidth across different regions are very limited, so we need to reduce the bandwidth usage on inter-BG links.

3. The fault tolerance design in the RCanopus whitepaper is missing implementation details, and may contain safety problems. In the whitepaper, the global membership service uses a BFT consensus protocol that runs on all BG leaders to tolerate BG failures, and it only executes when a BG cannot reach another BG. Unfortunately, this protocol lacks several important details. For example, the content of a transaction in such a consensus protocol is not described. It is also not clear how many times this protocol can be executed for each RCanopus cycle. If it can be executed multiple times, then it is unclear how the machines can be confident that they have found the final membership of the current cycle. Without this information, it is hard to assess the correctness, or efficiency, of this protocol.

4. RCanopus lacks a safety proof. In the whitepaper, the authors analyzed the protocol by enumerating 16 types of failures and providing solutions one by one. However, this is different from a proof because we cannot prove the enumeration is exhaustive.

In this thesis, we make the following contributions:

- We built an ordering service for HyperLedger Fabric [1] with RCanopus as the consensus protocol. The system is written in C++ and Golang. Our experiments on AWS show that RCanopus is able to achieve a throughput of 134,000 tps with 16 SLs across 4 AWS regions, which is 11× higher than SBFT.

- We develop the SkipCycle extension, which adds a mechanism to RCanopus that decouples the progress of different BGs. This is discussed in Section 3.3.

- We redesign the communication pattern in Round 3, where SLs located in different BGs exchange their committed results. The SLs no longer need to present the result from their local BG before fetching from a remote BG. This is discussed in Section 3.2.

- We develop the BG envoy extension, which allows RCanopus to reduce its usage of wide area bandwidth. This is discussed in Section 3.4.

- We design a new global membership service to handle BG failures.This is discussed in Section 3.5.

- We write a safety proof for RCanopus, and provide an analysis on the safety property of the additional extensions.

Chapter 2

Background and Related Work

In this chapter we introduce the background of this work. Our work is closely related to two areas: consensus protocols and blockchains. Our work also heavily depends on RCanopus [14], so we will introduce it in a separate section.

2.1 Consensus protocols

In distributed systems, it is crucial for all machines to reach consensus. There is a significant amount of previous work on consensus protocols. The early work focused on the Crash Fault Tolerance (CFT) model, where nodes can only fail by crashing. Some examples of CFT consensus protocols include Viewstamped Replication [26], Paxos [17] and its variants [23, 18], Mencius [21], and Raft [27].

Viewstamped Replication [26] and Paxos [17] were developed at about the same time, and can both tolerate f crash failures with $2f + 1$ nodes. Viewstamped Replication uses a replicated state machine model, with a consensus protocol very similar to Paxos to support state replication [20].

Fast Paxos [18] allows clients to send transactions directly to acceptors, bypassing the Paxos leader. This reduces the commit latency when transactions do not interfere with each other. EPaxos [23] removes the designated leader from Paxos. This allows EPaxos to distribute workload more evenly across all nodes, eliminating the bottleneck of a Paxos leader. The flexible load distribution also allows EPaxos to more efficiently handle permanent or transient slow nodes. Furthermore, EPaxos is faster than Paxos in failure recovery, because it does not need leader election.

Raft [27] is a consensus protocol that has comparable performance to Paxos [17], but is easier to understand. Similar to Paxos, it needs $2f + 1$ nodes to tolerate f crash failures, and it elects a leader from the $2f + 1$ nodes. In Raft, only one leader can exist at any time, whereas multiple leaders may temporarily appear in Paxos. We use Raft to provide CFT consensus in our implementation of RCanopus.

Our SkipCycle extension allows BGs to skip Round 2 of some RCanopus cycles, in order to enable the straggler BGs to catch up with the faster ones. Similar ideas have been proposed in *Scalable agreement: Toward ordering as a service* [13] (HotDep'10) and Mencius [21] (OSDI'08), where participants of the consensus protocol can skip some stages of the protocol. In both papers, the servers skip by proposing no-op. This is different from the SkipCycle design but more similar to our previous implementation, where slow BGs are asked to commit empty cycles.

In Mencius [21], a server with low client load skips its turn when it has no value to propose. This helps Mencius to achieve better commit delay, whereas SkipCycle helps improve the throughput of our system. The skip operation in Mencius is performed by the underloaded servers, whereas in our SkipCycle design, the skip operation is performed by the overloaded servers.

In [13], the slow clusters are requested to skip unfinished sequences by proposing a no-op. Similar to SkipCycle, this design ensures fast clusters in their system are not delayed by slow clusters. However, proposing empty cycles in slow BGs can be ineffective in our system, because this further reduces the throughput of the slow BGs.

Another key difference is that our system operates in a Byzantine environment, while both previous papers [21, 13] are describing systems in a crash-stop environment. Unlike RCanopus, the previous systems do not need to prevent malicious servers from arbitrarily skipping cycles in their protocol. As a result, the SkipCycle design is more complex than the skipping mechanisms in previous systems. In Section 3.3.2 and 3.3.3 we discuss how to skip RCanopus cycles safely.

Another family of consensus protocols is the Byzantine Fault Tolerance consensus protocols. In a Byzantine environment, a failed node may be malicious, and can perform arbitrary actions. The BFT consensus protocol is a crucial component in most blockchain systems, because blockchain systems are decentralized and it is more likely to have malicious nodes in the system.

PBFT [3] is based on Viewstamped Replication [26], but it can handle Byzantine failures. With $3f + 1$ nodes, PBFT can handle at most f Byzantine failures in the system. More details on the relations between Viewstamped Replication and PBFT can be found

in [19]. One drawback of PBFT is its poor scalability. With two rounds of all-to-all broadcasts, the number of messages sent in every PBFT cycle is $O(N^2)$.

SBFT is a BFT consensus protocol that is designed for large scale deployments. The design is partly based on PBFT [3] and Zyzzyva [16]. SBFT [12] addresses the scalability issue of PBFT [3] by adding *collector* nodes into the system. Instead of all-to-all broadcasts, all nodes send their messages to the *collector*, which then relays them to all nodes. With this optimization, the message complexity of every cycle is reduced to $O(N)$. However, SBFT still requires $O(N^2)$ messages in its view change protocol, which is used to recover from leader failures. In our implementation, RCanopus uses SBFT [12] to provide BFT consensus because it had state-of-the-art scalability when we started this project.

HotStuff [35] improves upon SBFT, achieving linear message complexity in the view change protocol. When the leader is not faulty, it also has the *responsiveness* property, which means the latency of the protocol is related to the actual network delay, as opposed to maximum network delay.

Table 2.1 summarizes the main difference between previous protocols.

Protocol name	Improves upon	Changes
Paxos	–	–
Viewstamped Replication	–	–
Fast Paxos	Paxos	Reduces commit latency by bypassing the leader
EPaxos	Paxos	No designated leader
Raft	–	More understandable than Paxos
PBFT	Viewstamped Replication	Tolerates Byzantine failures
SBFT	PBFT	Avoids all-to-all broadcasts in commit path
HotStuff	SBFT	Avoids all-to-all broadcasts in leader election

Table 2.1: Summary of consensus protocols

2.2 Blockchains

A large quantity of research work [7, 34, 10, 1] has emerged in the area of blockchains since the concept was first proposed in Bitcoin [24]. Blockchain systems often operate in Byzantine environments, thus they use BFT consensus protocols to reach agreement on the next block to be added to the existing chain.

Blockchain systems can be divided into two categories: permissioned blockchains and permissionless blockchains. The former relies on external methods to keep the membership

of participants. When a server is added or removed, the entire system must agree upon the change. Permissionless blockchains have open memberships, allowing any node to join or leave the system freely.

Permissionless blockchains need to have mechanisms to handle Sybil attacks, where one user impersonates multiple identities. If the Sybil attacks are not handled, one malicious user can introduce arbitrary number of malicious participants into the system, easily breaking the BFT consensus protocol. Most permissionless blockchains, including Bitcoin, use Proof-of-Work (PoW) to prevent Sybil attacks. Under PoW, the aggregate vote power of an entity is proportional to the computing power it controls, rather than the number of accounts. However, PoW has very poor performance and consumes a significant amount of energy. Bitcoin can achieve a throughput of 7 transactions per second, with an end-to-end latency of over 10 minutes [5]. As a cryptocurrency, the performance of Bitcoin is unable to meet people's daily trading needs.

Researchers have made several attempts to remove PoW in blockchain systems. Kiayias et al. [15] proposed a blockchain protocol based on Proof-of-Stake (PoS), which means the probability of any node becoming the leader is proportional to its stake, i.e. wealth. Unlike PoW, PoS does not require nodes to participate in heavy computation, such as solving a cryptographic puzzle. Therefore, the cost of leader election in PoS is much smaller compared to PoW. The logic behind PoS is that the main stakeholders in the system do not have incentive to attack the system. Once the attackers successfully perform a malicious operation in the system, the system will soon become valueless, which hurts the main stakeholders more than other users. Algorand [10] is another example of a PoS system. The main drawback of PoS is that it is mainly applicable to cryptocurrency systems. In other types of blockchain systems, it may be difficult to use PoS. For example, a blockchain can be used to trace the products in a supply chain, but it is hard to define the stake of a node in this system.

Another alternative to PoW is to control the membership of the blockchain system, turning it into a permissioned blockchain. Since all server nodes need permission to join the system, we can rely on the external membership controlling mechanisms to prevent Sybil attacks. One of the most popular permissioned blockchain is HyperLedger Fabric [1]. It is an open source blockchain platform that aims to provide blockchain service that satisfies industry needs. HyperLedger Fabric (HyperLedger) has a unique Execute-Order-Validate model, as opposed to the Order-Execute model in other blockchain systems. Fabric transactions are executed before they are sent to the *orderers*. After ordering, if a transaction has execution conflicts with previous transactions, it will be aborted in the validation phase.

The ordering service of HyperLedger Fabric is made replaceable, allowing it to leverage various consensus protocols. When Fabric [1] was published, it used Apache Kafka to provide CFT consensus among *orderers*, instead of BFT consensus. The author made this choice to maintain good performance, since Kafka outperformed existing BFT consensus protocols. As a consequence, the *orderer* machines must not be malicious. This weakens the safety properties of the system. RCanopus was designed to serve as a practical BFT ordering service for HyperLedger Fabric, because it scales well and is topology-aware in a geo-distributed deployment.

Sousa et al. [31] replaces the Kafka ordering service in HyperLedger Fabric with BFT-SMART [30]. Similar to RCanopus, BFT-SMART also targets geo-distributed deployments. However, BFT-SMART is optimized to achieve lower latency, while RCanopus is optimized to achieve higher throughput.

In 2019, Gorenflo et al. [11] managed to improve the throughput of HyperLedger Fabric from 3,000 tps to 20,000 tps. Their main contributions include improving parallelism and caching on transaction validation service, and separating transaction metadata before ordering. Since they used Apache Kafka to provide the ordering service, we expect a drop in throughput when it was replaced with a BFT consensus protocol, such as RCanopus. From their work, we can learn that the other components of HyperLedger Fabric can operate at a speed of 20,000 tps. Unlike the consensus protocol, these components do not need coordination across multiple machines, so their performance is not sensitive to network conditions.

2.3 RCanopus

RCanopus [14] is a BFT consensus protocol that aims to provide high throughput in a geo-distributed environment. Similar to Canopus [29], it organizes protocol participants into a hierarchical structure. This enables RCanopus to take advantage of the underlying network topology, as bandwidth is typically more limited on wide area links than on regional links. The hierarchical structure allows RCanopus to break down system-wide consensus into consensus within smaller groups, namely SuperLeafs and Byzantine Groups. The consensus protocols in these smaller groups can be executed in parallel. This can greatly improve the performance because the consensus protocols can finish much more quickly in smaller groups.

As a tradeoff, the fault tolerance properties are weaker than non-hierarchical BFT consensus protocols such as PBFT [3] and SBFT [12]. For example, with 16 nodes, a PBFT

protocol can tolerate 5 arbitrary Byzantine failures. However, a RCanopus deployment with 4 BGs and 16 SLs can only tolerate 1 Byzantine failure in each BG. When deploying RCanopus, the system administrator should be careful to make different SLs independent from each other. For example, if we use a cloud deployment, we should deploy different SLs in different cloud service providers. If this is done correctly, it should be very rare for many SLs in the same BG to fail at the same time.

RCanopus requires synchronous networks inside datacenters. The FLP impossibility result [8] proves that it is impossible to maintain both safety and liveness in an asynchronous network environment. Therefore, in order to preserve safety and liveness, RCanopus requires weak network synchrony across datacenters. This means RCanopus assumes that there is a finite upper bound on the message delay across different datacenters [14]. When this weak synchrony condition is not met, RCanopus may stall in the presence of network partitions, giving up liveness.

Before starting the BFT consensus protocol, RCanopus batches client transactions with a CFT consensus protocol inside each SL. Compared to reaching a BFT consensus on every individual transaction, committing transactions in batches helps improve the throughput of RCanopus.

In RCanopus, the execution flow of different cycles can overlap, forming a pipeline. Traditional BFT protocols use state machine models that commit transactions sequentially. This change allows RCanopus to more efficiently utilize network resources, especially on high latency links.

2.3.1 Hierarchical structure

In RCanopus, the server machines are organized into a hierarchical structure. We divide the machines into Byzantine Groups, which are then subdivided into SuperLeafs. A BG contains several SLs that are geographically proximate. For example, in global deployments, all machines inside a BG should be located in the same continent.

In order to provide proper Byzantine Fault Tolerance, different SLs in the same BG should be deployed in different datacenters. If the system is deployed in the cloud, different SLs should be deployed across different cloud service providers.

Figure 2.1 is taken from Canopus [29]. It illustrates a Leaf-Only Tree, which is the way RCanopus organizes machines in a deployment. In a Leaf-Only Tree, each physical node can represent itself, or any of its ancestor nodes. In RCanopus, we have two levels of virtual nodes: SuperLeaf and Byzantine Group. Thus, a physical machine in RCanopus may emulate its local SL and local BG.

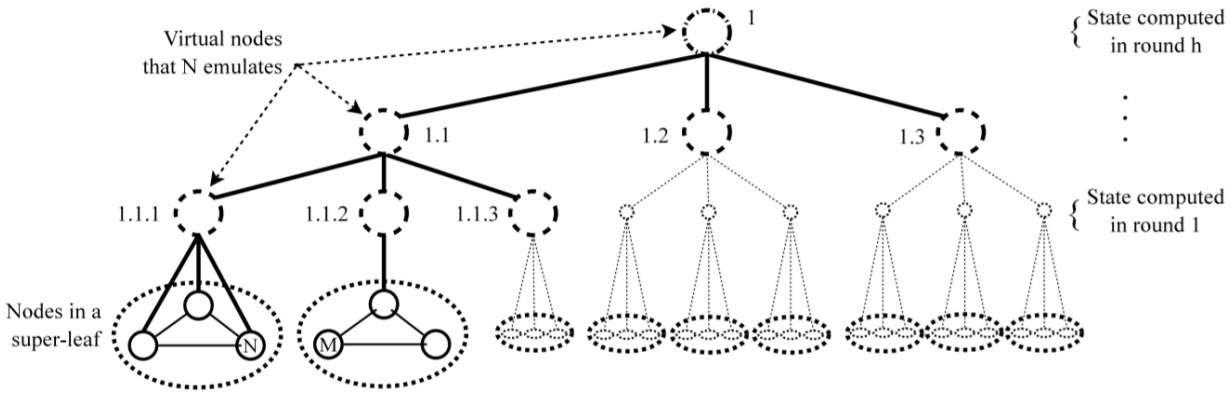


Figure 2.1: The structure of a Leaf-Only Tree (LOT). Full circles represent physical nodes. Dotted circles represent virtual nodes.

The Leaf-Only Tree provides a hierarchical layout that allows us to reduce the number of participants of upper level protocols by using delegation. For example, in RCanopus, only the leader of each SL participates in the BFT protocol. When the BFT protocol finishes, the leader of each SL replicates the result within its own SL.

2.3.2 Protocol

Like most consensus protocols, RCanopus executes in cycles. In each cycle, the system reaches consensus on a set of transactions. A cycle in RCanopus consists of three rounds[1]. If the system needs to tolerate crash failures of an entire BG, an additional membership service is required after Round 3 to agree on the membership of BGs.

Round 1

Round 1 achieves consensus on a transaction block within an SL. The clients submit transactions to RCanopus servers, and Round 1 runs periodically to commit them in a Crash Fault Tolerance (CFT) consensus protocol. The transaction block is created by concatenating client transactions. Round 1 can also start if the number of pending transactions exceeds the max number of transactions in a block.

[1]Note that *round* and *cycle* have different meanings in RCanopus. Different rounds are different phases in a cycle. This is how the terms are used in the RCanopus whitepaper.

Round 1 should elect a leader for the SL to participate in Round 2. Some consensus protocols, like Raft [27], already have built-in mechanisms to elect a leader. Once a leader has been elected, it should remain as the leader until it fails, in order to reduce the leader election cost.

Round 2

Round 2 achieves consensus on one or more transaction blocks within a BG. These transaction blocks are submitted by the leaders of SLs. They are committed using a BFT consensus protocol that runs on all SL leaders. The transaction blocks committed in one BFT round is included in a BFT result, which should also contain a valid quorum certificate. A quorum certificate contains digital signatures of enough machines to prove these machines have agreed on the message. In SBFT, a valid quorum certificate must contain signatures from at least $2f + 1$ machines, where f is the number of Byzantine failures the protocol can handle. Round 2 also assigns an RCanopus cycle number to each committed BFT result.

Most BFT consensus protocols elect a leader among its participants, which is responsible for starting a consensus cycle. To achieve high throughput, a new Round 2 starts immediately after the previous one finishes, as long as the BG leader still has pending transaction blocks. By default, a BG leader is also an SL leader.

After the SL leaders learn a BFT result, they replicate the result using the CFT protocol within the SL. After every SL finishes this replication, all machines in the BG will have the same result for this RCanopus cycle.

Round 3

In Round 3, each SL communicates with SLs in remote BGs to fetch their BFT results committed in Round 2. Within each SL, some machines are selected by the SL leader to perform this fetching operation, and they are called **representatives** of their local BG. When contacted by a representative, a node becomes an **emulator**, because it is emulating the virtual node of its BG in the LOT model. The emulator then responds with its local BFT result when it is available.

Note that all SLs in the same BG fetch from remote SLs independently. There is no BG-level coordination in Round 3. After a BFT result is successfully fetched, it is replicated within the SL.

Unlike Round 1 and Round 2, a machine can execute the Round 3 of multiple cycles at the same time. It is possible for a machine to fetch cycle 10 from a remote BG, and fetch cycle 14 from another one. When an SL has received the BFT results of a cycle from all BGs, this cycle is considered finished on this SL. The maximal number of concurrent cycles in Round 3 is a system configuration parameter.[2]

2.3.3 Integration with blockchain applications

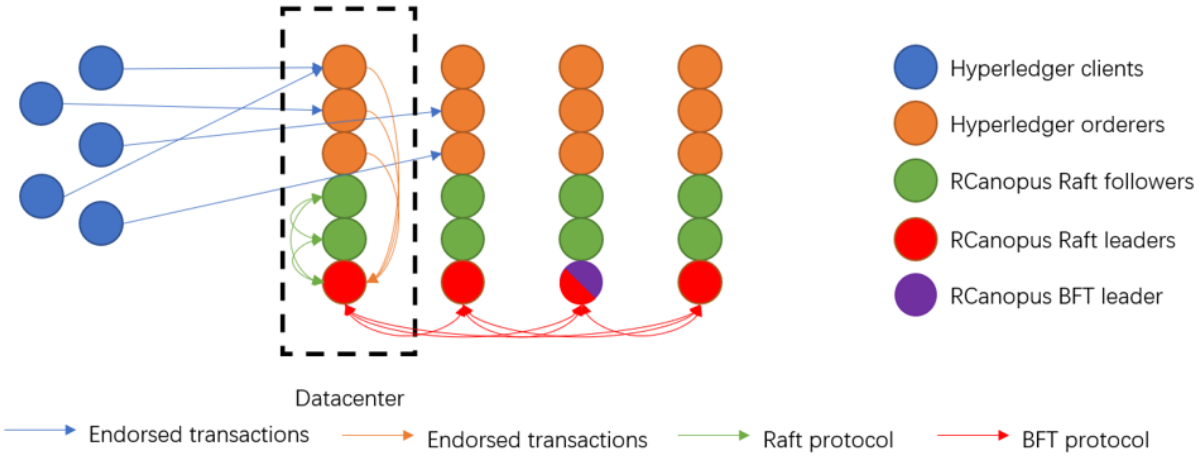


Figure 2.2: An RCanopus Byzantine Group deployed in 4 datacenters. The entire system is deployed as an ordering service for HyperLedger Fabric.

RCanopus has been designed to serve as an ordering service for HyperLedger Fabric, which is one of the most popular open source blockchain platforms. In this section, we illustrate how RCanopus can be used in HyperLedger Fabric.

Figure 2.2 shows a partial HyperLedger Fabric deployment with RCanopus as its ordering service. This diagram only shows the deployment in one geographical region, so we can only see one RCanopus Byzantine Group in the diagram.

HyperLedger Fabric uses an execute-order-validate model. The client transactions are first executed by the endorsing peers, which register the execution results in the endorsement. The transactions are then sent to the orderers, which verify the endorsements and

[2]This parameter is also referred to as **pipeline depth**.

forward them to the ordering service, to reach a consensus on the order of transactions. For simplicity, we omit the endorsement step in Figure 2.2. In every datacenter, we deploy one SL and several orderers. The orderers act as clients in RCanopus, and forward client transactions to RCanopus servers within the local datacenter. RCanopus will reach a consensus across all BGs and return the results to the committing peers in HyperLedger Fabric. If possible, we recommend system administrators to deploy the orderers, RCanopus servers and committing peers on the same rack inside each datacenter.

In a HyperLedger Fabric system with RCanopus as its ordering service, the entire workflow consists of 11 steps:

1. The client submits a transaction to HyperLedger Fabric **endorsers**.
2. **Endorsers** execute the transaction and send the endorsed transaction back to the client.
3. The client submits the endorsed transaction to a HyperLedger Fabric **orderer**.
4. The **orderer** validates the signatures from endorsers, and sends the endorsed transaction to an RCanopus server.
5. The transaction is inserted into a transaction block and replicated within an RCanopus SL.
6. The SL leader submits the transaction block to the BG leader, and waits for it to be committed in the BFT protocol.
7. The SL leader learns the committed result of the BFT protocol, and replicates it within the SL.
8. The SL leader selects representative machines to fetch the BFT results from remote BGs. Meanwhile, all machines in the SL also responds to fetching requests from remote BGs. The RCanopus cycle ends when the SL has the BFT results from all BGs.
9. To reveal the result of a cycle, RCanopus may wait for a few more cycles to complete. When it is ready, the **orderer** can learn the result from any RCanopus server in its own datacenter.
10. The HyperLedger Fabric **orderer** forwards the transactions to a **committing peer**.
11. The transactions in an RCanopus cycle are committed, the client can observe them on the blockchain.

Chapter 3

Protocol Extensions

In this chapter we introduce several extensions for RCanopus. They are proposed to address several design issues of RCanopus. We first present an approach to remove duplicate transactions. Duplicate transactions are unavoidable due to resending mechanisms, and it is very important for RCanopus to recognize them. Next, we introduce a design change on the inter-BG communication pattern, which helps reduce the bandwidth usage on wide area links. Our third extension, SkipCycle, mitigates the performance impact when we have straggler BGs. Due to different hardware and different network conditions, it is very common for BGs to complete cycles at different speed in a real deployment. SkipCycle allows every BG to make progress at its own pace, without having to wait for the slowest BG. Our fourth extension, BG envoy, is an approach to improve the scalability of RCanopus by reducing inter-BG communication and increasing intra-BG communication. When there is abundant intra-BG bandwidth and limited inter-BG bandwidth, this approach can increase the throughput of RCanopus. Finally, we introduce a way to handle BG failures. Although the design is slightly inefficient when compared to the described approach in the RCanopus whitepaper, it is safe and has all the details needed for implementation.

3.1 Transaction Deduplication

In the RCanopus whitepaper [14], transaction deduplication is briefly discussed in Section 5.4.1 . This covers the duplicate transactions submitted by a client to different SLs in the same BG, which is a result of the client side resending mechanism. The solution proposed in the whitepaper is to cache previous transactions in SL leaders, and remove any duplicate when executing the BFT protocol. However, this is not feasible because

RCanopus batches client transactions into transaction blocks before submitting them in SBFT, which is used to reach a BFT consensus within a BG. A transaction block may contain thousands of transactions. If the SBFT servers detect the same transaction in two different blocks submitted by two different SLs, it is infeasible to reject an entire block because of duplicate transactions. It is also impossible for SBFT participants to remove individual transactions from a block, because the transaction blocks are protected by digital signatures from SL leaders. Any modification to the batches would lead to a signature verification failure. Therefore, if the client submits the same transaction to multiple SLs, RCanopus must accept them and commit them in Round 2. The duplicated transactions committed in Round 2 should be handled at a higher level.

3.1.1 Design

When we were building RCanopus, we did not include a transaction deduplication mechanism because we intended to build our system as an ordering service for HyperLedger Fabric [1], in which the committing peers are responsible for dealing with duplicate transactions. However, our system should also work in more general environments. Therefore, we have developed a protocol extension to handle duplicate transactions.

In RCanopus [14], client transactions are identified by their unique client-generated nonces. We do not try to detect malicious clients that assign the same nonce to different transactions. If a client sends a transaction for the second time to our system with a different nonce, then it is treated as a completely new transaction.

In order to detect and remove duplicate transactions, we need a timestamp on every client transaction. This requirement is consistent with HyperLedger Fabric. The BFT protocol in Round 2 should also add a timestamp on the Preprepare message, which is assigned by the BG leader. Once a cycle is committed in Round 2, the timestamp will be protected by a quorum certificate. Note that the timestamp records the time when Round 2 starts, not when Round 2 finishes. We define a time limit t_1, e.g. 1 second, and if the difference between the timestamp of a transaction and the timestamp on the committed BFT result is larger than t_1, then this transaction is invalid and should be excluded when transactions are executed. If a transaction cannot be submitted to Round 2 within t_1, then this transaction is considered lost and the client must resend it later. By defining t_1, we can limit the history in which we search for duplicated transactions to a finite number of cycles.

For duplicate transactions that are submitted to Round 2 with a time difference smaller than t_1, they are considered the same transaction. When executing transactions, we can

keep a buffer of the hash values of recent transactions to remove any duplicates. We define t_{diff} to be the minimum time difference between any two BFT cycles. t_{diff} should be chosen to be small enough so that it does not affect normal BFT performance. The machines executing transactions committed in RCanopus should keep track of all transactions committed in (t_1/t_{diff}) cycles and avoid executing the same transaction twice within this range.

When a transaction cannot be committed in time, the client may consider it lost and resend it. Currently, clients wait for a fixed period of time before resending a transaction. A more precise way would be to wait for (t_1/t_{diff}) cycles. In the current design of RCanopus, there is a parameter K controlling pipeline depth. The servers can only reveal the result of cycle X to clients if cycle $X + K$ has finished. Therefore, if the client learns that the latest cycle number is X when it submits the transaction, the last cycle that can legitmately include this transaction is $X + 2K + (t_1/t_{diff})$. If this transaction is not committed in any cycle before $X + 2K + (t_1/t_{diff}) + 1$, the client should resend it.

In order to maximize the efficiency, the clocks on all servers should be as synchronized as possible. When all clocks are perfectly synchronized, the the machines executing transactions need to cache the transactions for (t_1/t_{diff}) recent cycles. If the clocks are not synchronized, they should keep a longer record, taking the time errors into account, in order to prevent transaction replay attacks. This will cause higher memory consumption.

3.2 Pull-Only Round 3

In RCanopus [14], after each BG finishes Round 2, all SLs fetch the transactions from remote BGs independently in Round 3. The representatives are the machines inside a SL that are selected to perform the fetching operation. The emulators are the machines contacted by the representatives in the remote BG. Each representative selects one emulator randomly from the remote BG.

RCanopus originally uses a push-pull model for this communication. When a representative machine in a BG tries to fetch the transactions from an emulator in a remote BG, it sends its own transactions first. The emulator will only reply with its transactions after receiving and validating the transactions from the representative. It is important for the representative to provide its transactions committed in Round 2, because we want to prevent malicious representative machines to constantly perform expensive fetching operations across BGs. When an emulator receives the transactions from the representative, it also replicates them inside its SL.

Consider the case where, in cycle 10 the representative R of (BG1, SL1) contacts the emulator machine E in (BG2, SL1). R will first send the transactions committed by BG1 in Round 2 of cycle 10. When E receives this message, it will verify the validity of the message, then replicate it within (BG2, SL1). If (BG2, SL1) has not started to fetch from BG1, then it will not initiate such an operation in cycle 10 because the transactions of BG1 has been replicated. After that, E will reply with the transactions committed by BG2 in Round 2 of cycle 10. R will check the validity and replicate this message in (BG1, SL1).

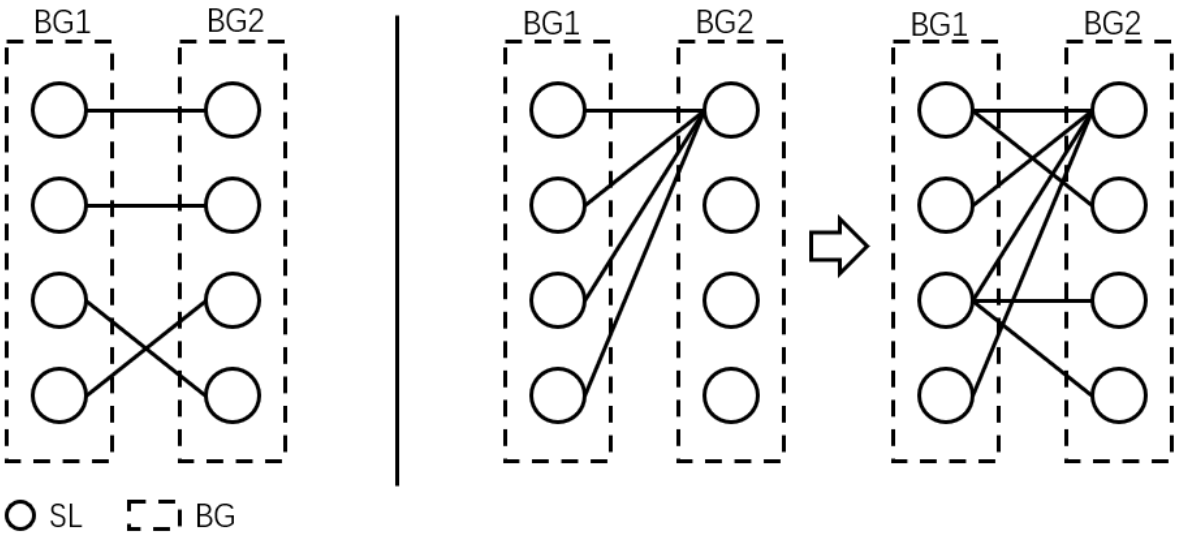


Figure 3.1: Best case and worst case of the communication pattern in Round 3

In Figure 3.1 on the left side we show the optimal case of this communication pattern between 2 BGs. Suppose all fetching operations are initiated by the BG1. In this case all 4 SLs happen to contact different SLs in BG2. As a result, 4 copies of the transactions from BG1 and 4 copies of the transactions from BG2 are transferred on the WAN links. On the right hand side of the figure, we show the worst case where all 4 SLs in BG1 contact the same SL in BG2. This is possible because every representative machine in BG1 chooses a random emulator in BG2 to contact. The remaining 3 SLs in BG2 have to fetch from BG1. 7 copies of the transactions from both BGs are transferred on the links.

Suppose we have n SLs in both BGs, in the best case we need to transfer the transactions n times, whereas in the worst case we need to transfer $2n-1$ times. We can see that in the worst case, our system consumes nearly 2x WAN bandwidth. And we cannot prevent this from happening, because all SLs operate independently without coordination in Round 3. In order to reduce the bandwidth usage, we replace the push-pull pattern with a pull-only

pattern, where the representative machines do not send their result to the emulators. As a result, the transactions of both BGs only need to be transferred n times, which is equal to the best case performance in our previous approach.

This change is non-trivial because we need to prove the representative has finished Round 2 without sending the actual transactions. In Section 3.3 we will explain how this is done. In Section 3.4 we further reduce the bandwidth consumption. With this extension, the transactions only need to be transferred once between any two BGs.

3.3 SkipCycle: Decoupling the progress of different BGs

SkipCycle is an extension designed to improve the throughput of RCanopus when different BGs finish Round 2 at different speeds. The slow BGs are allowed to skip some RCanopus cycles to catch up with the fast BGs, while the fast BGs can still make progress at their own paces.

A similar idea has been proposed in *Scalable agreement: Toward ordering as a service* [13] (HotDep'10) and Mencius [21] (OSDI'08), where participants of the consensus protocol can skip some stages of the protocol. In both papers, the servers skip by proposing no-op. This is different from the SkipCycle design but more similar to our previous implementation, where slow BGs are asked to commit empty cycles.

3.3.1 The importance of SkipCycle

In this section we will show that it is very common for different BGs to execute a BFT protocol at different speeds, and we will explain why this could impact our performance.

Let us first take a look at the factors that can affect the performance of Round 2. Usually throughput and latency are two different metrics for the performance of a BFT protocol. However, in our current design the BFT cycles do not overlap, so higher latency will lead to lower throughput. They can be affected by the CPU performance of the machines as well as the latency and bandwidth of the network. These parameters are very likely to be different across all machines in a deployment. For example, in a real world deployment we deploy each SL in a datacenter, so the latency between them will be different.

Our experiment results show that we often have one slow BG in the system, and other BGs have to wait for it to complete Round 3. For example, the slow BG may be executing

SBFT for cycle 15, but the other BGs are already trying to fetch the results for cycle 22. Since multiple cycles can execute their Round 3 in parallel, the fast BGs can still process transactions, but those transactions will not count towards the system's throughput until Round 3 is finished. The slow BG now becomes a bottleneck for the entire system.

Before we had SkipCycle, our solution was to start empty cycles on straggler BGs, because empty cycles can be processed in the BFT protocol faster. However, this is not effective, because empty cycles are bubbles in the pipeline, and they only make the stragglers fall behind even more. Suppose BG 1 is processing Round 2 of cycle 15 while all other BGs have finished Round 2 of cycle 20. Since BG 1 is slower than other BGs, it is likely to have more queued transactions, since we send transactions to all BGs at an equal rate in our experiments. If BG 1 commits 5 empty cycles to catch up with other BGs, it is wasting time while processing no transactions. The incoming transaction queue will grow larger in BG 1, and the other BGs are processing transactions normally. Therefore, when BG 1 finishes the 5 empty cycles, the number of pending transactions will become more unbalanced across all BGs. This forces BG 1, the straggler, to process more transactions in subsequent cycles, which leads to larger message size and longer latency in Round 2. SkipCycle allows the straggler BGs to catch up by skipping some of the RCanopus cycles. It decouples the BFT sequence number inside a BG and the RCanopus cycle number. As shown in Figure 3.2, previously every BG must execute one BFT cycle in the Round 2 of every RCanopus cycle. With SkipCycle, some RCanopus cycles may not have BFT cycles on the straggler BGs. As a result, they can skip ahead in cycle numbers.

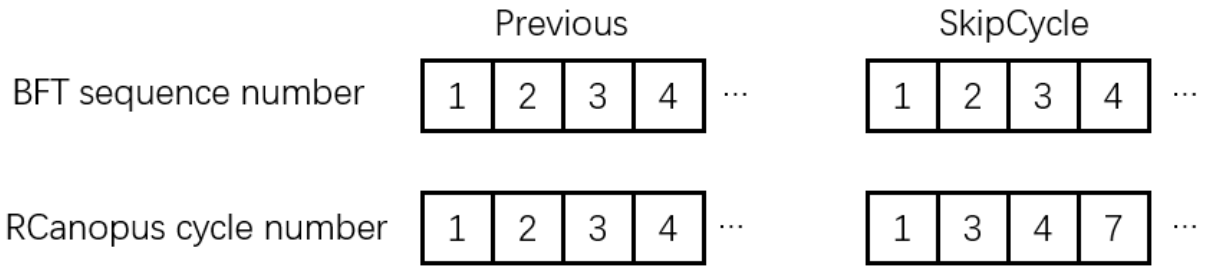


Figure 3.2: Mappings between SBFT sequence numbers and RCanopus cycle numbers

SkipCycle enables our system to fully utilize the processing speed on all BGs. The fast BGs can make progress without waiting for the slow BGs, and those committed transactions can be exchanged with the slow BGs. Therefore, SkipCycle is a very important feature of our system. In the following sections we will discuss the detailed design of SkipCycle.

3.3.2 SkipCycle design

Assigning cycle numbers in SBFT

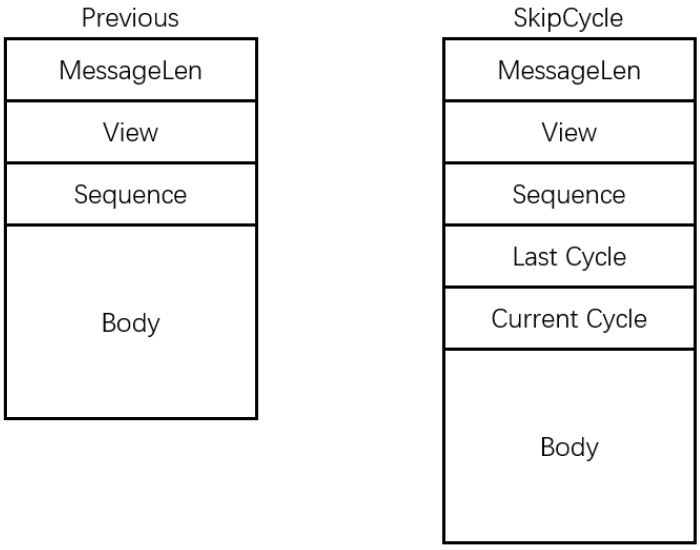


Figure 3.3: Change in Preprepare message structure

In RCanopus [14], an RCanopus cycle number is attached to a committed BFT result in Round 2. In Round 1, transactions are organized into blocks, but they are not associated with an RCanopus cycle number. When the SBFT leader includes a transaction block in a Preprepare message, it gives this message an SBFT sequence number. This sequence number is increased by 1 for every SBFT cycle. Before we introduced SkipCycle, this sequence number always equals to the RCanopus cycle number, so we do not need a separate field in the message header to record the RCanopus cycle number.

With SkipCycle, the sequence number in SBFT no longer equals to RCanopus cycle number, so we need to introduce additional fields in the header, as shown in Figure 3.3. Since this field is in the message header of a Preprepare message, it is provided by the SBFT leader. Obviously, we need to prevent a malicious leader from arbitrarily setting this field. This is done by requiring a **BFTCompletionProof** for every skip action. Details of **BFTCompletionProof** will be introduced in Section 3.3.3. A **BFTCompletionProof** proves that a remote BG has completed a cycle in Round 2, so it is necessary for the local BG to skip to this cycle. An SL will learn a remote BG has completed cycle c

when a representative from that BG contacts the SL to fetch its results for cycle c. If the local BG is falling behind the remote BG, the result for cycle c will not be ready, so the SL will submit its next transaction block with a **BFTCompletionProof** to its own BG. The SBFT leader validates this **BFTCompletionProof** by checking whether it has enough signatures from a remote BG. When one or more transaction blocks have valid **BFTCompletionProof** attached, the SBFT leader will set

$$currentCycle = max(lastCycle + 1, max\ requested\ cycle\ number)$$

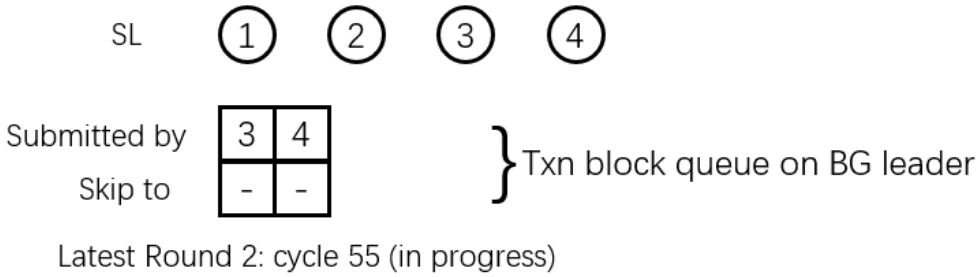
Note that due to network asynchrony and queueing delay at the SBFT leader, the max requested cycle number may be smaller than $lastCycle + 1$. If no SL requests such a skip action, then the SBFT leader must set $currentCycle = lastCycle + 1$.

Fetching BFT results in Round 3

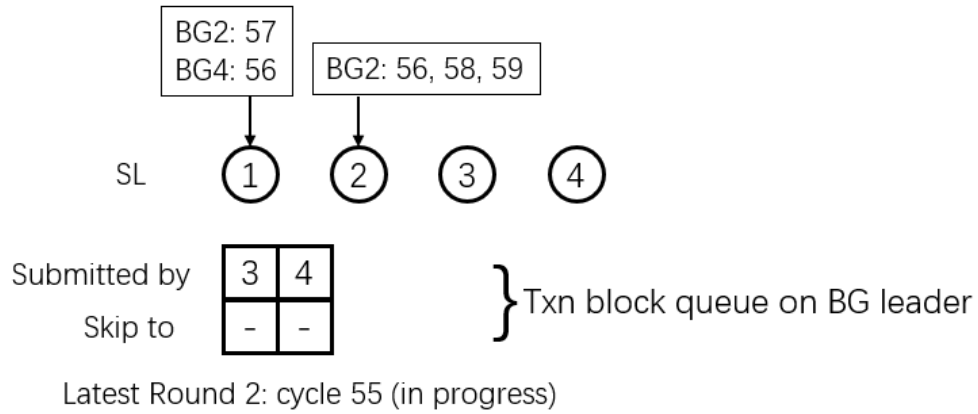
In Section 3.2 we mentioned that the representatives do not send their BFT results when they fetch from a remote BG. The representatives now need to send a **BFTCompletion-Proof**, in order to prove that they have finished Round 2. The emulator will receive this **BFTCompletionProof**, and it can be used to initiate a skip action in Round 2 if the emulator's BG is behind.

Responding to fetching requests in Round 3

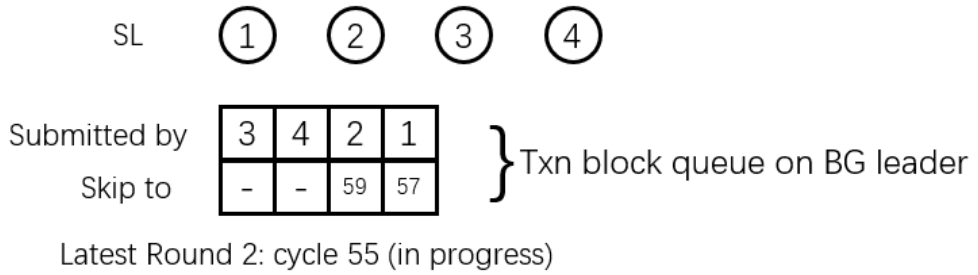
With SkipCycle, a BG may lack the BFT results for a cycle, because the Round 2 of this cycle have been skipped on this BG. When this happens, the BG effectively commits 0 transactions during Round 2 in this cycle. However, this is different from committing a message with 0 transactions via SBFT. Because the entire SBFT cycle is skipped, we do not have a quorum certificate to prove its validity. In this scenario, the **BFTCompletion-Proof** of the next non-skipped cycle on this BG is used as a proof for the skipped cycle(s). For example, consider BG 1 skipping cycles 5, 6, 7, then cycle 4 and 8 are two non-skipped cycles on BG 1. Cycle 5 to 8 is finished at the same time, and the **BFTCompletion-Proof** is used to prove cycle 5 to 7 are empty. This is because the **BFTCompletionProof** shows the *lastCycle* and *currentCycle* fields (see Figure 3.3) in the Preprepare message. In Round 3, when an SL is requested for a cycle that has been skipped in the local BG, it uses the **BFTCompletionProof** of the next non-skipped cycle as a response.



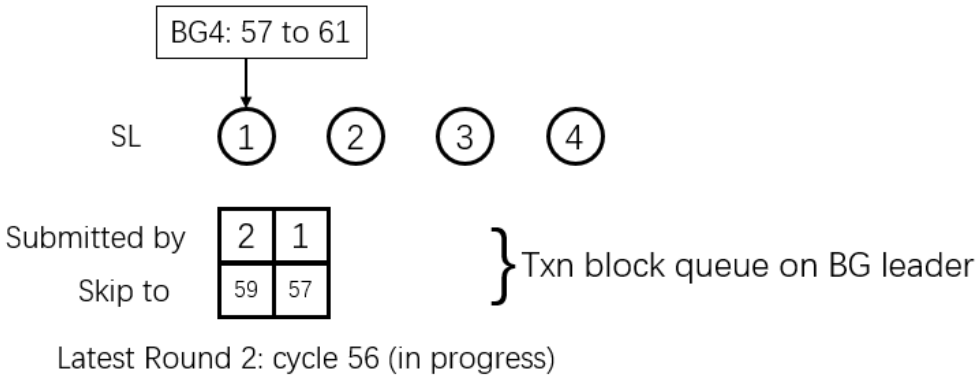
(a) SBFT is processing cycle 55.



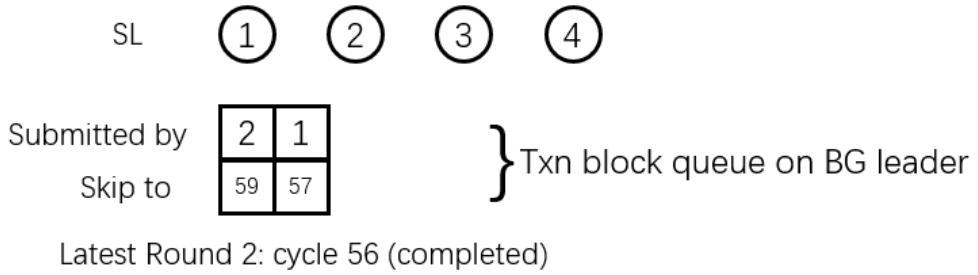
(b) SL 1 and 2 receive incoming fetching requests.



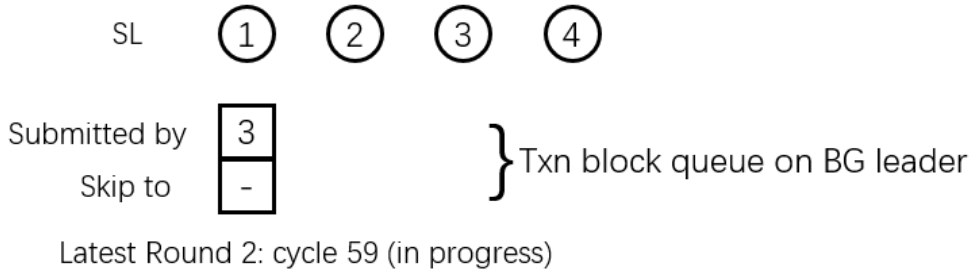
(c) SL 1 and 2 submit their transaction blocks with the BFTCompletionProof attachments.



(d) SBFT starts another cycle, processing transaction blocks submitted by SL 3 and 4. Since these two transaction blocks do not have BFTCompletionProof attachments, the new cycle number is 56.



(e) SL 1 cannot submit a new transaction block because its previous block has not been processed.



(f) SBFT starts another cycle, processing transaction blocks submitted by SL 1 and 2. The new cycle number is 59. Cycle 57 and 58 are skipped.

Figure 3.4: An example of the SkipCycle extension

An example of SkipCycle

Figure 3.4 demonstrates a concrete example of SkipCycle on a straggler BG. The initial situation is illustrated in Figure 3.4a, where BG1 is a straggler and it is currently processing Round 2 of cycle 55. There are 4 SLs (SL1 to SL4) inside BG1.

Figure 3.4b shows the scenario where SL1 and SL2 are contacted by representatives from remote BGs. Before BG1 finishes Round 2 of cycle 55, SL1 has received a request from BG2 to fetch the BFT result of cycle 57, and a request from BG4 to fetch the BFT result of cycle 56. SL2 has received 3 requests from BG2 to fetch the BFT results of cycle 56, 58, and 59. As shown in Figure 3.4c, when submitting the next transaction block to SBFT, SL1 will also submit a request to skip to cycle 57, and SL2 will submit a request to skip to cycle 59.

When BG1 finishes Round 2 of cycle 55, it proceeds to replicate transaction blocks submitted by its SLs. In Figure 3.4d, the leader of BG1 processes transaction blocks from SL3 and SL4 in the next BFT cycle. Since they do not have **BFTCompletionProof** attachments, BG1 treats them as cycle 56. Before BG1 finishes cycle 56, SL1 receives requests from BG4 to fetch the BFT results of cycle 57, 58, 59, 60, and 61. SL1 cannot immediately submit a request to increase the cycle number because it already has a pending transaction block at the SBFT leader. Therefore, the transaction block queue on the leader of BG1 remains unchanged, as shown in Figure 3.4e. This is a limitation of the SBFT implementation described in Section 5.3. SL1 will submit the request when its pending transaction block is processed by the SBFT leader.

Figure 3.4b to Figure 3.4e show that the **BFTCompletionProof** attachments submitted by SL1 and SL2 may not immediately affect the next cycle. Figure 3.4f shows the calculation of cycle number when **BFTCompletionProof** attachments are present in the transaction blocks. After BG1 finishes Round 2 of cycle 56, all emulators are able to respond to previous fetch requests for cycle 56, and the representatives in all SLs now start to fetch remote BFT results for cycle 56 . At the same time, the SBFT leader processes the transaction blocks submitted by SL1 and SL2. Both of them have valid **BFTCompletionProof** attached. SL1 requested to skip to cycle 57, whereas SL2 requested to skip to cycle 59. Since SL2 requested a cycle number larger than $lastCycle + 1$, the cycle number is set to 59. This cycle number applies to the entire Preprepare message, which includes both transaction blocks. Cycle 57 and 58 are now skipped on BG1.

When BG1 finishes Round 2 of cycle 59, the emulators in BG1 are able to respond to previous fetch requests for cycle 57, 58 and 59. For cycle 57 and 58, they will reply with the **BFTCompletionProof** of cycle 59, showing these cycles are skipped on BG1.

For cycle 59, they will send the response with the committed transactions as normal. The representatives in BG1 now start to fetch remote BFT results for cycle 57, 58, and 59 with the **BFTCompletionProof** of cycle 59.

3.3.3 An efficient way to prove Round 2 completion

In Section 3.2 and 3.3 we have mentioned that we need an efficient way to prove Round 2 is completed on a Byzantine Group. We need the message to be small enough so that it is cheap enough to be sent on a WAN link. We also need this **BFTCompletionProof** to show any skipped cycles between the last cycle and the current cycle on this BG.

In the original RCanopus design, the representative proves the completion of Round 2 by sending over the committed result of SBFT. The committed result contains a quorum certificate[1] which has at least $2f + 1$ SBFT servers in a BG that can tolerate f Byzantine failures. The emulator can then verify the quorum certificate, and the cycle number is stored inside the message.

With our SkipCycle extension, we have observed that the transactions are not needed to prove the completion of Round 2. We only need the quorum certificate and the *lastCycle* and *currentCycle* fields protected by the quorum certificate. A potential solution is to generate a separate quorum certificate only protecting the message headers. This is illustrated in Figure 3.5. The additional quorum certificate allows us to verify the validity of the message headers without reading the message body. Therefore, a representative can send the message header along with its quorum certificate as a **BFTCompletionProof** for the cycle. This proof message is small enough, as the message header plus the signature is smaller than 100 bytes. The problem with this approach is that it requires a lot of intrusive changes in SBFT to generate two quorum certificates in a single cycle. This will significantly increase the time required to implement the system.

To improve this solution, we further investigated the signature schemes SBFT. It is a very common practice to generate signatures on a hash of the original message instead of the entire message, and SBFT also follows this scheme. Since the hash is much shorter than the original message, it reduces the computational overhead when performing asymmetric cryptography operations. In cryptography, a hash function H have 3 main properties:

1. *Pre-image resistance* Given the hash result $h = H(x)$, it is difficult to find the original message x.

[1] A quorum certificate contains digital signatures of enough machines to prove these machines have agreed on the message. In SBFT, a valid quorum certificate must contain signatures from at least $2f + 1$ machines, where f is the number of Byzantine failures the protocol can handle.

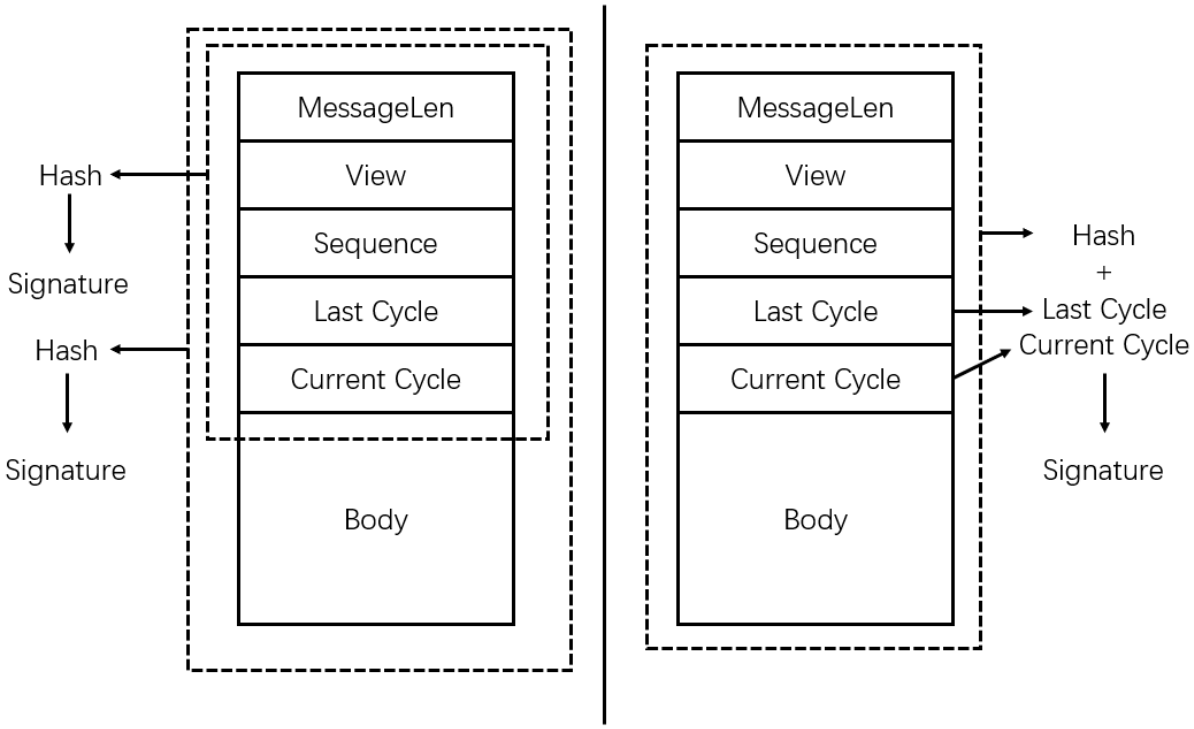


Figure 3.5: Two ways of generating certificates

2. *Second pre-image resistance* Given the hash result $h = H(x)$, it is difficult to find a different message y so that $h = H(y)$,

3. *Collision resistance* It is difficult to find two messages x, y so that $H(x) = H(y)$.

The hash function previously used in our implementation is SHA256. It is believed to provide all three properties of cryptographic hash functions. However, we do not need the pre-image resistance property, because the transactions are not confidential in our system. Since we are positioning our system as a component of a blockchain system, all the transactions committed in our system will be publicly available on the blockchain. Therefore, we can afford to weaken the pre-image resistance property of our hash functions.

Our second solution is illustrated on the right side of Figure 3.5. We extract the *lastCycle* and *currentCycle* fields from the committed BFT result and add them as two fields in the digest:

$$Digest(x) = SHA256(x) \mid\mid x.lastCycle \mid\mid x.currentCycle \tag{3.1}$$

Here $||$ is the concatenation operator. Compared to the old digest containing only the SHA256 hash, this new digest leaks some information about the original message, which might help the attackers reveal the original message. However, leaking information here is actually beneficial because we can learn the *lastCycle* and *currentCycle* fields of the original message from the digest, making it possible to use the digest and the signature as a **BFTCompletionProof**. This leakage in the digest causes no harm to our protocol, because the original message, including these two fields, will be exchanged in clear text. Compared to the SHA256 digest, our new digest is introducing 8 bytes extra length for the two int32 fields. The size of the entire **BFTCompletionProof** message is 72 bytes (40 bytes digest plus 32 bytes threshold signature).

A possible concern with this approach is that it sends a hash value without the pre-image. They argue that we should always recalculate hash values from original messages and never trust a hash value transferred from remote machines. This type of concern is unnecessary because the hash value itself is a message, and it should be well-protected by the asymmetric encryption algorithms.

Suppose we have a message M, and H is the hash value.

$$H = hash(M) \tag{3.2}$$

When we generate the signature with a key pair (K_{pub}, K_{priv}), we get:

$$sig = sign(H, K_{priv}) \tag{3.3}$$

Now if we send H and sig to another machine, then it can verify the signature (assuming it already has the public key).

$$fValid = verify(sig, H, K_{pub}) \tag{3.4}$$

Since the digital signature is generated on H, if a malicious node modifies H to H', it should be difficult to find a sig' such that $verify(sig', H', K_{pub})$ is true. This is because the owner of K_{priv} has never signed H', and forging a signature is guaranteed to be computationally hard under asymmetric encryption. In practice, the quorum certificates in SBFT [12] carry BLS [2] threshold signatures. Such a threshold signature can be a combination of many individual signatures, so it requires multiple keys to verify. However, the basic cryptographic properties are the same.

3.4 BG Envoy

In this section, we introduce a mechanism to reduce our WAN bandwidth usage. Instead of allowing every SL to gather results from remote BGs individually, we select envoy SLs in the BG to do the WAN data transfer. This allows us to send fewer copies of the Round 2 results from all BGs on the WAN links.

3.4.1 The importance of BG enovy extension

In a real world deployment, the network bandwidth across different geographical regions is usually much lower than the bandwidth of short distance links. From table 6.1 we can see that the bandwidth between different AWS regions are significantly lower than between different datacenters in the same region. Therefore, we should optimize our system to make the best use of available WAN bandwidth resources.

In RCanopus, there are currently 3 types of data transfer: intra-datacenter data transfer of Raft [27], inter-datacenter data transfer of SBFT [12], and inter-region data transfer in Round 3. Due to the hierarchical design, the 3 rounds in RCanopus use 3 different types of bandwidth resources. We call them local bandwidth, regional bandwidth and WAN bandwidth, respectively. In these 3 types of data transfer, only Round 3 uses WAN bandwidth, which is the most precious bandwidth resource in our system.

Since the Raft protocol in Round 1 only executes inside a SL, it only uses local bandwidth, which can be easily obtained by adding more hardware resources. Therefore we do not focus on optimizing the bandwidth usage in Round 1.

For Round 2 and Round 3, we calculate their theoretical bandwidth usage[2] in the original RCanopus design. For simplicity, we assume that each BG has the same number of SLs, and the transaction batch size is the same across all BGs. Suppose we have M BGs, each having N SLs. Suppose the size of a client transaction batch is T. We know that in the SBFT [12] protocol, the majority of the bandwidth consumption comes from broadcasting the Preprepare message. The subsequent messages in an SBFT cycle is irrelevant to the transaction size. In Round 1, client transactions are concatenated into transaction blocks. The maximum size of a transaction block is around 2 MB. The maximum size of the Preprepare message in Round 2 is set to around 4 MB. The other messages in SBFT are around 100 bytes long, so they can be neglected in the bandwidth usage calculation. Therefore, the bandwidth usage of an SBFT cycle is NT. The bandwidth used by Round 2 across all BGs is MNT.

In Round 3, each SL in a BG contacts every remote BG to fetch its BFT consensus results. Note that when an SL contacts a BG with $N = 3f + 1$ SLs, it should contact $f + 1$ SL because at most f SLs in the remote BG may be malicious and non-responsive. Since the results of Round 2 are protected by quorum certificates, the remote SLs cannot forge a BFT result. A malicious remote SL may not respond, or may respond with invalid messages. By contacting $f + 1$ SLs in the remote BG, we ensure that we can get at least one correct BFT result. To optimize towards the optimistic case, we contact the remote SLs one at a time. If this SL is non-responsive, we select other f SLs in the remote BG and contact them. Because an SL contains multiple machines, there is another system configuration parameter, called the number of representatives R, to control the number of machines executing this remote fetch. Having a larger R increases the system's ability to tolerate representative crash failures, which is caused by a crash failure inside the SL. In summary, there are R machines in an SL executing this remote fetch, and each of them contact $M - 1$ BGs in the optimal case. Now we can calculate the total bandwidth consumption for the optimal case.

$$bw_{best} = MNR(M - 1)T \tag{3.5}$$

In the worst case, each representative machine in the SL needs to contact $M(f+1)$ SLs in M remote BGs. Thus we can calculate the upper bound of total bandwidth consumption.

$$bw_{worst} = MNR(M - 1)(f + 1)T \tag{3.6}$$

[2]Here we calculate the bandwidth usage by calculating the size of data we need to transfer on the network links. In order to complete a cycle in a fixed time period, the bandwidth resource we need is proportional to the amount of data we need to send on the links.

From the above calculation, we have two observations. First, the bandwidth consumption in Round 2 is linearly proportional to M, N, and T, but the bandwidth consumption in Round 3 is proportional to M^2. In presence of SL failures, Round 3 bandwidth consumption is also proportional to N^2, since $N = 3f + 1$. This means the bandwidth consumption will rapidly increase when we deploy with more datacenters and more geographical regions, harming the scalability of the system. Second, the bandwidth consumption in Round 3 is actually larger than the bandwidth consumption in Round 2, since we have

$$\begin{aligned} bw_{best} &= MNR(M-1)T \\ &\geq MN(M-1)T \\ &\geq MNT \end{aligned} \tag{3.7}$$

We now consider a baseline approach where we use one global SBFT instance to achieve consensus across all MN SLs. The total size of client transaction batches would be MT, and it is broadcast to all MN SLs. Note that there are N SLs in the same BG as the global SBFT leader, so the SBFT leader only need to broadcast to $(M-1)N$ SLs on WAN links. Therefore, the total WAN bandwidth consumption of running a global SBFT instance can be calculated as:

$$bw_{baseline} = M(M-1)NT \tag{3.8}$$

If we compare it to formula 3.5, we can find that the WAN bandwidth consumption of RCanopus is comparable to the baseline only when we set R to 1. Otherwise, RCanopus would consume more bandwidth even in its optimal scenario.[3] So we can see that the current RCanopus design does not save WAN bandwidth when compared to running a global SBFT.

Now we have seen that the bandwidth consumption in Round 3 is higher than Round 2, but the available WAN bandwidth is much less than regional bandwidth. This can easily cause the WAN bandwidth to be a performance bottleneck in RCanopus. In the rest of Section 3.4 we will discuss the concept of BG envoy, which is introduced to tackle this problem by trading some regional bandwidth for WAN bandwidth.

3.4.2 BG envoy design

The basic design of the BG envoy extension is very simple. In this section, we first present the basic design, then discuss how to handle the case where envoys fail to fetch from remote BGs.

[3]This is also why we keep $R = 1$ in our experiments.

A BG envoy is an SL chosen in Round 2 to fetch remote BFT results in Round 3. A BG may select multiple enovy SLs in an RCanopus cycle. Inside an envoy SL, only the selected representative nodes perform the fetching operations. All nodes can be contacted by representatives in remote BGs, so it is still possible for any node to become an emulator. We define envoys as SLs instead of individual machines in order to keep the hierarchical design philosophy and remain compatible with the representative / emulator design in RCanopus [14].

During Round 2, the SBFT leader of a BG proposes the Preprepare message. At this time, it also chooses $2f + 1$ SLs to be the envoys for this cycle. Here N is the number of SLs in this BG, and f is the max number of Byzantine failures this BG can handle. From definition 4.1.3 we know that $N \geq 3f + 1$. When Round 2 finishes, every SL will learn the same set of envoys. If an SL is selected as an envoy for the current cycle, the representatives will be selected inside this SL as in the original RCanopus protocol.

After the representative(s)[4] in an envoy SL fetch the BFT results from remote BGs, the results are replicated inside the envoy SL. Then, the SL leader submits the results to the SBFT leader one by one. We call it a **positive response** when an envoy submits a remote BFT result to SBFT. These BFT results are labeled as messages from remote BGs to avoid confusion with local transaction blocks. They are replicated to all SLs with the SBFT protocol.

It is necessary to replicate the remote results within the envoy SL before submitting to the SBFT leader because only the SL leader has active connection with the BG leader. It is possible to batch all remote BG results and submit to the SBFT leader at once. However, this may lead to a very large message. Our current implementation sets the size limit of a single BG result to 4 MB. Suppose there are 4 BGs in the system, then batching all remote BG results can yield a message up to 12 MB. By replicating remote BG results individually, we can keep the maximum message size in SBFT and gRPC at 4 MB. Keeping messages at similar sizes can help us find the optimal TCP parameters for the experiments.

During Round 3, if an envoy $SL1$ fails to fetch the result of $BG1$, it will time out and submit a message to the SBFT leader, indicating the result of $BG1$ cannot be fetched. We call this a **negative response** message. Of course, the negative response message is created by the envoy and does not carry a quorum certificate of $BG1$. Therefore it is possible for a malicious envoy to forge a negative response. Therefore, the BG leader will replicate a negative response only after it receives negative responses from $f + 1$ different

[4]There may be multiple representatives selected in an SL. It is a configuration parameter of RCanopus. However, setting it to any number larger than 1 would significantly impact the performance, as previously discussed in Section 3.4.1

envoy SLs against the same remote BG. Like all messages submitted to SBFT, this negative response message should carry a signature of the sender SL, which is the envoy SL in this case.

3.5 New Global Membership Service

In this section we will discuss how to develop a global membership service for RCanopus [14]. The global membership service is used to maintain a list of active, non-faulty BGs. The global membership service is executed by every BG leader, and it is consulted after each BG finishes Round 3.[5] When the BG elects a new leader, the global membership service is also transferred to the new leader. In presence of BG failures, the global membership service should use a protocol to generate consistent membership output for inquiries from all BGs, even if the different BGs cannot communicate with each other.

We have found some flaws in the global membership service design in the RCanopus whitepaper, so we have developed a new global membership service for RCanopus. Our new GMS is not as performant as the original design in the whitepaper, but it is easier for us to analyze its correctness. For convenience, in the rest of this thesis, we refer to our new GMS as **SIMPLE-GMS**.

3.5.1 Why GMS is important

The output of RCanopus is controlled by both the transactions committed in every BG and the BG membership, which determines a list of active BGs for every cycle. In RCanopus [14], we assume that no BG is malicious. If an entire BG becomes malicious, then it may commit forged transactions, and other BGs cannot detect this failure, which is beyond the fault tolerance scope of RCanopus. The membership service handles BG crash failures. This can be caused by enough SL failures inside a BG, or a network partition between two or more BGs. In the latter case, every BG will treat the unreachable remote BGs as crashed, and the results from the unreachable BGs will be excluded from the current cycle.

In RCanopus, SLs are hosted in different datacenters and a BG consists of multiple SLs in a geographical region. We also suggest our users to deploy the SLs across different cloud service providers to achieve heterogeneous hardware / software configuration. When a BG

[5] A BG finishes Round 3 when it finishes fetching Round 2 results from all other BGs. A fetching operation finishes when the Round 2 result is successfully fetched, or the fetching times out. The maximum time for a fetching operation is a system configuration parameter.

crash failure happens, there are either too many failed datacenters in a region or a system-wide network partition that separates one or more BGs from others. Such an incident is rare but not impossible. In October 2018, a network partition between GitHub's clusters caused a service degradation of over 24 hours [33]. Therefore, the global membership service is a crucial component in a real world deployment.

3.5.2 Flaws in the original membership service design

In the RCanopus [14] whitepaper, two ways of managing the BG membership were proposed. One of them is the convergence module (CM). The convergence module has a more complicated design and is only briefly discussed in the main text. The other method requires a BFT protocol running on all BG leaders. This method is discussed extensively throughout the whitepaper, and for convenience we call it **BFT-GMS**. In this section, we will describe several flaws in the BFT-GMS design, in order to demonstrate the need for a new GMS design.

Summary of the BFT-GMS design

The BFT-GMS achieves a consensus among all BG leaders with the SBFT [12] protocol. All cycles need to have a valid membership committed by the global membership service. RCanopus [14] protocol states that different BGs are deployed in different geographical regions. Running SBFT across all regions in every cycle will become the performance bottleneck for our system. The solution in RCanopus is to cache the latest membership at the BG leaders, and a new membership consensus is only needed when one or more BG cannot contact all BGs included in the latest membership. This means a BG is allowed to reuse a stale membership if it can get the results from every BG included in this membership result.

Periodically, all BGs stop processing new transactions and synchronize their progress. At this time all BGs will reach a new consensus on the membership, and they can add new BGs or previously failed BGs into the system. For instance, the BGs can be configured to synchronize every 500 cycles. Between two synchronization points, the membership service can only remove BGs from the membership. Once a BG has been removed, it can only be added back at a future synchronization point.

Flaws in the BFT-GMS design

The first flaw is that BFT-GMS greatly weakens the fault tolerance property of the system. The BFT-GMS relies on the SBFT protocol executed on all BG leaders to achieve consensus. Note that BG leaders can be malicious. In order to tolerate f malicious nodes among the BG leaders, there must be at least $3f + 1$ BGs in the system. In the RCanopus whitepaper, this is added as a separate condition of the safety property. However, this condition severely weakens the fault tolerance of the system, because SBFT [12] cannot prevent a malicious node from becoming the BG leader. Even detecting a malicious BG leader is impossible, since it can behave normally in Round 2 and Round 3 but perform malicious actions in the membership service.

For example, when we have 4 BGs in the system and 10 SLs in each BG, our system should be able to handle 3 Byzantine failure in each BG if we do not execute the membership service. When we add the membership service to handle BG failures, an attacker can break the safety with 2 malicious nodes if they are both selected as BG leaders, and we have no methods to prevent malicious nodes from becoming BG leaders.

The second flaw is that BFT-GMS imposes an upper limit on the unfinished cycles in Round 3. RCanopus allows multiple cycles to execute their Round 3 in parallel for better utilization of WAN bandwidth. As a result, different BG do not finish a cycle at the same time, and a BG can finish cycles out of order. To ensure safety, we must set a limit on the max number of unfinished cycles in Round 3.

Consider the following scenario: $BG1$ finishes cycle 10 normally, and $BG2$ fails afterwards, so all other BGs go through the membership service to remove $BG2$ in cycle 10. If $BG1$ reveals the output to the clients immediately after it finishes cycle 10, its output would be different from all other BGs, violating the safety property. Suppose the maximum number of unfinished cycles in Round 3 is 5, then $BG1$ must wait until it finishes cycle 15 to reveal the results of cycle 10. Similarly, it must finish cycle 20 before it reveals the output of cycle 15. The idea is that all BGs must finish cycle 10 before they start cycle 15, If a BG has failed in cycle 10, all BGs will learn this fact no later than cycle 15 because they cannot fetch the results from the failed BG during Round 3.

This limit is a configuration parameter of the system. Increasing the limit will allow the system to utilize more available bandwidth. The throughput will increase, but it might also greatly increase the latency. Decreasing the limit will cause the BGs to more frequently wait for previous cycles to finish, thus reducing the throughput.

The third problem is that BFT-GMS fails to maintain the safety property in certain scenarios. In RCanopus, a BG can finish cycles out of order because of the asynchronous

design of Round 3. Consider the scenario where there are 7 BGs and BG_1 finishes cycle 11 and 12 before BG_2 fails, but no other BG is able to finish cycle 11 or 12 due to this failure. All BGs except BG_2 reaches a consensus to remove BG_2 from cycle 11. Later, BG_3 fails in cycle 12 and all the alive BGs remove BG_3 from cycle 12. It is possible for the leader of BG_1 to hide the membership change on cycle 11 from its SLs. Because BG_1 already has the result from both BG_2 and BG_3 in cycle 11, the SLs in BG_1 cannot detect the membership change. The reason for this problem is that BFT-GMS allows reusing membership information from previous cycles, but it lacks the mechanism to detect stale membership.

3.5.3 SIMPLE-GMS design

SIMPLE-GMS executes in every cycle, so there is no need to limit the number of unfinished cycles. It is executed by all BG leaders, but all actions in the membership protocol need to be backed by a quorum certificate of its own BG. Therefore, we can minimize the impact of malicious BG leaders. In order to tolerate f BG failures, there must be at least $3f + 1$ BGs in the system. The BGs achieve consensus on the membership in 3 steps.

Step 1

When a BG finishes Round 3 with the BG envoy extension, it should have either a positive response or a negative response for every other BG. These responses form a view of connectivity $CView_i$ between the local BG (BG_i) and all remote BGs. Table 3.1 is an example of connectivity view in a 4 BG deployment. It shows a connectivity view from BG_1. For each cell with a '+' or '-' sign, the connectivity view message includes a BFT-CompletionProof of the corresponding BG. We use f_i to denote the maximum number of Byzantine SLs tolerated in BG_i. In this case, BG_1 has successfully fetched the BFT results from BG_2 and BG_3, but at least $f_1 + 1$ envoys could not fetch from BG_4. As a result, the connectivity view of BG_1 consists of 3 positive responses, including itself, and 1 negative response. Figure 3.6 illustrates how this connectivity view is implemented as a message. The first field stores the ID of the BG to which this connectivity view belongs. In this diagram, the connectivity view is from BG_1. The rest of the message is a concatenation of several BFTCompletionProof messages, proving that their corresponding BFT results have been successfully replicated in BG_1. For simplicity, some fields are omitted in this diagram, such as the total number of BGs in the system.

Since all positive and negative responses from the Round 3 envoys have been replicated through the local SBFT protocol in BG_i, we can prove $CView_i$ with BFTCompletionProof

messages generated on BG_i. This guarantees that any machine in BG_i must provide the same $CView_i$ for the current cycle.

The envoys now communicate with remote BGs to fetch their connectivity views and submit them to the local BG leader. Similar to Round 3, the envoys may submit positive responses or negative responses for each remote BG. After a BG leader has received all positive and negative responses for the connectivity views, it replicates them within the BG using the SBFT protocol.

$CView_1$	BG_1	BG_2	BG_3	BG_4
BG_1	+	+	+	-

Table 3.1: An example of connectivity view.

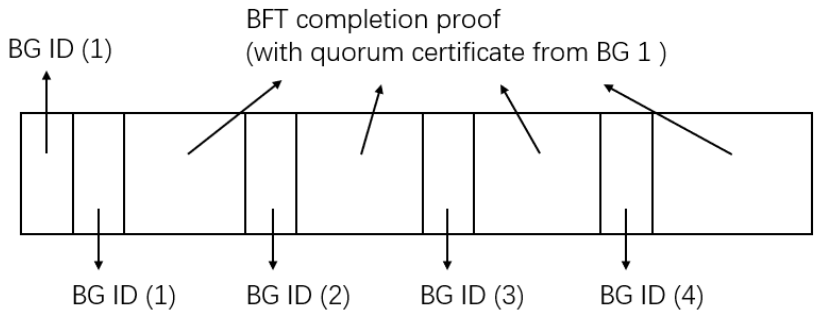


Figure 3.6: The message structure of a possible implementation of a connectivity view message.

Step 2

After BG_i has replicated the connectivity view messages from all BGs, it has a view of membership $MView_i$. As shown in Table 3.2, $MView_i$ is a combination of all connectivity views received by BG_i. Each row represents a connectivity view from the BG listed in the row header. The '+' and '-' signs in the cells represent positive and negative responses in Round 3. A row marked with '/' means the envoys failed to fetch the corresponding connectivity view message. In this example, BG_1 has received the connectivity views from BG_2 and BG_3.

The envoys now communicate with remote BGs to fetch their membership views and submit them to the local BG leader. Similar to Round 3, the envoys may submit positive

responses or negative responses for each remote BG. After a BG leader has received all positive and negative responses for membership views, it replicates them within the BG using the SBFT protocol.

$MView_1$	BG_1	BG_2	BG_3	BG_4
BG_1	+	+	+	-
BG_2	+	+	+	-
BG_3	+	+	+	-
BG_4	/	/	/	/

Table 3.2: An example of membership view.

Step 3

We now describe how each server deterministically calculates the membership of the current cycle with all the membership views it has received. Every server now calculates a union of all connectivity views inside the membership views, generating the final membership table. Since all connectivity views are protected by the quorum certificates inside BFTCompletionProof messages, there must not be any conflicts on '+' and '-' cells. Only rows previously marked with '/' may be filled with a new connectivity view. The cells on the diagonal are also marked as '+', even if they were previously marked with '/'. The final membership itself is not protected by any quorum certificates. This is OK because we do not disseminate it to other BGs.

Table 3.3 shows a final membership table calculated by a server S in BG_1. In this example, the connectivity view for BG_4 is still missing. We use N to denote the total number of BGs in the system. Upon generating this table, S counts the number of '+' and '-' signs in each column. A BG is considered a member of the system for the current cycle if its BFT result has been successfully replicated in at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs, including itself. If at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs report that they failed to replicate the BFT result of a specific BG, then that BG is excluded from the membership for the current cycle. If the membership status of any BG cannot be determined, then the server generating such a membership table must not complete this cycle.

It is worth pointing out that the goal of SIMPLE-GMS is to fix the flaws with the previous design, not to improve its performance. In fact, our design introduces higher latency because it has to be run on every cycle, whereas the previous design only runs on faulty cycles.

	BG_1	BG_2	BG_3	BG_4
BG_1	+	+	+	-
BG_2	+	+	+	-
BG_3	+	+	+	-
BG_4	/	/	/	+

Table 3.3: An example of the final membership.

One major improvement of SIMPLE-GMS over BFT-GMS is that SIMPLE-GMS uses envoys to perform inter-BG communications. The envoys may be malicious, but we select enough envoys for each cycle so that their collective result can be trusted. In contrast, the BFT-GMS design from the whitepaper uses only BG leaders, which makes the system more vulnerable when there are malicious BG leaders.

Chapter 4

Safety Analysis

In this chapter we analyze the safety property of RCanopus and our extensions.

4.1 Safety Proof of RCanopus

In the RCanopus whitepaper [14], the authors presented a safety argument by enumerating possible system failures and addressing them one by one. One potential problem with this approach is that we can never guarantee that we correctly enumerate all possible failures in our system. We have decided to write a proof for the same property. We are going to prove that our system provides a valid approach for a client to verify the output of our system. Any correct client would discard invalid results, and then we prove all valid results satisfy the following properties:

Definition 4.1.1. Safety properties of RCanopus

The results of RCanopus system must satisfy the following conditions.

(1) The results for the same cycle fetched from any server contain the same set of client transactions.

(2) There is a deterministic way to sort the committed transactions.

(3) All transactions in the valid results are submitted by the clients, i.e. they are not forged by a malicious server.

Now we introduce the safety assumptions of RCanopus. RCanopus can only tolerate a certain number of failures in the system. In this section, we exclude the membership

service component from RCanopus, because it is not in our implementation. Therefore, we are going to prove RCanopus meets the safety requirements when there is no BG failure in the system.

Definition 4.1.2. SL status

For any SL with N machines, if there are any Byzantine failures, i.e. malicious machines, then this SL is a **failed SL**. Alternatively, if a SL contains more than $\lfloor \frac{N-1}{2} \rfloor$ crashed machines, then this SL is a **failed SL**.

If a SL has not failed, then it is a **normal SL**.

Definition 4.1.3. BG status

For any BG with N SL, if there are more than $\lfloor \frac{N-1}{3} \rfloor$ **failed SLs**, then this BG is a **failed BG**. Otherwise, it is a **normal BG**.

Assumption 4.1.4. RCanopus safety condition

There are no failed BGs.

Lemma 1. *Any BG can at most commit once for a certain RCanopus cycle number.*

Proof. In SBFT, the leader is responsible for proposing the content of an SBFT cycle. SBFT cycles are identified by view-sequence pairs, e.g. (View 1, Sequence 1). In our implementation, an SBFT cycle contains the transactions for an RCanopus cycle. The SBFT leader is responsible for assigning an RCanopus cycle number when it creates the Preprepare message for an SBFT cycle. Between any two consecutive SBFT cycles, the RCanopus cycle number should be increased by 1.

We require that at most one SBFT cycle can be mapped to an RCanopus cycle. For example, we cannot have two committed cycles (View 1, Sequence 15) and (View 2, Sequence 1) that are both mapped to RCanopus cycle 15. The solution is to configure the SBFT participants to reject a Preprepare message if a previous SBFT cycle already used the same RCanopus cycle number. Algorithm 1 shows the procedure of an SBFT server accepting a Preprepare message from the SBFT leader. In Line 8, the server compares the RCanopus cycle number in the new Preprepare message to the cycle number extracted from the latest committed log entry. This ensures the RCanopus cycle number increases monotonically, so the same RCanopus cycle number cannot be assigned to two different SBFT cycles.

Suppose we have a BG with N SLs and the leader proposes a Preprepare message with an RCanopus number that has already been used. When a Preprepare message is proposed,

Algorithm 1: SBFT servers accepting a Preprepare message

input: A Preprepare message *ppm*
Result: True if *ppm* can be accepted
`/* Standard procedure to check the validity of transactions */`
for *transaction* $\in$ *ppm* **do**
  if *transaction* does not have a valid signature from a client **then**
    `// This transaction is invalid`
    return False
  end
end

`/* Check if the RCanopus cycle number has been used */`
`/* Extracts the RCanopus cycle number stored in` *ppm* `*/`
$cc \leftarrow ppm.rcanopus\_cycle$
`/* Retrieve the RCanopus cycle number from the last committed cycle in the local SBFT log */`
$cc\_old \leftarrow this.sbft\_log[this.sbft\_log.size() - 1].rcanopus\_cycle$
if $cc \neq cc\_old + 1$ **then**
  return False
end
return True

SBFT requires signatures from at least $N - f$ participants [12], where $f = \lfloor (N-1)/3 \rfloor$ is the maximum number of failed SLs. Since the RCanopus cycle number has already been used before, it will be rejected by any non-malicious SL. It can only obtain signatures from failed SLs. According to definition 4.1.3, if the BG has not failed, there are at most f SLs that will sign this Preprepare message. This means the Preprepare message cannot collect enough signatures, which prevents SBFT from committing a second SBFT cycle with the same RCanopus cycle number. □

Theorem 2. *When there is no failed BG in RCanopus, it has the properties defined in 4.1.1.*

Proof. First we look at property (1). From Lemma 1 we know that each BG generates at most one BFT result with a valid quorum certificate in each cycle. Since there is no BG failure in the system, the result of every cycle must contain all BGs. In Algorithm 3, a

Algorithm 2: BG result validity check

input: A BG result R; System configuration cfg
Result: True if R is valid
`/* Extract the BG ID and the threshold signature from the BFT result */`
1 $i \leftarrow R.BGid$
2 $sig \leftarrow R.signature$
`/* Extract the information about the BG from system configuration */`
3 $sbft\_pubkey\_list \leftarrow cfg$.listPubKeys($i$)
4 $n \leftarrow cfg$.BGsize(i)
5 $f \leftarrow cfg$.BGMaxFailure(i)
`/* Verify the threshold signature is signed by at least` $n - f$ `servers */`
6 **if** !VerifyThresholdSignature($hash(R), sig, sbft\_pubkey\_list, n - f$) **then**
7 | **return False**
8 **end**
9 **return True**

client only accepts BFT results with valid quorum certificates. This check is performed in Line 4. Therefore, all clients must see the same set of BFT results. The safety property of SBFT further guarantees that they contain the same transactions.

Property (2) can be achieved by sorting the BFT results by the BG ID. Alternatively, we can sort them by a nonce generated during Round 2.

Property (3) is guaranteed by SBFT. If a forged transaction is submitted to SBFT, the SBFT servers will refuse to commit this transaction, as it cannot pass the check on Line 2 in Algorithm 1. Since the BGs do not fail, there are enough non-malicious servers to reject forged transactions. □

4.1.1 Safety Proof of SkipCycle extension

Since the SkipCycle extension has been added to our RCanopus system implementation, it is very important to ensure it does not break RCanopus' safety property. In this section we prove that our system is still safe with the SkipCycle implementation.

Theorem 3. *When there is no failed BG, the RCanopus system with the SkipCycle extension has the properties defined in 4.1.1.*

Algorithm 3: System output validity check

input: A list of BG results *Rlist*;
A cycle number *cycle*
Result: True if *Rlist* is a valid output

```
/* The output of RCanopus consists of multiple BFT results committed
   in Round 2 on different BGs */
res_BG_list ← [ ]
for i ← 0 to Rlist.len()−1 do
    R ← Rlist[i];
    // Calls Algorithm 2
    if !validate( R, this.sysConfig ) then
        return False
    end
    if cycle ≠ R.rcanopus_cycle then
        return False
    end
    /* Extract the BG ID from the BFT result */
    id ← R.BGid
    if id ∈ res_BG_list then
        // This output contains two results from the same BG
        return False
    end
    res_BG_list.append(id)
end
BG_list ← this.sysConfig.listBGids()
if BG_list ≠ res_BG_list then
    // The results from some BGs are missing
    return False
end
```

Algorithm 4: BFTCompletionProof validity check

input: A BFTCompletionProof R; System configuration cfg
Result: True if R is valid
`/* Extract the BG ID and the threshold signature from the BFT result */`
1 $i \leftarrow R.BGid$
2 $sig \leftarrow R.signature$
`/* Extract the information about the BG from system configuration */`
3 $sbft\_pubkey\_list \leftarrow cfg$.listPubKeys($i$)
4 $n \leftarrow cfg$.BGsize(i)
5 $f \leftarrow cfg$.BGMaxFailure(i)
`/* Verify the threshold signature is signed by at least` $n - f$ `servers */`
6 **if** !VerifyThresholdSignature($R.digest, sig, sbft\_pubkey\_list, n - f$) **then**
7 | **return False**
8 **end**
9 **return True**

Proof. Introducing SkipCycle has no influence on the properties (2) and (3) listed in Definition 4.1.1. Therefore we mainly focus on the property (1).

Since there is no BG failure, a valid output of a cycle should contain the results from every BG. With SkipCycle, some of the results may be replaced by **BFTCompletionProof** messages. Different types of messages can be clearly distinguished. With the API provided in gRPC, it is trivial to separate **BFTCompletionProof** message from regular BG results. This can be seen in Line 7 , Algorithm 6. Thus we have a deterministic algorithm for the clients to examine whether an output is valid.

In Algorithm 6, after fetching the final result of an RCanopus cycle, the client handles **BFTCompletionProof** and normal BG results differently. In line 8, the client verifies that the skipped cycle number is included in the valid range of the **BFTCompletionProof**, but is not equal to *currentCycle*. This ensures that BG_i has skipped this cycle, so there are no transactions committed in BG_i. In line 12, we verify that the BG result has the correct number. This is how we check the result before SkipCycle is introduced. Note that the operation in line 4 is different for skipped cycles. Algorithm 4 differs from Algorithm 2 because it uses the digest stored in a **BFTCompletionProof**, instead of calculating it with a hash function.

We have proved for Lemma 1 that each BG can commit at most once in every cycle.

Algorithm 5: SBFT servers accepting a Preprepare message (with SkipCycle extension)

input: A Preprepare message *ppm*
Result: True if *ppm* can be accepted
`/* Standard procedure to check the validity of transactions */`
1 **for** *transaction* $\in$ *ppm* **do**
2 **if** *transaction* does not have a valid signature from a client **then**
 `// This transaction is invalid`
3 **return False**
4 **end**
5 **end**

`/* Check if the RCanopus cycle number has been used */`
`/* Extracts the RCanopus cycle number stored in` *ppm* `*/`
6 $cc \leftarrow ppm.rcanopus\_cycle$
`/* Retrieve the RCanopus cycle number from the last committed cycle in the local SBFT log */`
7 $cc\_old \leftarrow this.sbft\_log[this.sbft\_log.size() - 1].rcanopus\_cycle$
8 **if** $cc \leq cc\_old$ **then**
9 **return False**
10 **end**
11 **return True**

SkipCycle extension makes a small change in Algorithm 1. The only difference between Algorithm 5 and Algorithm 1 is Line 8. With SkipCycle, the cycle number may increase more than one. Since the cycle number is monotonically increasing, the proof for Lemma 1 still holds.

With Algorithm 6, we can guarantee that:

1. Skipped cycles contain no transactions.

2. Non-skipped cycles must have valid quorum certificates.

From Lemma 1, each BG generates at most one BFT result with a valid quorum certificate for each cycle. Therefore, these two properties can guarantee that both skipped and normal cycles have the exact same transactions in all outputs accepted by the clients.

Algorithm 6: System output validity check (with SkipCycle extension)

input: A list of BG results *Rlist*;
A cycle number *cycle*
Result: True if *Rlist* is a valid output

```
1  res_BG_list ← [ ]
2  for i ← 0 to Rlist.len()−1 do
3  |   R ← Rlist[i];
   |   // Calls Algorithm 2 or Algorithm 4 depending on the message type
   |      of R
4  |   if !validate( R, this.sysConfig ) then
5  |   |   return False
6  |   end
7  |   if R is a BFTCompletionProof message then
8  |   |   if R.lastCycle ≥ cycle OR R.currentCycle ≤ cycle then
9  |   |   |   return False
10 |   |   end
11 |   else
12 |   |   if R.currentCycle ≠ cycle then
13 |   |   |   return False
14 |   |   end
15 |   end
   |   /* Extract the BG ID from the BFT result */
16 |   id ← R.BGid
17 |   if id ∈ res_BG_list then
   |   |   // This output contains two results from the same BG
18 |   |   return False
19 |   end
20 |   res_BG_list.append(id)
21 end
22 BG_list ← this.sysConfig.listBGids()
23 if BG_list ≠ res_BG_list then
   |   // The results from some BGs are missing
24 |   return False
25 end
26 return True
```

□

4.1.2 Safety analysis of BG envoy extension

The main safety concern in the BG envoy design is that SLs can be malicious, and it is difficult to detect Byzantine SLs. If we select malicious SLs as envoys, we must not allow them to break the safety property or stall the system.

Since all remote BG results are protected by quorum certificates, it is not possible for a malicious envoy to forge results. However, a malicious envoy may forge a negative response indicating a timeout during the fetching process. We should also take notice that a malicious envoy might not respond.

Suppose there are N SLs in the BG and we can tolerate f malicious SLs. According to our design, $2f + 1$ SLs are selected as envoys in every cycle. At most f envoys can be malicious and submit no response to SBFT, so at least $f + 1$ envoys will submit either a positive response or a negative response. Therefore, we should be able to collect enough responses from the envoys when there are no more than f Byzantine SLs in the BG.

4.1.3 Safety analysis of SIMPLE-GMS

In SIMPLE-GMS, we calculate the membership of an RCanopus cycle by finding the BGs that have replicated their Round 2 BFT results in over a half of BGs in the system. In this section we will discuss the safety property of this approach. For simplicity, we use N to denote the total number of BGs.

Since all connectivity views are protected by **BFTCompletionProof** messages in SIMPLE-GMS, we have two observations:

1. If a server does not stall, its calculated membership will contain all BGs that have successfully replicated their BFT results in at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs.

2. If a server does not stall, its calculated membership will not contain any BGs that have not successfully replicated their BFT results in at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs.

To exclude a BG from the membership, there must be at least $(\lfloor \frac{N}{2} \rfloor + 1)$ '-' signs in its column in the final membership table (See Table 3.3). It is important to notice that '+' and '-' signs cannot replace each other. Either '+' or '-' may be replaced with '/'

due to missing connectivity view message. If the BFT result of BG_i has been successfully replicated in at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs, there will be $(\lfloor \frac{N}{2} \rfloor + 1)$ '-' signs, so BG_i must be in the membership. Similarly, if BG_i has not replicated its BFT result in at least $(\lfloor \frac{N}{2} \rfloor + 1)$ BGs, there will not be enough '+' signs to include BG_i in the membership.

From these observations, we can learn that the final membership will be the same for all servers that do not stall. This ensures that SIMPLE-GMS is safe.

However, under some rare scenarios, it is possible for SIMPLE-GMS to stall even if there are very few BG failures in the system. Consider the scenario where there are 101 BGs in the system, and BG_{101} fails in Round 3. Only BG_1 to BG_{50} have successfully replicated the BFT result of BG_{101}. During Step 2 in SIMPLE-GMS, BG_1 fails so no BG can receive its membership view. As a result, when the servers calculate the final membership, they will see 50 BGs successfully replicated the BFT result from BG_{101} (BG_2 to BG_{50} and BG_{101}) and 50 failed to do so. All servers will have to stall because they cannot decide whether to include BG_{101} in the membership or not.

A similar problem exists for BFT-GMS. Because it uses a global SBFT instance to determine the membership, it requires a super-majority of BGs to agree on any membership change. If a BG fails after disseminating its result to half of the remaining BGs, we do not have enough BGs to either vote out the failed BG or keep it in the system. So the system will stall.

If this exact scenario happens in SIMPLE-GMS, one possible solution is to let all BGs exchange their results and retry the membership protocol for this cycle. In this way, a partially disseminated result from a failed BG can be relayed to more BGs. However, this does not always solve the problem. In our example above, BG_1 permanently fails so the other BGs can never know its membership view. Moreover, BG_1 may have calculated the membership for this cycle and revealed it to the clients. Therefore, the other BGs must stall to ensure safety.

Chapter 5

Implementation

In order to verify the correctness of RCanopus [14] and compare its performance to existing solutions, we implemented RCanopus as an ordering service for HyperLedger Fabric [1]. The majority of our code is written in the Go programming language. The SBFT [12] implementation used by RCanopus is from a third-party repository, and it is written in C++. We use gRPC to handle the necessary communication between these two parts.

5.1 Main Components of the System

In Figure 5.1 we can see the main components of the system. Note that the arrows show possible interactions between components. Not all interactions will happen on every node in the system. For example, SBFT instances are only deployed on SL leaders. Therefore, Raft follower machines cannot participate in SBFT directly. They must receive Round 2 results from their leader.

Every server has a transaction input queue, storing the transactions coming from clients. For Raft follower nodes, these transactions are forwarded to the Raft leader. The transactions are removed from the queue once they are committed in a transaction block within the SL. The BFT manager on an SL leader is responsible for submitting transaction blocks to the BG leader, and receiving committed BFT results. The results from SBFT are also replicated within the SL with Raft. In this way, the Raft follower machines can learn the results of Round 2. The remote result collector is responsible for fetching results from remote BGs, and responding to such fetching requests.

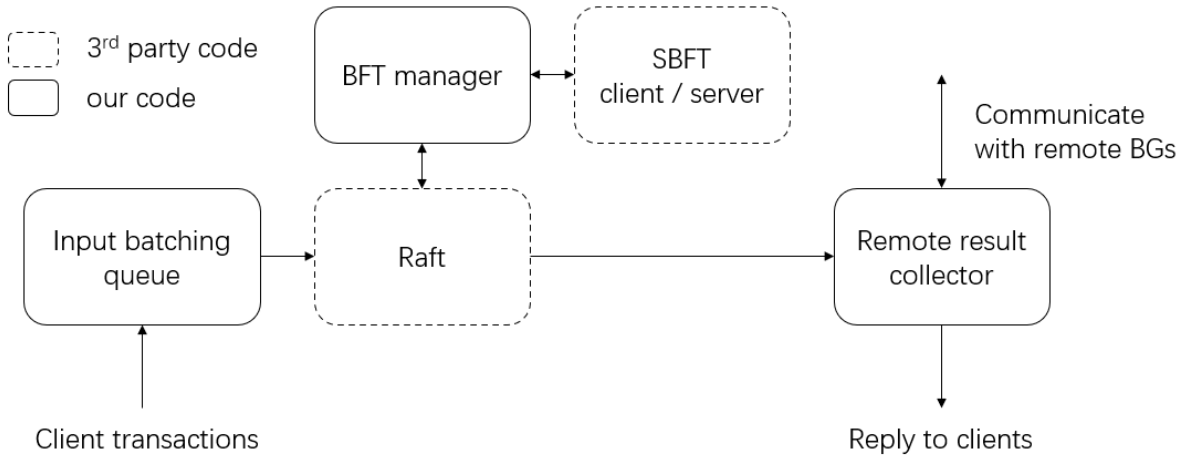


Figure 5.1: Main components in our RCanopus implementation

5.2 Consensus Building Block

RCanopus requires other consensus protocols to provide basic fault tolerance properties. It is also designed to ease the adoption of new consensus protocols. When new protocols with better performance are proposed, they can be easily plugged into RCanopus.

We chose Raft [27] to provide crash-stop fault tolerance for Round 1, and SBFT [12] to provide Byzantine Fault Tolerance for Round 2. The biggest advantage of Raft is that it is widely used, and there are Raft libraries in many programming languages. We chose to use etcd-raft[1] which is written in Go. SBFT is a BFT consensus protocol designed to achieve high scalability. In a deployment with N nodes, SBFT is capable of reducing the number of messages exchanged in a consensus cycle from $O(N^2)$ to $O(N)$. We selected SBFT because it had state-of-the-art performance when we started building RCanopus. We used the SBFT GitHub repository[2], which was published by the authors of the protocol.

[1]https://github.com/etcd-io/etcd/tree/master/raft
[2]Concord-bft, https://github.com/vmware/concord-bft/

5.3 Implementation limitations

The goal for our system implementation is to test the performance of the consensus protocol, RCanopus [14]. There are some details in the design that were not fully implemented, We also prioritized testing the performance in no-failure scenarios, so some fault tolerance features were not included in the implementation.

As mentioned in Section 3.1, the transactions in our system are not de-duplicated. This has little impact to the performance of the entire system, because we do not have malicious node replaying transactions. Moreover, we set the message resending timeouts to large enough values in our system, in order to prevent accidentally triggering this fault tolerance feature when there is no failure.

Another limitation is that SBFT does not check client signatures in the transactions. Since this security feature is missing, our clients also do not sign the transactions. This may cause our experiments to over-report the performance, because signature verification involves a large number of asymmetric encryption operations, which are known to be heavy computational workloads. However, we should note that signature verification consumes CPU resource, and we expect it to not be the bottleneck of our system. Our current experiments show that our system is bandwidth-bound. Even if the CPU resource becomes the bottleneck after we enable all the signatures, we can add more CPU resources relatively easily, comparing to the wide area bandwidth resources. In a cloud deployment, we can add CPU resources by using a machine type with more computing power, but there is no easy way to increase the bandwidth between datacenters. Also, we can always move signature verification after the consensus is reached, optimizing for the no failure case.

In the SBFT implementation we used, the leader can only store 1 outstanding transaction from each client in its buffer. In RCanopus, each SL leader serves as an SBFT client. SL leaders submit transaction blocks, which are formed by concatenating multiple client transactions. This limitation in SBFT means each SL cannot submit new transaction blocks before its previous block is committed. We have not found related description in the SBFT [12] paper and have thus guessed this is a pure implementation choice to prevent the leader from being flooded. This leads to the following consequences.

1. If an SL submits a batch before its previous batch is committed, the BG leader will simply drop this message, and the SL can only detect it after a timeout. This wastes bandwidth and also can greatly increases the latency. Nevertheless, even if the SL waits until the previous batch has been committed, its new batch may still be rejected by the leader, because the leader has not learned the previous batch has

been committed. This is very likely to happen, because in SBFT the final commit message is sent from an SBFT *collector* node, which can be any participant node in the protocol. This results in a race condition between the BG leader and other SLs.

2. When the SL submits a new batch after it gets the reply for the previous batch, this new batch will likely miss the next SBFT cycle. This is because the leader starts a new SBFT cycle immediately after the previous one ends, whereas the new batch only arrives after a roundtrip.

Since this defect in SBFT has serious consequences, we plan to fix it in our future work. Another option is to use a different BFT protocol, such as PBFT [3] or HotStuff [35][3].

5.4 Prototype for performance evaluation

We have implemented a prototype to demonstrate our BG envoy extension and the new membership service design. The prototype is written in C++ with around 460 KB code. Since our two extensions mainly focus on wide area communication, we decide to omit Round 1 in our prototype. Each instance of the prototype is deployed on a separate physical machine and emulates an entire SL. The network connection between two machines now resembles an inter-SL connection.

Our prototype contains a new implementation of the SBFT [12] protocol, which is independent from the Concord-BFT implementation. For simplicity, we only implement the fast path in SBFT, which is used when there is no failure in the BG. Our custom implementation of SBFT resolves a few performance-related issues of the Concord-BFT implementation. For instance, the leader no longer discards client requests, so we do not have to use extra bandwidth to resend the requests.

When we implement the BG envoy design, we optimize for the no failure case and select one primary envoy to fetch remote results. The other $2f$ envoys only start fetching if the primary envoy cannot return a positive response in time. A similar optimization exists in the original RCanopus [14] design, where the representative only contacts one SL in the remote BG in its first try.

In our prototype, a client is implemented as a separate thread in the same program as the server. This is because our prototype program represents the entire SL, so the communication between the client and the server should not consume network resources.

[3]A version is available on GitHub https://github.com/hot-stuff/libhotstuff

This is consistent with our previous experiments, where each client is assigned a local SuperLeaf. This also aligns with our plan for HyperLedger Fabric integration, as described in Section 2.3.3. In our HyperLedger integration, endorsed transactions are directed from the orderer nodes to an SL, and then the committed transaction blocks are directed to the committing peers. The orderers and committing peers should be located in the same datacenter with the SL in order to save network cost.

In order to have a fair comparison, our prototype has three modes: **Protonova**, **Protobase**, and **Protohyb**. Protonova uses the BG envoy extension in Round 3 and uses the SIMPLE-GMS to determine BG membership. Protobase uses the original Round 3 design in RCanopus [14] to exchange BFT results between BGs. For the membership service, Protobase executes a SBFT protocol deployed on all BG leaders every cycle. This is to show that our new membership service has a reasonable performance, because BFT-GMS in the RCanopus paper uses an SBFT protocol among all BG leaders to agree on membership changes, but it only execute when a membership change is necessary. As mentioned in Section 3.5, we believe that the BFT-GMS design does not contain enough details to prevent replaying a stale membership, so a reasonable fix is to run the system-wide consensus protocol in every cycle. In order to separate the influence of BG envoy and SIMPLE-GMS, we implemented Protohyb, which uses the same Round 3 design as Protobase, but uses the SIMPLE-GMS for membership service.

When there are fewer than 4 BGs, the membership service in Protobase cannot tolerate any BG failure, but it will still execute all steps in the SBFT protocol. Similarly, SIMPLE-GMS will also execute all the necessary steps although it cannot tolerate a BG failure.

The code is available at `https://github.com/duanqn/RCanopusMembership`.

Chapter 6

Evaluation

In this section we present our evaluation results. Our evaluation has two parts. The first part (Section 6.1) evaluates the performance of RCanopus, which is implemented as an ordering service for HyperLedger Fabric. This implementation includes all stages of the RCanopus protocol except the membership service. It also includes the extensions described in Section 3.2 and 3.3. The second part (Section 6.2) evaluates our prototypes introduced in Section 5.4: **Protonova**, **Protobase** and **Protonhyb**. The purpose of this evaluation is to explore the performance of the BG envoy extension and SIMPLE-GMS.

It is worth noting that the evaluation results in Section 6.1 and Section 6.2 are not directly comparable, because the experiments are conducted on different implementations, and they are deployed in different environments. In Section 6.1, we deployed the system on AWS, whereas in Section 6.2 we deployed the system on our lab cluster.

6.1 Evaluation of the RCanopus ordering service

In this section, we present our evaluation on our implementation of RCanopus. The evaluation is performed on AWS EC2. We use c5.9xlarge instances for both clients and servers, which have 36 virtual cores and 72 GiB memory. We also use mounted SSD volumes for storage to reduce the performance impact of disk I/O. Unless otherwise specified, transactions in this section are 240 bytes long.

Machines in AWS EC2 are organized into different *regions* by their geographical location. Each region has several datacenters that are located within close proximity. These datacenters are also called *availability zones*. In our deployment, machines within the same

	us-east-1	us-west-2	ap-southeast-1	ap-southeast-2	eu-west-1	eu-west-2
us-east-1	4.8 Gbps					
us-west-2	312 Mbps	4.8 Gbps				
ap-southeast-1	94.1 Mbps	133 Mbps	4.8 Gbps			
ap-southeast-2	106 Mbps	161 Mbps	254 Mbps	4.8 Gbps		
eu-west-1	325 Mbps	175 Mbps	119 Mbps	40 Mbps	4.8 Gbps	
eu-west-2	310 Mbps	159 Mbps	128 Mbps	37.4 Mbps	2.3 Gbps	4.8 Gbps

Table 6.1: Available bandwidth between AWS regions, or between different availability zones in the same region.

	us-east-1	us-west-2	ap-southeast-1	ap-southeast-2	eu-west-1	eu-west-2
us-east-1	0.55					
us-west-2	75.50	0.16				
ap-southeast-1	192.30	162.60	0.10			
ap-southeast-2	197.74	138.20	91.00	0.55		
eu-west-1	74.30	129.90	177.80	259.45	0.59	
eu-west-2	74.90	137.60	170.60	273.67	9.50	0.12

Table 6.2: Roundtrip latency (ms) between AWS regions, or between different availability zones in the same region.

SL are deployed in the same availability zone. SLs that are in different BGs are deployed in different AWS regions. For SLs that are in the same BG, we deploy them in the same region if possible. For regions that do not have enough availability zones to hold all SLs within a BG, we deploy a BG across two nearby AWS regions. Each SL contains three c5.9xlarge instances,

Table 6.1 and 6.2 show the bandwidth and latency between AWS regions used in our experiments. We deploy two BGs in us-east-1 (North Virginia) and us-west-2 (Oregon). Since individual AWS regions in Europe and Asia do not have enough availability zones, we are spreading the machines in the same BG across two regions. We deploy one BG across eu-west-1 (Ireland) and eu-west-2 (London), and another BG across ap-southeast-1 (Singapore) and ap-southeast-2 (Sydney). From the table we can see that the bandwidth between different availability zones that are located in the same region is approximately 5 Gbps. Across different regions, the bandwidth is much lower, ranging from 40 Mbps to 330 Mbps.

In practice, a BFT system like RCanopus should be deployed across different cloud service providers to ensure they do not share common security vulnerabilities. However, for simplicity and cost saving reasons, we are only using AWS for our experiments.

6.1.1 Scalability Experiments

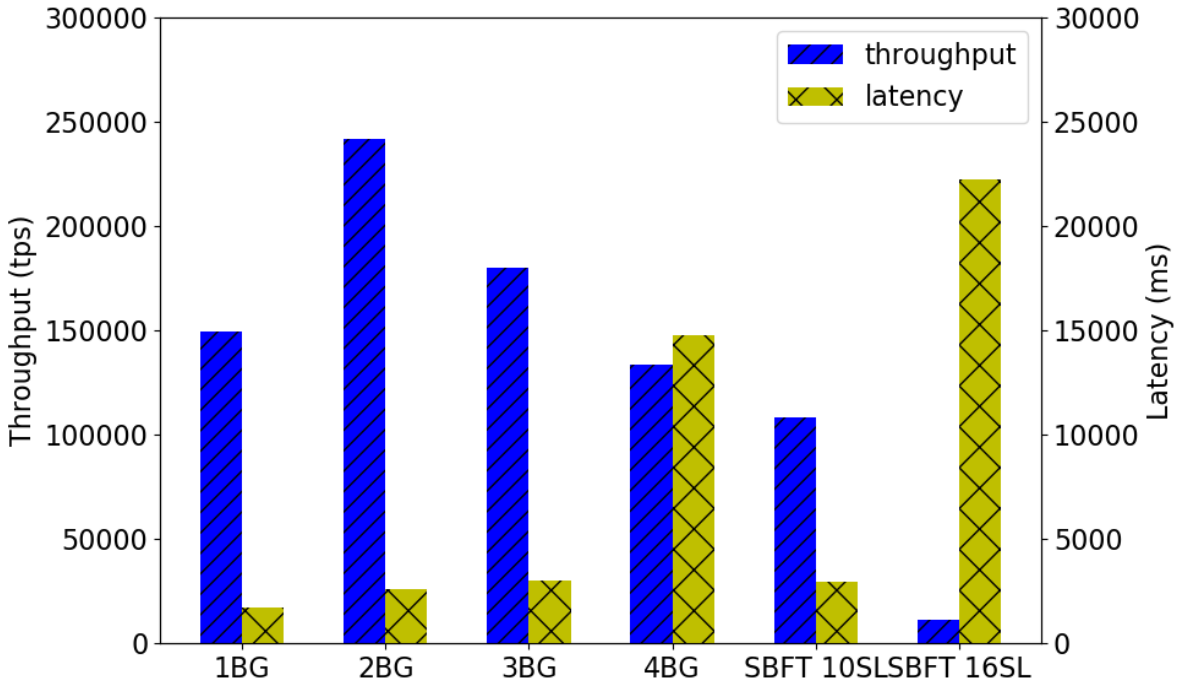


Figure 6.1: Peak throughput and median latency for various configurations

Our first series of experiments is to evaluate the scalability of RCanopus when we increase the number of BGs in the system. In these experiments, each BG has 4 SLs, and each SL has 3 machines. Therefore, the total number of machines varies from 12 to 48. We used a pipeline depth of 16, which means there can be at most 16 cycles executing their Round 3 in parallel. The transaction size is set to 240 bytes, which is the same size used in the SBFT paper. In Round 1, at most 8000 transactions can be included in one transaction block.

Figure 6.1 shows the throughput and median latency for different configurations. The experiments are 90 seconds long. We remove the first and last 15 seconds in each log file, in order to avoid the performance disturbance from the warm-up period or after the clients have stopped sending new transactions. On each machine, we remove the 15 seconds after the first cycle finishes, and 15 seconds before the last cycle finishes. The median latency is calculated over the end-to-end latencies of all transactions committed in the remaining

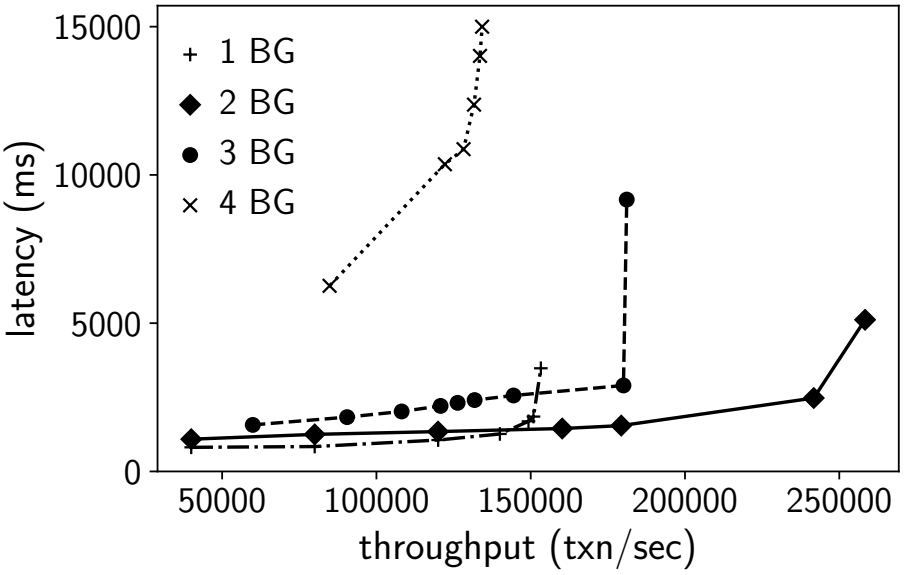


Figure 6.2: Throughput-latency curves for various configurations

period. All throughput and median latency values are average values of multiple runs of the experiment.

RCanopus with an 1 BG configuration has an average peak throughput of 149,278 tps with a median latency of 1,712 ms. This setting does not have Round 3 as there is no remote BG to contact. Hence the latency is lower than multi-BG setups. The median latency increases to around 3 seconds for the 2 BG and 3 BG experiments. When we add the 4th BG, the latency greatly increases to 14,789 ms. In Figure 6.2 we can also see that the latency greatly increases when we add the 4th BG. This is because the 4th BG we added is located in the Asia Pacific region, which has high latency and low bandwidth to all other three regions, as shown in Table 6.1 and 6.2. We have also observed packet drops in the communication across AWS regions. Since the 4th BG has a higher latency to other BGs, packet drops also have a larger impact on the end-to-end latency.

From Figure 6.1 and Figure 6.2 we can see that the peak throughput increases to 241,730 tps when we add the second BG, but drops to 180,056 tps with 3 BGs, and further drops to 133,561 tps with 4 BGs. We believe there are two factors affecting the peak throughput. When we add more BGs, there are more BFT instances processing transactions, so the total processing speed of Round 2 will increase. At the same time, extra BGs increase the

communication cost of Round 3, because the messages need to be disseminated to more recipients. With the specific settings in this series of experiments, our system has the highest peak throughput when there are 2 BGs.

In order to compare with SBFT, we deployed our system with 1 BG and 16 SLs, using the same machines as our 4 BG deployment. This 1 BG deployment does not have Round 3, so the performance primarily depends on the SBFT running in Round 2. The peak throughput is only 11,400 tps, and the median latency is over 22 seconds. This is because SBFT does not utilize network resources efficiently. The machines are constantly waiting for responses from remote machines to proceed in the protocol. Since SBFT cycles do not overlap, the high latency also lead to the low throughput.

We also deployed our system with 1 BG and 10 SLs in the same AWS region. This is to compare against our 1 BG experiment, where we had 12 machines forming 3 SLs in the same region. Since the Concord-BFT implementation only supports node numbers in the form of $3f + 1$, we set up 10 participants in SBFT to have a fair comparison. This means we have 30 machines in total. We believe this reflects the performance of SBFT with 10 nodes, because Round 1 is not the dominant factor in performance. Comparing this to our RCanopus 1 BG experiment, RCanopus has 38% higher throughput and 40% lower latency.

6.1.2 Transaction Size Experiments

Different applications may use transactions of varying sizes. In the previous section, we evaluated our system with transactions of 240 bytes because this is the size used in the SBFT [12] paper. A transaction in HyperLedger Fabric can reach 3 KB, including the message header. Therefore, it is important to evaluate the performance of our system under different transaction sizes.

In this series of experiments, we deploy 2 BGs across 8 availability zones in 2 AWS regions (us-east-1 and us-west-2). The entire deployment contains 24 machines. We deploy 1 client in each availability zone. For each transaction size, we vary the sending rate on the clients and record the peak throughput, as well as the corresponding median latency.

As shown in Figure 6.3, the throughput drops with increasing transaction sizes. With 240 byte transactions, our system is able to achieve nearly 250,000 tps throughput. When the transaction size increases to 2000 bytes, the throughput drops to around 30,000 tps. Throughout the experiments, the product of throughput and transaction size remains around 59 MB per second. This shows that the throughput of our system, when measured in bytes per second, is largely unaffected by the transaction size.

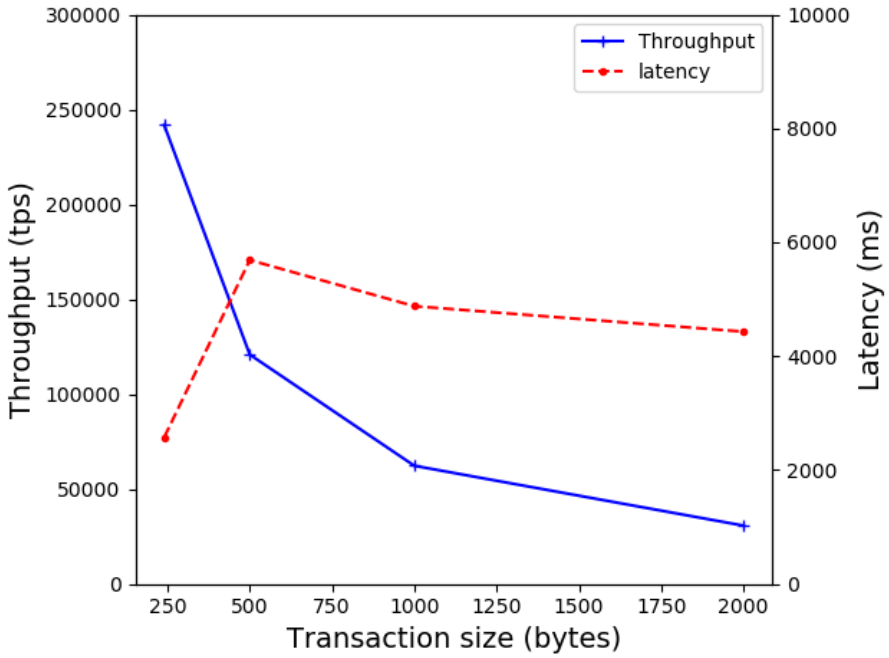


Figure 6.3: Peak throughput and median latency with different transaction sizes

We expect the end-to-end latency to remain constant in this series of experiments, but our data shows that the 240-byte experiment has a much lower median latency (around 2.5 seconds) than other experiments. We suspect this is because the 240 bytes experiments were run at a different time than the others, and the network conditions are different during the two time period. The 240 bytes experiments were run together with other scalability experiments. We have observed packet drops on wide area links between BGs. Since the wide area links are external to AWS, the network condition may vary from time to time. A higher packet drop rate can lead to more packet retransmission, which can significantly increase the end-to-end latency.

6.2 Evaluation of BG envoy extension and SIMPLE-GMS

In this section, we present the evaluation results on our prototype. This prototype is implemented to demonstrate the effect of the BG envoy extension and the new global

membership service. As described in Section 5.4, it has 3 modes (Protonova, Protobase, Protohyb) in order to compare the performance difference of BG envoy and SIMPLE-GMS seperately.

In these experiments, we test the performance of our system with increasing sending rates on the client side. In each run, the clients send transactions for 300 seconds, and we record the throughput and average latency. In our calculation, we exclude the first 15 seconds after the first cycle is committed, and the last 10 seconds before the last cycle is committed. The throughput is calculated over this period. The average latency is calculated as an average of end-to-end latencies of all transactions committed during this period. For each configuration, we run the experiment multiple times and show a 95% confidence interval in our graphs.

We ran these experiments on our local cluster. Each machine has two Intel Xeon E5-2620 processors, which have 12 cores in total, with hyperthreading disabled. Each machine has 64 GB memory and an Intel S3700 SSD, and is running Ubuntu 16.04 . The machines are connected to switches with 10 Gbps links.

In order to emulate a real world network condition, we use *tc* to limit the bandwidth on inter-BG links. Figure 6.4 shows how we limit the bandwidth between the machines. Note that a node represents an entire SL in the original RCanopus protocol, because our prototype omits Round 1. On the left side, the two machines represent two SLs in the same BG, so the link bandwidth between them is not limited. The node on the right side represents an SL from a different BG. On every machine, we limit the aggregate outbound bandwidth to all SLs in remote BGs.

We understand that the bandwidth constraints shown in Figure 6.4 cannot precisely emulate the network environments in the real world. With *tc*, we cannot control the inbound bandwidth on each BG. Also, in our configuration, the traffic from one SL to other SLs located in different geographical regions compete with each other, which might not be true in the real world.

6.2.1 Scalability Experiments

In this series of experiments, we set up our system with 2 to 4 BGs to evaluate the scalability of Protonova, Protobase, and Protohyb. In the first set of experiments, we set inter-BG roundtrip latency to 100 ms, intra-BG roundtrip latency to 10 ms, and inter-BG bandwidth to 1 Gbps. Each transaction is 256 bytes long in these experiments.

Figure 6.5 shows the performance of Protonova using BG Envoy extension and the new global membership service. The peak throughput for 2, 3, and 4 BGs are 74,925 tps, 90,033

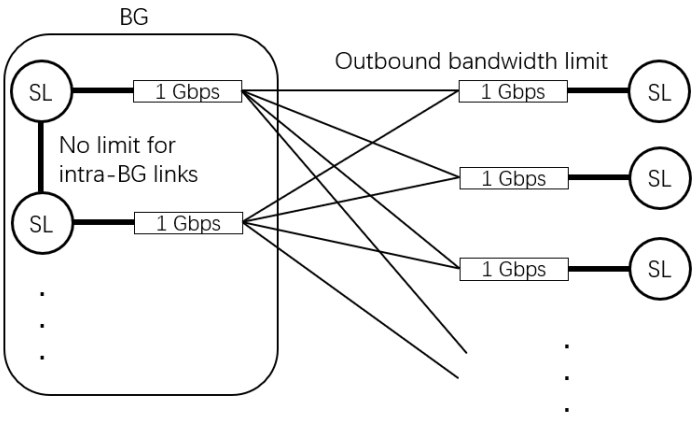
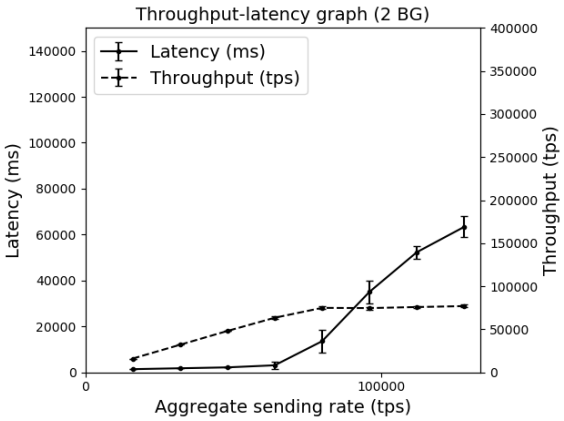


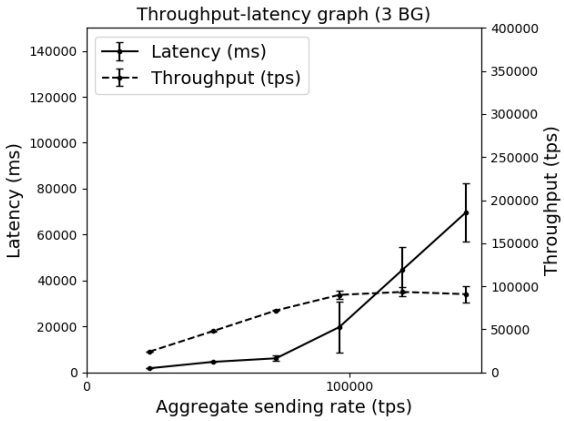
Figure 6.4: A diagram showing how the link bandwidth is controlled. Unannotated links have 10 Gbps bandwidth.

tps, and 93,553 tps, respectively. When we increase the number of BGs in the system, we have more SBFT instances running in parallel. This explains the increasing throughput. More BGs will also lead to more inter-BG traffic, which can cause the throughput to drop. However, this influence is not as significant because BG Envoy extension can greatly reduce the inter-BG bandwidth consumption.

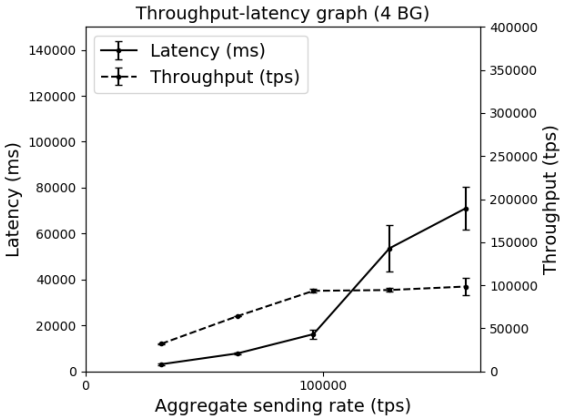
Figure 6.7 shows the performance of Protobase, which does not have the BG Envoy extension or the SIMPLE-GMS. The throughput drops rapidly from 330,115 tps in the 2 BG configuration, to 83,003 tps in the 4 BG configuration. This is because the BFT result of a BG needs to be broadcast to all other BGs 4 times, as there are 4 SLs in each BG. Therefore, the increasing traffic in Round 3 quickly consumes the available inter-BG bandwidth resources.



(a) 2 BG, 1000 Mbps, 100 ms



(b) 3 BG, 1000 Mbps, 100 ms



(c) 4 BG, 1000 Mbps, 100 ms

Figure 6.5: Scalability experiments for **Protonova**. Intra-BG roundtrip latency is set to 10 ms. The number of BGs, the bandwidth and roundtrip for inter-BG links are listed in the caption of each subfigure.

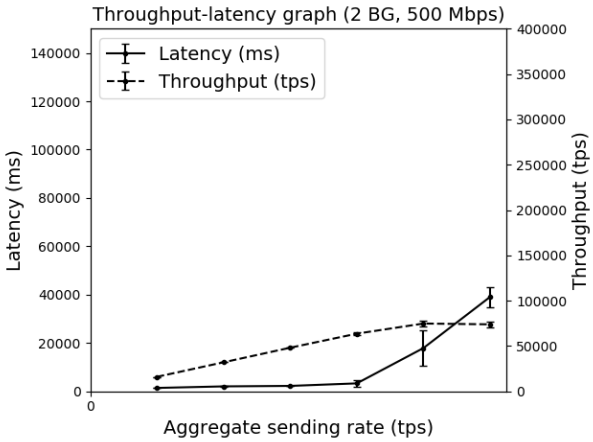
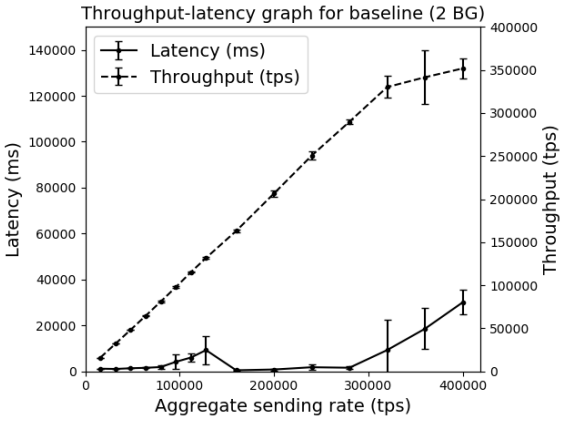
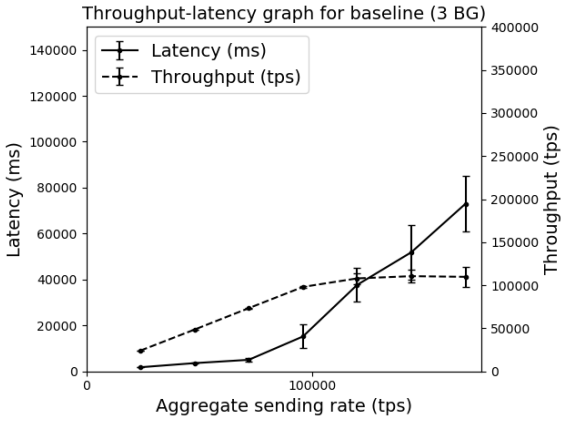


Figure 6.6: Additional experiments on 2BG configuration for Protonova. Inter-BG latency is 100 ms. Inter-BG bandwidth limit is 500 Mbps.

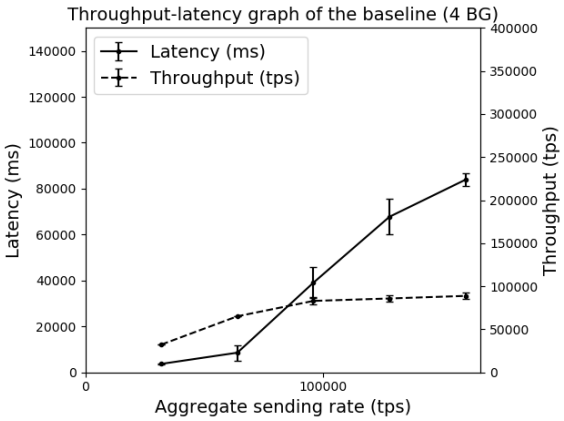
The peak throughput of the 2 BG configuration of Protobase (330,115 tps) is much higher than Protonova (74,925 tps). In order to find out the reason, we ran the 2 BG experiments of Protonova with 500 Mbps instead of 1 Gbps. The result is shown in Figure 6.6. We can see that the peak throughput is about the same as in Figure 6.5a. This means the inter-BG communication is not the decisive factor in the 2 BG setup. The low performance of Protonova is because SIMPLE-GMS introduces many extra local BFT rounds, which prevents Protonova from utilizing the bandwidth efficiently.



(a) 2 BG, 1000 Mbps, 100 ms

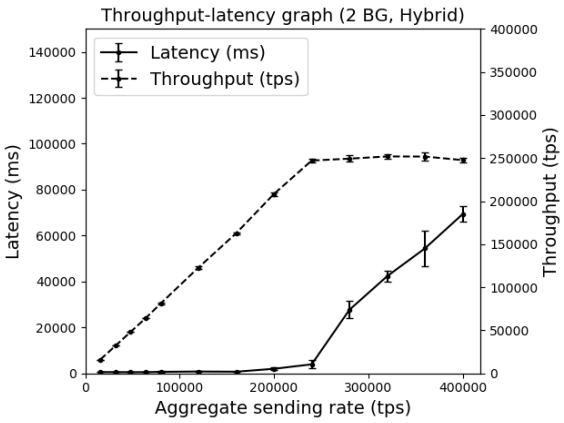


(b) 3 BG, 1000 Mbps, 100 ms

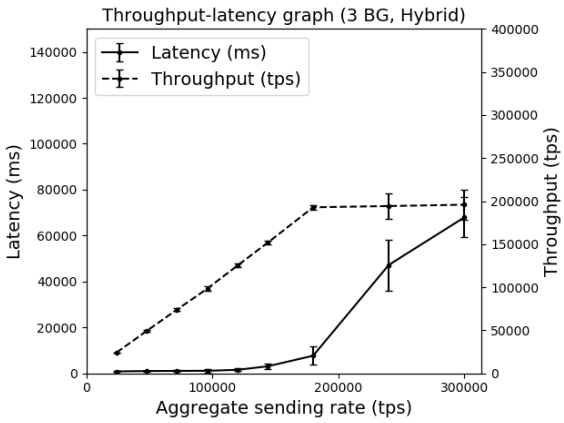


(c) 4 BG, 1000 Mbps, 100 ms

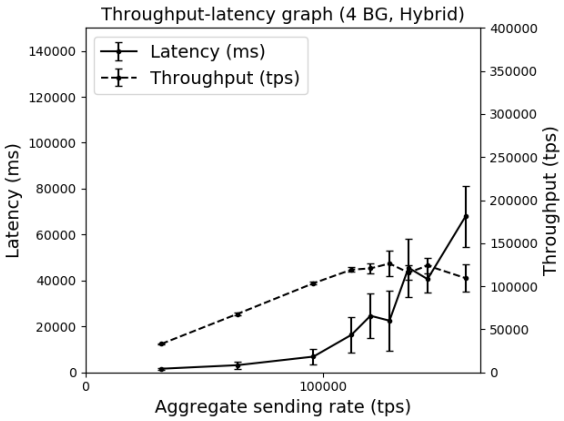
Figure 6.7: Scalability experiments for **Protobase**. Intra-BG roundtrip latency is set to 10 ms. The number of BGs, the bandwidth and roundtrip for inter-BG links are listed in the caption of each subfigure.



(a) 2 BG, 1000 Mbps, 100 ms



(b) 3 BG, 1000 Mbps, 100 ms



(c) 4 BG, 1000 Mbps, 100 ms

Figure 6.8: Scalability experiments for **Protohyb**. Intra-BG roundtrip latency is set to 10 ms. The number of BGs, the bandwidth and roundtrip for inter-BG links are listed in the caption of each subfigure.

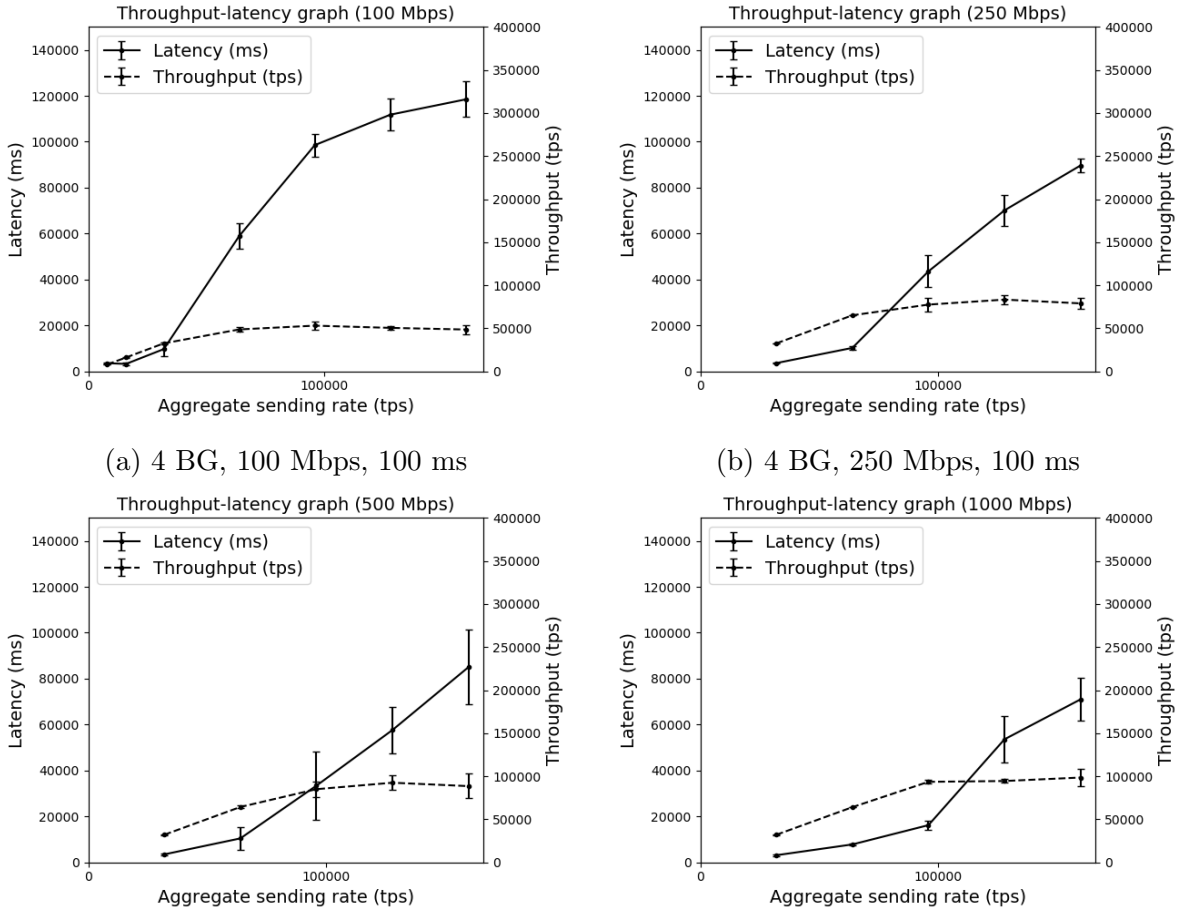
In order to compare the effect of BG Envoy and the membership service separately, we have implemented a hybrid mode in our prototype. This hybrid mode uses BG Envoy extension, but uses the same membership service protocol as the baseline. Figure 6.8 shows the performance of the hybrid mode. The peak throughput for 2, 3, and 4 BGs are 247,112 tps, 192,873 tps and 120,765 tps. Comparing Figure 6.8 to Figure 6.5 and Figure 6.7, we can see that the BG Envoy has a positive effect on system throughput, while the new global membership service has a negative impact. The latter is mainly because the membership

service introduces additional rounds of BFT consensus in a single cycle. Because local BFT cycles cannot overlap, this becomes a bottleneck of the system.

In summary, these are our findings from these experiments.

- Protonova have better scalability than the baseline. When we have 4 BGs and set the inter-BG bandwidth limit to 1 Gbps, Protonova has 12.7% higher peak throughput than Protobase.
- Protonova performs much better than Protobase in low WAN bandwidth settings. With 100 Mbps inter-BG bandwidth, a 4-BG Protonova deployment has 114.3% higher peak throughput than a 4-BG Protobase deployment.
- The BG envoy extension helps increase the throughput of our system. The SIMPLE-GMS has worse performance when compared to the baseline, mainly because it introduces many more local BFT rounds. This is within our expectation because the top priority for SIMPLE-GMS is to address the design problems on safety. For example, in order to use a consensus protocol that only runs on BG leaders to determine membership, BFT-GMS sets a limit on the maximum number of malicious BG leaders. This is impractical because we cannot prevent a malicious machine from becoming an SBFT leader.

6.2.2 Inter-BG bandwidth sensitivity



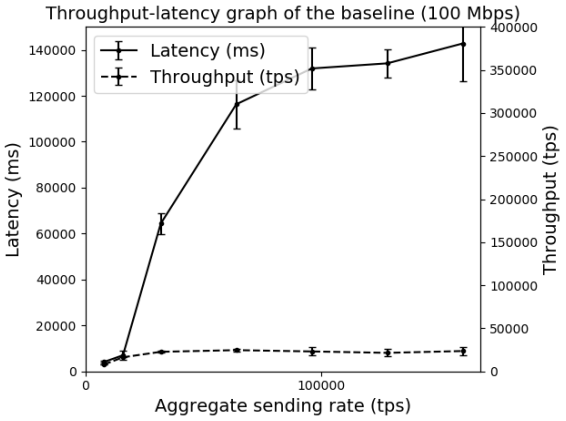
(a) 4 BG, 100 Mbps, 100 ms

(b) 4 BG, 250 Mbps, 100 ms

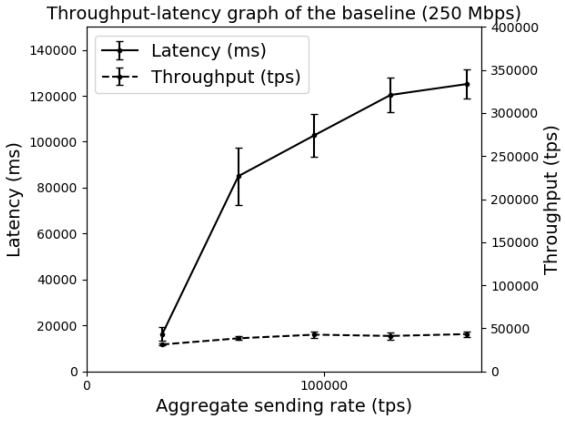
(c) 4 BG, 500 Mbps, 100 ms

(d) 4 BG, 1000 Mbps, 100 ms

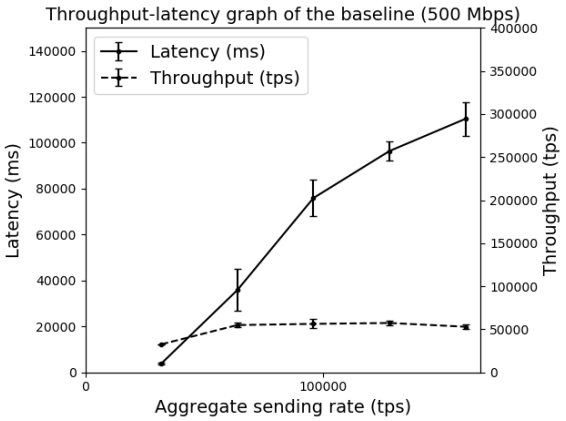
Figure 6.9: Bandwidth experiments for **Protonova**. Intra-BG roundtrip latency is set to 10 ms. The number of BGs, the bandwidth and roundtrip for inter-BG links are listed in the caption of each subfigure.



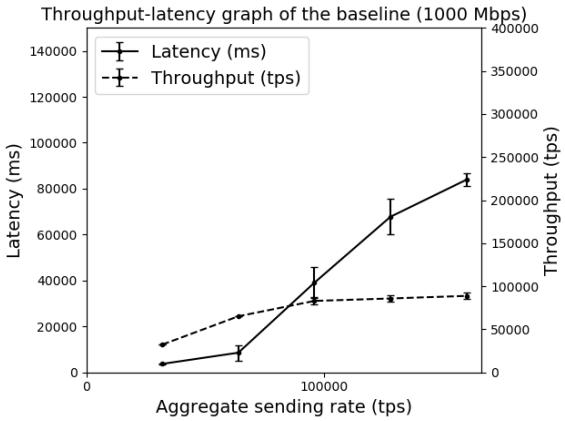
(a) 4 BG, 100 Mbps, 100 ms



(b) 4 BG, 250 Mbps, 100 ms



(c) 4 BG, 500 Mbps, 100 ms



(d) 4 BG, 1000 Mbps, 100 ms

Figure 6.10: Bandwidth experiments for **Protobase**. Intra-BG roundtrip latency is set to 10 ms. The number of BGs, the bandwidth and roundtrip for inter-BG links are listed in the caption of each subfigure.

In this series of experiments, we use a 4 BG deployment to compare the performance under different inter-BG bandwidth settings. We set intra-BG roundtrip latency to 10 ms, inter-BG roundtrip latency to 100 ms. Each transaction is 256 bytes long. Since the bandwidth usage of the global membership service is very low when compared to Round 3, we directly compare the performance of Protonova and Protobase, in order to show the effect of the BG envoy extension.

Figure 6.9 shows the peak throughput and average latency when using BG Envoy ex-

tension and the new global membership service design. We can see that the peak throughput decreases when we reduce the inter-BG bandwidth, indicating that our system is bandwidth-bound under this configuration. The throughput drops from 93,553 tps to 65,143 tps (30% decrease) when we reduce the bandwidth from 1 Gbps to 250 Mbps (75% decrease). At 100 Mbps, the peak throughput drops to 48,733 tps. This is a 48% decrease in throughput comparing to the 1 Gbps data.

Figure 6.10 shows the peak throughput and average latency for the baseline. When we set the inter-BG bandwidth to 1 Gbps, the peak throughput is 83,003 tps. As we reduce the bandwidth, the peak throughput gradually drops to 54,975 tps, 38,442 tps and then finally reaches 22,744 tps at 100 Mbps. By comparing the 100 Mbps configuration and the 1 Gbps configuration, we can observe a 73% drop in peak throughput.

Our experiment results show that our new design outperforms the baseline in low inter-BG bandwidth settings, and the throughput is less sensitive to bandwidth reduction. This is because our BG Envoy design leverages the tradeoff between intra-BG bandwidth and inter-BG bandwidth to reduce wide area traffic. Suppose every BG commits a 4 MB BFT result in a specific cycle. In the baseline, every BG needs to fetch 12 MB $\times$ 4 = 48 MB on WAN links, because the 4 SLs fetch independently. With the BG Envoy design, every BG only needs to fetch 12 MB, but we need to replicate them within the BG locally, which results in extra 36 MB of intra-BG traffic.

Chapter 7

Conclusion

7.1 Contributions

In order to test the performance of RCanopus, we have created an implementation that complies with the interface for ordering services in HyperLedger Fabric. We have also implemented a baseline ordering service which executes an SBFT instances over all SL leaders. Our evaluation result shows that with a 16-SL deployment across 4 AWS regions, the baseline can only reach 11,400 tps with over 22 seconds end-to-end latency. On the other hand, if we divide the 16 SLs into 4 BGs by their regions, RCanopus can achieve 133,561 tps peak throughput with an end-to-end latency of 14.7 seconds. This is a 10.7× improvement in peak throughput. This result shows that by dividing BFT participants into smaller Byzantine Groups in a topology-aware way, RCanopus is able to improve the aggregate throughput of the system. This is because the Byzantine Groups can execute the BFT protocol in parallel, achieving a better utilization of the network resources.

In Chapter 3 we proposed two extensions to reduce the WAN bandwidth usage of RCanopus. In Section 3.2, we redesigned the communication pattern when one SL fetches the Round 2 result from a BG. In Section 3.4, we reduced the number of SLs in a BG that need to fetch the results from remote BGs. Previously, all SLs inside a BG need to fetch independently. With the BG envoy extension, only 1 SL need to fetch when there is no failure in the system. The effect of the BG envoy extension can be seen from our evaluation results in 6.2.1.

In Chapter 4 we provided a safety proof for the original RCanopus protocol, and for the SkipCycle extension. The original design does not tolerate BG failures. In order to

enhance the safety of RCanopus, in Section 3.5 we designed a protocol called SIMPLE-GMS to handle BG crash failures. We also analyzed the safety of the BG envoy extension and SIMPLE-GMS in Chapter 4. In our evaluation, we find that this protocol lowers our system throughput when compared to using a global SBFT instance across all BGs, mainly because it introduces many more local BFT rounds inside a BG. The impact becomes smaller when we increase the number of BGs, as the inter-BG bandwidth becomes the decisive factor of the performance.

7.2 Limitations and Future Work

Like most research work, our project can be further extended to address additional limitations. We now list them out, in hopes that other people can pick them up and continue our journey.

Although our evaluation has shown the BG envoy design is capable of improving the throughput of RCanopus, it is not included in our implementation. Due to the time constraint, we only built a prototype to demonstrate its potential. Our next step is to integrate it into RCanopus. Theoretically, it should be compatible with all existing designs.

The SBFT implementation used in RCanopus does not fully meet our requirements. After reading the code, we find that it is implemented for small transaction size and low communication latency, preferably with all participants in the same datacenter, although these are not required by the protocol itself. In the future, we can either improve the implementation, or switch to a different protocol such as HotStuff [35].

When we integrate our RCanopus ordering service into HyperLedger Fabric, the peak throughput is only around 7,000 tps. This is mainly because we used much larger transactions for HyperLedger Fabric, which are around 3 KB each. This is still very far from the performance of HyperLedger Fabric when using Kafka as an ordering service. The performance of the integrated system is not very stable, and may require further investigation. The evaluation about this integrated system is not shown in the thesis because this part was led by another student.

In this thesis we have provided detailed discussion on the safety of the protocol and its extensions. However, we did not prove the liveness property to guarantee the system can make progress, which is also very important for consensus protocols. Although the FLP impossibility [8] tells us it is impossible to guarantee safety and liveness at the same time, it is still useful to prove liveness under some special conditions. For example, when there is no BG failure in the system, we expect the system to have liveness property. In the

future, we will do a liveness analysis of RCanopus, which may lead to minor changes in the protocol.

References

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[2] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[3] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[4] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[5] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[6] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140, 2019.

[7] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: a scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, pages 45–59, 2016.

[8] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[9] The Linux Foundation. Blockchain showcase. `https://www.hyperledger.org/resources/blockchain-showcase`. Accessed: 2020-05-10.

[10] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[11] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463. IEEE, 2019.

[12] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

[13] Manos Kapritsos and Flavio Paiva Junqueira. Scalable agreement: Toward ordering as a service. In *HotDep*, 2010.

[14] Srinivasan Keshav, W Golab, Bernard Wong, Sajjad Rizvi, and Sergey Gorbunov. Rcanopus: Making canopus resilient to failures and byzantine faults. *arXiv preprint arXiv:1810.09300*, 2018.

[15] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[16] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[17] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[18] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[19] Barbara Liskov. From viewstamped replication to byzantine fault tolerance. In *Replication*, pages 121–149. Springer, 2010.

[20] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[21] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 369–384. USENIX Association, 2008.

[22] 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.

[23] Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

[24] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[25] Karl J O'Dwyer and David Malone. Bitcoin mining and its energy footprint. 2014.

[26] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[27] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 305–320, 2014.

[28] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[29] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 426–438, 2017.

[30] Joao Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[31] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[32] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, pages 323–336. USENIX, 2015.

[33] Jason Warner. October 21 post-incident analysis. `https://github.blog/2018-10-30-oct21-post-incident-analysis/`. Accessed: 2020-06-22.

[34] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[35] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.