# ShallowForest: Optimizing All-to-All Data Transmission in WANs

by

Hao Tan

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, 2019

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**

All-to-all data transmission is a typical data transmission pattern in both consensus protocols and blockchain systems. Developing an optimization scheme that provides high throughput and low latency data transmission can significantly benefit the performance of those systems. This thesis investigates the problem of optimizing all-to-all data transmission in a wide area network (WAN) using overlay multicast. I first prove that in a congestion-free core network model, using shallow tree overlays with height up to two is sufficient for all-to-all data transmission to achieve the optimal throughput allowed by the available network resources. Based on this finding, I build ShallowForest, a data plane optimization for consensus protocols and blockchain systems. The goal of ShallowForest is to improve consensus protocols' resilience to skewed client load distribution. Experiments with skewed client load across replicas in the Amazon cloud demonstrate that ShallowForest can improve the commit throughput of the EPaxos consensus protocol by up to 100% with up to 60% reduction in commit latency

**Acknowledgements**

I would like to thank my supervisors, Professor Wojciech Golab and Professor Srinivasan Keshav, for their invaluable advice and guidance. I sincerely appreciate the opportunity to work with them during my master's program. Without their kindness, patience and knowledge, I would not have completed this thesis. In addition, I would like to thank Professor Bernard Wong for his constructive advice and comments throughout all stages of this research. Last but not least, I want to thank Professor Samer Al-Kiswany for his valuable comments on the thesis.

Also, I must express my very profound gratitude to my parents and to my fiancee for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

## Dedication

This is dedicated to the one I love.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Being highly available in the presence of machine failures and network partitions is crucial to today's network services. State machine replication (SMR) [25] is a well-established technique to build fault-tolerant distributed systems. By having a group of replicated state machines collectively play the role of a server, the service can continue to operate when some of the machines fail. In SMR, each state machine executes an unbounded sequence of commands that update the current state. To make server state consistent across replicas, all replicated state machines must execute the same sequence of commands. To solve this challenging problem, replicated state machines communicate according to a specific consensus protocol to agree upon on a single sequence of commands to execute. Due to the asynchrony of the system, where messages can be delayed arbitrarily and processes can become arbitrarily slow, the replicas of a replicated state machine cannot always be in exactly the same state.

Traditionally, consensus protocols have been crucial building blocks in modern distributed systems for replicating important data and providing a strict ordering of updates to a small number of machines [9, 23]. Recently, blockchain [33, 10, 5, 40] has become an emerging category of systems that require large scale consensus involving hundreds of nodes across different geographical regions connected by a wide-area network (WAN). Both consensus protocols and blockchain systems require multicasting data to a group of receivers. The following communication pattern dominates the normal operation of consensus protocols: upon receiving client requests, a replica broadcasts a message with commands to all other replicas and commits the request after receiving a certain number of responses. Such a communication pattern can be abstracted as an all-to-all data transmission, where each node in the cluster broadcasts an infinite stream of data to all other participating nodes.

## 1.1 A Simple Example

In a network consisting of $n$ geo-distributed sites $v_1, \ldots, v_n$, using a single overlay for data dissemination can be suboptimal in terms of both throughput and network latency. Consider the case where each site has equal uplink and downlink capacity $B$ Mbps and the network latency between each pair of sites is $L$ ms. Assume $v_1$ needs to broadcast $B$ Mbps to every receiver. By having $v_1$ send the data directly to each receiver, each receiver can only receive the data at the rate of $\frac{B}{n-1}$. Another alternative is to use a path joining all sites starting at $v_1$, which broadcasts data at the rate of $B$ Mbps. However, the communication latency incurred by the last receiver on the path equals to $(n-1)L$ ms.

Above examples are sub-optimal in terms of either transmission throughput or latency. However, when using multiple overlays for data dissemination, we can reduce the cost of latency for achieving high data transmission throughput. For this approach, the original stream can be evenly partitioned into $n-1$ streams which are first sent to $v_2, \ldots, v_n$ respectively. Upon receiving the data, each site then broadcasts the data to the remaining $n-2$ sites. By using this approach, the network latency incurred by the data transmission becomes $2L$ ms while the transmission throughput remains $B$ Mbps.

The above examples consider a simple case which motivates the research presented by this thesis. It raises a fundamental problem: given a set of nodes connected by a WAN and each node having a stream of data to broadcast to all other nodes, how can we maximize the aggregated broadcast throughput while minimizing the latency for each node's data to reach all other nodes?

## 1.2 Motivation

Leader-centric consensus protocols like Paxos [26] and Raft [34] have a stable leader to handle all client requests. Since internet protocol (IP) multicast is not generally available in a WAN environment, the stable leader in those protocols sends the data directly to all other replicas using multiple unicast transmissions. Assuming each site in the network is associated with an uplink capacity that limits the aggregated throughput of outgoing flows to other sites, this approach would render the leader as the bottleneck. As the number of replicas grows, each transmission will have less share of the available uplink capacity at the leader. Some protocols [30, 29] addressed this issue by handling data transmission using one or more ring overlays to maximize bandwidth utilization. However, a ring overlay is not an ideal option in a WAN environment due to the high latency of WAN links.

Other protocols [28, 31, 7] alleviate the single leader bottleneck by distributing the load of data dissemination across all nodes. This strategy works best when the load is spread uniformly across all replicas. However, workloads in the real world can be highly skewed across different geo-areas and continuously changing over time. When each replica sends data directly to other replicas, it may yield sub-optimal throughput and lead to a load imbalance across replicas. Therefore, I argue that data dissemination should be handled in a more flexible way for consensus protocols to achieve high commit throughput and low commit latency.

## 1.3 Contributions

In light of these challenges, I propose *ShallowForest*, an algorithm that optimizes data transmission for consensus protocols using overlay broadcast. ShallowForest computes data transmission overlays according to client load and available network capacity at each replica to make consensus protocols achieve high throughput and low latency. The central idea behind ShallowForest is inspired by overlay multicast protocols such as SplitStream and Bullet[11, 24], which partition the data stream at the sender and broadcast each stream partition with a potentially different tree overlay. However, the primary difference between ShallowForest and prior overlay multicast protocols is that ShallowForest only uses shallow tree overlays to reduce the network latency subject to the data transmission. This preference over shallow tree overlays is also backed by the sufficiency of shallow tree overlays in achieving the optimal throughput of all-to-all data transmission, which is described in details in Chapter 4.

In this thesis, I make the following contributions:

1. I prove that the optimal throughput of all-to-all data transmission is achievable by using tree overlays with height up to 2 in a congestion-free core network model.

2. I formulate the data transmission overlay optimization problem as a linear programming (LP) problem and build ShallowForest.

3. I apply ShallowForest to Egalitarian Paxos (EPaxos) and conduct experiments in the Amazon Elastic Compute Cloud (EC2) [1]. The experiment results demonstrate that ShallowForest can improve the commit throughput of EPaxos by 100% while reducing the commit latency by 60% with skewed client load across replicas.

## 1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 goes through the background of consensus problems and related consensus protocols. Chapter 3 presents the benchmark results of data transmission throughput in the Amazon EC2 cloud. Chapter 4 presents the proof for the sufficiency of shallow tree overlays in achieving the optimal throughput. Chapter 5 presents the details of the ShallowForest optimization. Chapter 6 describes Amoeba Paxos (APaxos), which uses ShallowForest to optimize data transmission for EPaxos. Chapter 7 evaluates APaxos through experiments conducted in the Amazon EC2 cloud. Chapter 8 surveys the related work and Chapter 9 concludes the thesis and discusses future work.

# Chapter 2

# Background

This chapter provides an overview of the consensus problem and consensus protocols related to this thesis.

## 2.1 Consensus Problems

The consensus problem is a fundamental problem in distributed computing that requires multiple processes to agree on a common output. Real world agreement problems include but are not limited to leader election, state machine replication and committing distributed transactions. The consensus problem is defined as follows by [6]:

> In a system with $n$ processes $p_1, \ldots, p_n$, each process $p_i$ has an input value $x_i$ and an output $y_i$, which is also known as the decision. Initially, $x_i$ is drawn from a known set of values and $y_i$ is undefined. A solution to the consensus problem must satisfy the following properties:
>
> 1. Termination: In every admissible execution, $y_i$ is eventually assigned a value, for every non-faulty process $p_i$.
> 2. Agreement: If $y_i$ and $y_j$ are assigned for non-faulty processes $p_i$, $p_j$, then $y_i = y_j$.
> 3. Validity: In every execution, if, for some value $v$, $x_i = v$ for all non-faulty processes $p_i$, and if $y_i$ is assigned for some non-faulty processes $p_i$, then $y_i = v$.

Consensus protocols coordinate multiple processes to solve the consensus problem. Solving the consensus problem is challenging when processes can be faulty and communication channels are unreliable. A correct process acts according to the protocol until the protocol terminates, while a faulty process suffers a failure prior to the termination of the protocol. The failure can be one of the following types according to [20]:

1. **Fail-stop failure:** A process crashes and stops receiving or sending messages indefinitely.
2. **Omission failure:** A process fails to send or receive messages when it should.
3. **Byzantine failure:** A process sends erroneous messages that compromise the protocol.

Based on the assumption on message delivery and processing speed, the system can be synchronous, partially-synchronous or asynchronous. The upper bound on message transmission delay is known in a synchronous system, while a message may take an arbitrarily long time to deliver in an asynchronous system. The partially-synchronous system lies in between synchronous and asynchronous; it assumes the upper bound exists but is not known apriori. The consensus problem has been intensively studied by previous works for various assumptions on failure model and system types. [20] has proven that for the consensus problem in partially synchronous systems, there has to be at least $2f + 1$ processes to tolerate $f$ non-Byzantine failures and $3f + 1$ processes to tolerate $f$ Byzantine failures. According to the FLP impossibility result [21], consensus problem is not solvable within finite steps in an asynchronous system with just fail-stop failures. [8] proves that probabilistic protocols can guarantee either termination of agreement deterministically at the expense of the other in an asynchronous environment.

## 2.2 Paxos

Paxos [26], proposed by Leslie Lamport, is one of the most influential and widely used protocols. In general, Paxos solves the consensus problem in an asynchronous system with non-Byzantine failures. It can tolerate up to $f$ failures with $2f + 1$ processes. Basic Paxos, also known as single-decree Paxos, coordinates a collection of processes to choose a single value such as a command proposed by a client. Although being able to choose a single value can hardly be useful in practice, basic Paxos is the building block of Multi Paxos, which can be used to implement SMR in a production environment.

In basic Paxos, there are three types of processes involved in the protocol: *proposer*, *acceptor* and *learner*. The proposer sends a proposed value $v$ to a set of acceptors. An acceptor may accept the $v$ if some conditions are met. The value $v$ is said to be *chosen* when a majority quorum of acceptors accept $v$ or assign $v$ to their decisions. Proposer and acceptor play central roles in deciding the chosen value and learners are often omitted in the discussion as they simply learn the chosen value.

Basic Paxos is a two-phase algorithm:

1. Prepare Phase:

   (a) Upon receiving some value $v$ from the client, the proposer broadcasts a prepare message $Prepare(n)$ to all acceptors, where $n$ is the proposal number the proposer selects. The proposal number selected must uniquely identify a proposal. One possible form of proposal number is $< r, id >$ where $r$ is the round number and $id$ represents the ID of the proposer process. Each process can store the largest round number $r_{max}$ (initialized as 0) it has ever seen and use $r_{max} + 1$ as the round number of its new proposal number.

   (b) Upon receiving $Prepare(n)$, the acceptor $p_i$ will send a response based on following variables it maintains:

      i. $l_i$: the highest proposal number $p_i$ has received but not necessarily accepted.

      ii. $a_i$: the highest proposal number $p_i$ has accepted.

      iii. $v_i$: the value $p_i$ has accepted.

      If $n > l_i$ and $p_i$ has accepted some proposal, the acceptor $p_i$ will update $l_i = n$ and reply $PrepareAns(a_i, v_i)$. If the $p_i$ has not accepted any proposal yet, it will reply $PrepareAns(n, \varnothing)$.

   The proposer can achieve two goals during the prepare phase:

   (a) Block other proposals with lower proposal numbers.

   (b) Learn the value accepted by acceptors.

2. Accept Phase:

   (a) Having received $PrepareAns$ messages from the majority of acceptors, the proposer will broadcast $Accept(n, v)$. If $v_i \neq \varnothing$ in any received $PrepareAns$ messages, $v$ is replaced with the $v_i$ associated with the highest $a_i$ among all the received $PrepareAns$ messages.

(b) Upon receiving $Accept(n, v)$, if $n \geq l_i$, acceptor $p_i$ will set $a_i = n$, $l_i = n$ and $v_i = v$. In the end, the acceptor reply to the proposer with $l_i$

The value $v$ is considered chosen when the majority of acceptors replied $(n, v)$ during phase $2.b$. The proposer goes back to phase $1.a$ with a higher proposal number when it fails to receive enough responses in either phase. Built on the basis of the basic Paxos, Multi Paxos divides the consensus into a sequence of instances. By running basic Paxos for each instance, processes can reach consensus on a sequence of values. To avoid executing the prepare phase for every single instance, Multi Paxos sets one process to be a stable leader. The prepare phase is only needed at the start of the protocol or when the current stable leader fails. In the discussion of consensus protocols, Paxos often refers to Multi Paxos.

## 2.3 Egalitarian Paxos

Most Paxos-based consensus protocols are leader-based. For instance, in Multi-Paxos, the designated leader will handle all client requests and broadcasting protocol messages to all other replicas. Such a design has two major problems in a WAN environment: 1) The leader becomes a single point of failure that impairs the availability of the service; 2) High commit latency: client requests originating from the site other than the leader site incur an extra round of wide-area transfer; 3) The computing and network resources of the leader process will limit the system throughput.

EPaxos is a variant of Paxos designed to solve the single leader bottleneck issue. Unlike other Paxos variants, there is no designated leader in EPaxos and clients can send their commands to any replica. The leaderless nature of EPaxos allows the system to evenly distribute the load across all replicas. EPaxos also significantly reduces the communication overhead of geo-distributed state machine replication by allowing clients to send requests to their closest replicas. When the majority of replicas are correct, EPaxos commits a command with a single round of network communication and it takes at most two rounds of network communication to commit a command that interferes with other concurrently proposed commands. In the context of a key-value store, two commands interfere with each other if one of them is a write command and both commands operate on the same key.

The central idea behind EPaxos is to establish consensus on ordering constraints for each proposed command, which are used by every replica to compute the same order of commands locally. The ordering constraints of a command $\gamma$, consist of two components:

dependency list ***dep*** and sequence number ***seq***. *dep* is a list of all instances that contain commands that interfere with $\gamma$, where each instance is a batch of commands. *seq* is the smallest number that is greater than the sequence number of all other interfering commands in *dep*.

Like Paxos, EPaxos also involves two phases: Phase 1 and Phase 2. The goal of Phase 1 is to establish the ordering constraints for a command $\gamma$. Upon receiving a command $\gamma$ from a client, a replica becomes the command leader and sends a *PreAccept* message to all other replicas that constitute a fast path quorum. A *PreAccept* message contains $\gamma$, *deps* and *seq* where *deps* and *seq* are computed based on the command leader's local state. When the *PreAccept* message reaches a replica, the replica updates *deps* and *seq* according to its local state and replies to the command leader with the updated *deps* and *seq*. If the command leader receives replies from a fast path quorum of replicas with unchanged *deps* and *seq*, it sends a *Commit* message to all other replicas to commit $\gamma$ and replies to the client. Committing a command after Phase 1 is also referred to as committing a command on the fast-path.

Since another replica's state may contain commands interfering with $\gamma$ that are not present in the state of the command leader, the command leader might fail to receive responses with unchanged *deps* and *seq* from a fast-path quorum of replicas. In such a case, it computes a new *deps* by taking the union of all received dependency lists and updates *seq* to be larger than the largest sequence number in the received response. After that, the command leader proceeds to Phase 2. During Phase 2, the command leader sends an *Accept* message including updated *deps* and *seq* to at least a majority quorum. Upon receiving the *Accept* message, the replica updates its state and sends a reply to the command leader. When the command leader receives replies from a majority of replicas, it sends a *Commit* message to all replicas and replies to the client. In some scenarios such as the command leader fails, a command $\gamma$ cannot be committed by its original command leader. In such case, other replicas received $\gamma$ through *PreAccept* message will compete to become the command leader of $\gamma$ and commit $\gamma$ through a Paxos-like protocol.

Using the protocol described above, each non-faulty replica is able to build a dependency graph for not-yet executed commands in the log using *deps* associated with each command. In the case that the dependency graph contains a cycle, the replica breaks the cycle using *seq* associated with each involved command. During the execution phase, each replica executes the command in the topological order of the dependency graph.

# Chapter 3

# Measuring Inter-DC WAN Throughput

Many consensus protocols and blockchain systems are deployed on a public cloud across multiple geographical areas. It is worth-while knowing the performance of inter-datacenter (Inter-DC) networking on a public cloud before proposing any optimization schemes for the data transmission in such an environment. Some previous works [36, 41, 22] provided a quantitative measurement of the performance of Inter-DC networking on a public cloud. Below is a summary of the conclusions and key observations presented in those works:

1. Public cloud providers often set per-VM rate limits to ensure performance isolation across tenants such that the maximum aggregated throughput of a single VM is capped [41, 22].

2. There is also a per-flow rate limit for inter-DC flows regardless of the sending and receiving window size of TCP configurations. Increasing the number of flows will result in higher aggregated throughput until the upper limit is reached [41].

3. There is no per-flow rate limit for intra-DC flows. However, the maximum throughput of an intra-DC flow is still capped by the per-VM rate limit [41].

4. The maximum aggregated throughput varies spatially between different pairs of sites. However, the temporal difference in aggregated throughput between the same pair of sites is not significant [22].

To verify the observations made by prior works and get a better sense of how an inter-DC WAN performs, I conduct measurement on the Amazon EC2. Besides measuring the

throughput of one-to-one data transfers between each pair of VMs as previous works did, I also benchmark the throughput of one-to-many transfers and many-to-one transfers, which are two dominant communication patterns in applications such as consensus protocols and cryptocurrencies. Based on the benchmark results, I present the network model used in this thesis.

## 3.1 Throughput of one-to-one data transfer

I start my measurement by benchmarking the throughput of one-to-one data transfer between each pair of VMs. In my measurement, I set up VMs with high network performance (m4.xlarge instance) in four different geo-areas: Ireland, Tokyo, US-East (Virginia), and US-West (Oregon). To measure the throughput, I use iperf3 [2] to initiate 30 flows, keep them running for 90 seconds, and record the average throughput reported by iperf3. Table 3.1 shows a summary of the results. As opposed to up to 3X variation in throughput measured using a single TCP flow [36], the benchmark results measured using multiple flows are similar across data centers. The variation among different sender-receiver pairs is approximately 10%. All aggregated throughputs are capped around 750 Mbps.

Table 3.1: Average aggregated throughput of inter-DC links.

| | Virginia | Oregon | Tokyo | Ireland |
|---|---|---|---|---|
| Virginia | - | 753 Mbps | 702 Mbps | 749 Mbps |
| Oregon | 761 Mbps | - | 760 Mbps | 705 Mbps |
| Tokyo | 715 Mbps | 738 Mbps | - | 694 Mbps |
| Ireland | 763 Mbps | 700 Mbps | 687 Mbps | - |

## 3.2 Throughput of one-to-many data transfer

One of the questions I want to investigate is: does the per-VM rate limit apply to one-to-many data transfer. To answer this question, I set the VM in one datacenter to be the sender and let it send data to all other VMs concurrently. I use Iperf3 to start 30 TCP flows from the sender VM to each receiver VM and keep them running for 60 seconds. After 60 seconds, I record the average data transmission throughput of each sender-receiver

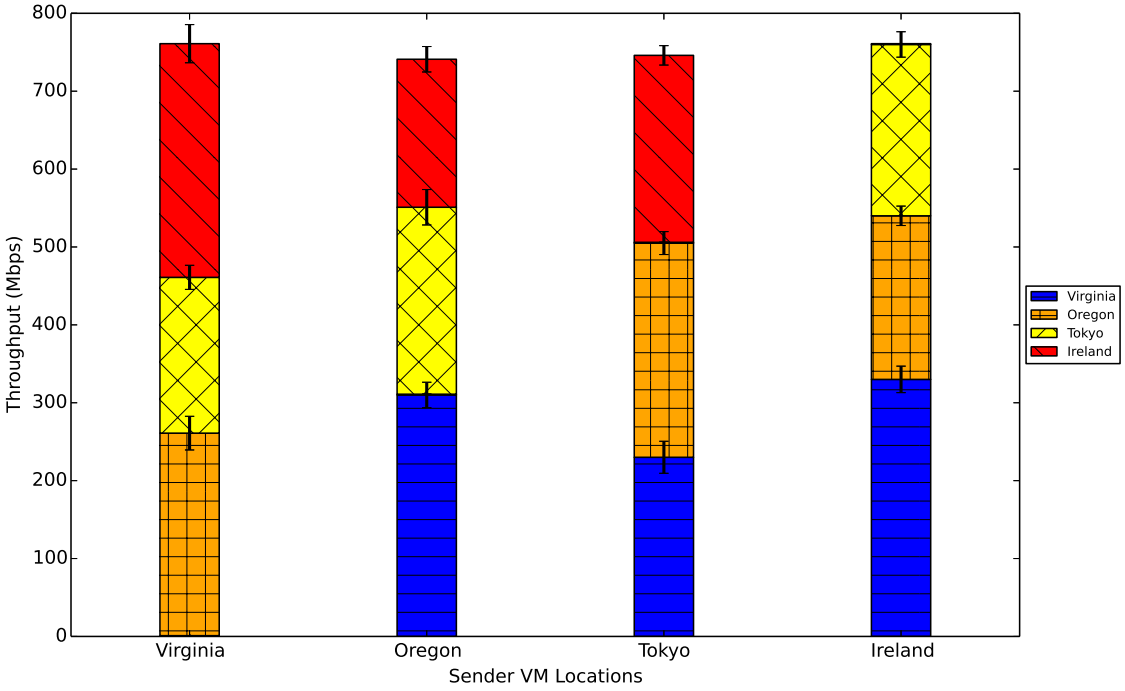


Figure 3.1: Aggregated throughput of one-to-many data transfers.

pair. The results are presented in Figure 3.1, where the x-axis represents the location of the sender VM and each bar represents the aggregated throughput of the one-to-many data transmission at a specific location. Each segment with a particular colour scheme represents the throughput of the data transmission between the sender and a particular receiver. From the graph, we can see that, when there are multiple receivers, the aggregated throughput of data transmission to all receivers is capped around 750 Mbps. This rate is roughly equivalent to the per VM rate limit. Due to the congestion control mechanism of TCP, the size of the sending window of a closer receiver grows faster than that of a distant receiver. If a sender sends data aggressively to each receiver using TCP connections, the closest receiver usually receives the greatest portion of the sender's uplink capacity. I also observe similar trends in many-to-one data transfers where a single receiver receives from multiple senders at the same time.

## 3.3 Remarks

As a complement to the conclusion and observation made by prior works, I list conclusions below based on my benchmark result:

- According to Table 3.1, using multiple inter-DC flows for one-to-one transfer results in not only higher aggregated throughput, but also less spatial variation. Although the spatial variance is insignificant in my measurement, [22] reported 2X variation between EU-US throughput and US-ASIA throughput when using VMs with higher per-VM rate limit.
- According to Figure 3.1, multiple one-to-one transfers originating at the same sender VM will contend for the available uplink capacity. The aggregated throughput is still constrained by the per-VM rate limit. The same arguments apply to the downlink capacity for multiple one-to-one transfers destined at the same receiver VM.

## 3.4 Network Model

Instead of assuming a specific network topology for the WAN, this thesis uses the hose network model [19] which models the network as a set of sites connected by a core network with unlimited capacity as demonstrated by Figure 3.2. All sites can send and receive data from each other, bottlenecked only by the edge link capacity between each site and

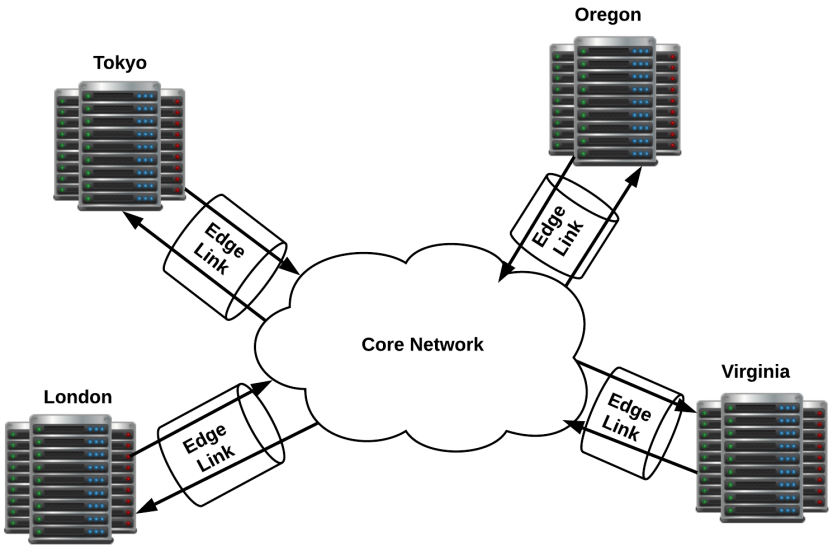


Figure 3.2: An illustration of the hose network model.

the core network. Such a model is often used to represent a WAN in works related to network flow optimizations [37, 13]. It is not only simple but also valid as shown by recent measurements [17] and my own measurement presented in Section 3.1 and Section 3.2.

In the remainder of this thesis, the network topology is a directed complete graph $G = (V, E)$ with $n$ vertices. Each vertex in $V$ represents a geo-distributed site, and each edge in $E$ represents the logical link between two sites. Due to a WAN's heterogeneous bandwidth availability, there are two functions $C_u : V \to \mathbb{R}^+$ and $C_d : V \to \mathbb{R}^+$, which respectively define the uplink and downlink capacity of the edge link between a site and the core network. The network latency between each pair of sites is denoted by $L : E \to \mathbb{R}^+$. The uplink and downlink capacity of a site is shared by all unicast data transmissions associated with that site. For instance, a sender directly multicasting to $n$ receivers at the rate of $R$ will consume $nR$ of the sender's uplink capacity and $R$ of each receiver's downlink capacity.

# Chapter 4

# Shallow Overlay Trees Suffice for High-Throughput Consensus

This chapter demonstrates that, when modelling the network as a congestion-free core and leveraging overlay multicast, it is possible for all-to-all data transmission to achieve both high throughput and low latency for data to reach all other nodes. The general idea is similar to SplitStream and Bullet [11, 24], which split the source stream at each node into multiple partitions and broadcast each partition with different overlay trees. I build on this prior work by proving that the optimal data transmission throughput is always achievable by using broadcast trees with height up to two.

## 4.1 Terminology

The section below provides the definition of some terminologies used in this chapter.

#### Overlay

In a network $G(V, E)$, an overlay $O(V, E')$ is a spanning tree of $G$ rooted at some site $v \in V$. It defines a broadcast transmission with site $v$ as the sender and the remaining sites as receivers. Each edge $(v_i, v_j) \in E'$ represents the transmission of $v$'s data from $v_i$ to $v_j$.

### Client Data Stream

In a network $G(V, E)$, a client data stream $s$ is an infinite sequence of data bits from clients to be broadcast to all other sites in the network. The rate $R_i$ of a client data stream $s_i$ represents the incoming rate of client data at site $v_i \in V$. For instance, letting $r$ be the number of incoming client requests per second at site $v$ and letting $S$ be the size of the data payload of each request, the client data rate at site $v$ is $rS$. I assume that a site's client data stream does not consume its downlink capacity as client requests arrive through the local area network. For all-to-all data transmission, each site $v_i \in V$ is associated with a client data stream $s_i$ that must be received by all other sites.

### Partitioning Scheme

Assume for simplicity of analysis that a client data stream can be split at arbitrary fine granularity. A partitioning scheme $P(s_i, n)$ of a client data stream $s_i$ with rate $R_i$ splits elements of $s_i$ into $n$ streams $s_{i,1}, \ldots, s_{i,n}$ with rates $r_{i,1}, \ldots, r_{i,n}$ such that $R_i = \sum_{j=1}^{n} r_{i,j}$. I refer to each split of the stream a *sub-stream* of $s_i$.

### Aggregated Throughput

In an all-to-all data transmission in a network $G(V, E)$ with $n$ sites, each site $v_i$ broadcasts its data to all other sites at the rate $R_i$ without violating the uplink and downlink capacity at any site. Then the aggregated throughput of this all-to-all data transmission is defined as $\sum_{i=1}^{n} R_i$.

### Base Overlays

Base overlay refers to the following types of overlays:

1. 1-level tree
2. 2-level tree with only one non-leaf node (excluding the root)

Figure 4.1 demonstrates these two base overlays in a network with four sites.

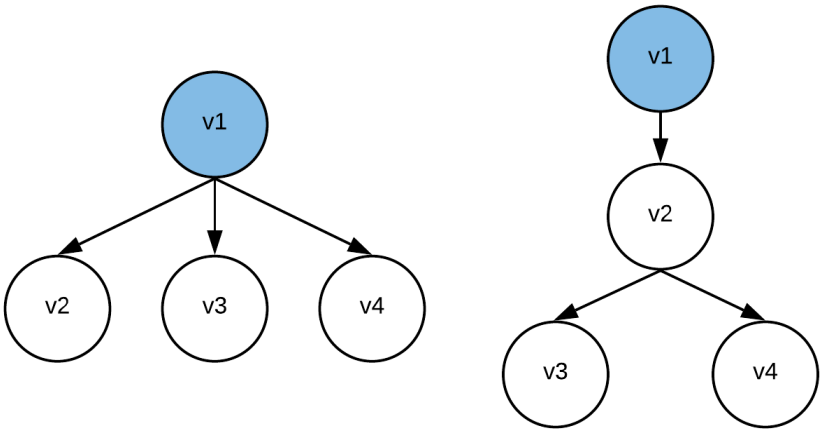


Figure 4.1: Two types of base overlays in a cluster of four nodes.

## 4.2 The Main Property of Sustainable Rates

**Sustainable Rates**

Client data streams $s_1, \ldots, s_n$ with rates $R_1, \ldots, R_n$ are said to be *sustainable* if the following four conditions are all met:

1. For each $v_i \in V$, $R_i \leq C_u(v_i)$.
2. For each $v_i \in V$, $\sum_{j \neq i} R_j \leq C_d(v_i)$.
3. $(n-1) \sum_{i=1}^{n} R_i \leq \sum_{i=1}^{n} C_u(v_i)$.
4. $(n-1) \sum_{i=1}^{n} R_i \leq \sum_{i=1}^{n} C_d(v_i)$.

Intuitively, being sustainable is the minimum requirement for a set of client data streams to be broadcast at their incoming rates. Condition (1) ensures that each site has enough uplink capacity to send out its data at least once to other nodes. As each site has to receive from all other peers, condition (2) ensures that the aggregated rate of incoming streams does not exceed a site's downlink capacity. Condition (3) derives from the fact that the client data at each site must be sent at least $n-1$ times. Similarly, the fact that client data at each site has to be received $n-1$ times lead to condition (4). Note that, condition

(4) is the direct result of condition (2) by summing over all possible $i$. If any of the above conditions are violated, the aggregated throughput of all-to-all data transmission will be less than $(n-1)\sum_{i=1}^{n} R_i$.

**Theorem 4.2.1.** *For client data streams $s_1, \ldots, s_n$ with sustainable rates $R_1, \ldots, R_n$, there exists a partitioning scheme for each client data stream and a one-to-one mapping from sub-streams to overlays such that:*

1. *Each sub-stream can be broadcast using its overlay at its rate without violating downlink and uplink capacity constraints at any site.*
2. *The height of each sub-stream's overlay is at most 2.*

## 4.3 Proof of Theorem 4.2.1

I prove Theorem 4.2.1 by proving the following stronger theorem:

**Theorem 4.3.1.** *For client data streams $s_1, \ldots, s_n$ with sustainable rates $R_1, \ldots, R_n$, there exists a partitioning scheme for each client data stream and a one-to-one mapping from sub-streams to overlays such that:*

1. *Each sub-stream can be broadcast using its overlay at its rate without violating downlink and uplink capacity constraints at any site.*
2. *Each sub-stream's overlay is a base overlay.*

Theorem 4.3.1 is a stronger version of Theorem 4.2.1 because Theorem 4.3.1 constrains overlay candidates to two specific types while there are other alternative forms of 2-level tree overlays. Later sections provide the proof of Theorem 4.3.1. The general idea is to construct a possible partitioning scheme for each client data stream and associate each sub-stream with an overlay such that the resulting data transmission will not violate downlink and uplink capacity constraints at any site.

### 4.3.1 Constructing Sub-stream Overlays

Each client stream $s_i$ will be split into $n$ sub-streams $s_{i,1}, \ldots, s_{i,n}$ with rates $r_{i,1}, \ldots, r_{i,n}$. The data of the special sub-stream $s_{i,i}$ is sent directly from $v_i$ to all the remaining sites. The data of sub-stream $s_{i,j}$ for $i \neq j$ is sent from $v_i$ to $v_j$ first, and then $v_j$ will broadcast the data to the rest of the sites. All overlays defined previously are base overlays.

### 4.3.2 Computing Sub-stream Rates

Table 4.1: Table of notations

| Symbol | Definition |
|---|---|
| $G(V, E)$ | the network topology |
| $s_i$ | the client data stream originates at $v_i \in V$ |
| $R_i$ | the incoming rate of $s_i$ |
| $R'_i$ | the residual rate of $s_i$ to be partitioned |
| $U'_i$ | the reserved uplink capacity of $v_i$ for sending out $s_i$ once |
| $U_i$ | the residual uplink capacity of $v_i$ after reserving $R_i$ for $U'_i$ |
| $s_{i,j}$ | the $j_{th}$ sub-stream of $s_i$ |
| $r_{i,j}$ | the rate of $s_{i,j}$ |

**Algorithm 1:** Sub-stream rate assigning algorithm

**Input :** $G(V, E)$
$C_u : V \rightarrow \mathbb{R}+$
$R_i \dots R_n$

**Output:** $r_{i,j}, 1 \leq i, j \leq n$

```
1  r_{i,j} := 0, 1 ≤ i, j ≤ n; // initialize sub-stream rates
2  U_i := C_u(v_i) − R_i, 1 ≤ i ≤ n; // initialize each residual uplink capacity
3  for i := 1 to n do // go through all source sites
4      R'_i := R_i;
5      for j := 1 to n do // loop through all sub-streams
6          if (n − 2)R'_i > U_j then
7              r_{i,j} := U_j / (n−2);
8          else
9              r_{i,j} := R'_i;
10         U_j := U_j − (n − 2)r_{i,j};
11         R'_i := R'_i − r_{i,j};
12         if R'_i = 0 then
13             break;
14 return r_{1,1} ... r_{n,n};
```

Algorithm 1 computes the rate of each sub-stream defined in the previous section. It does not aim to compute the latency-optimal partitioning scheme, which favours 1-level tree overlays. The purpose of Algorithm 1 is to construct a partitioning scheme and an overlay mapping for proving Theorem 4.3.1. Table 4.1 summarizes the notations used in Algorithm 1.

For each node $v_i$, its uplink capacity is divided into two parts: $U_i' = R_i$ and $U_i = C_u(v_i) - R_i$. $U_i'$ represents the reserved uplink capacity for $v_i$ to send out all its data at least once and $U_i$ is the residual uplink capacity such that $U_i + U_i' = C_u(v_i)$. The algorithm iterates over all site pairs in $\{(i,j)|1 \leq i \leq n, 1 \leq j \leq n\}$ in lexicographical order to compute sub-stream rates. Using such an order is just for the clarity of the proof and has no impact on the correctness of the output of Algorithm 1.

For the iteration when sub-stream rate $r_{i,j}$ is computed, the algorithm greedily allocates as much of $U_j$ as possible to $r_{i,j}$ until either $U_j$ is exhausted or the aggregated sub-stream rate reaches $R_i$. According to the overlay trees defined in the previous section, sending $s_{i,j}$ consumes $r_{i,j}$ of $U_i'$ and $(n-2)r_{i,j}$ of $U_j$. This rule also applies to the case $i = j$, where sending $s_{i,i}$ consumes $r_{i,i}$ of $U_i'$ and $(n-2)r_{i,i}$ of $U_i$. As a result, sending $s_{i,i}$ consumes in total $(n-1)r_{i,i}$ of $C_u(v_i)$.

Figure 4.2 provides the visualization of Algorithm 1. It demonstrates an example with three sites $v_1, v_2, v_3$ with uplink capacity $C_u(v_1) = 2$, $C_u(v_2) = 10$, $C_u(v_3) = 6$. All sites have the same downlink capacity which equals to 9. Each entry represents the overlay associated with the sub-stream $s_{i,j}$. Let the client data stream rates be $R_1 = 1$, $R_2 = 3$, $R_3 = 5$. After the first iteration, $r_{1,1} = 1$, $r_{1,2} = 0$ and $r_{1,3} = 0$. After the second iteration, $r_{2,1} = 0$, $r_{2,1} = 3$ and $r_{2,3} = 0$. After the final iteration, $r_{3,1} = 0$, $r_{3,2} = 4$ and $r_{3,3} = 1$.

### 4.3.3 Correctness Criteria

The output of Algorithm 1 is a set of sub-stream rates $r_{1,1}, \ldots, r_{n,n}$. A correct output satisfies the following three criteria for all $1 \leq i \leq n$:

1. **Valid Partition Constraint:** The aggregated rate of all sub-streams of a client data stream is equal to that client data stream's rate, which is equivalent to $\sum_{j=1}^{n} r_{i,j} = R_i$.

2. **Uplink Capacity Constraint:** The aggregated rate of all sub-streams sent by $v_i$ is less than or equal to $C_u(v_i)$.

3. **Downlink Capacity Constraint:** The aggregated rate of all sub-streams received by $v_i$ is less than or equal to $C_d(v_i)$.

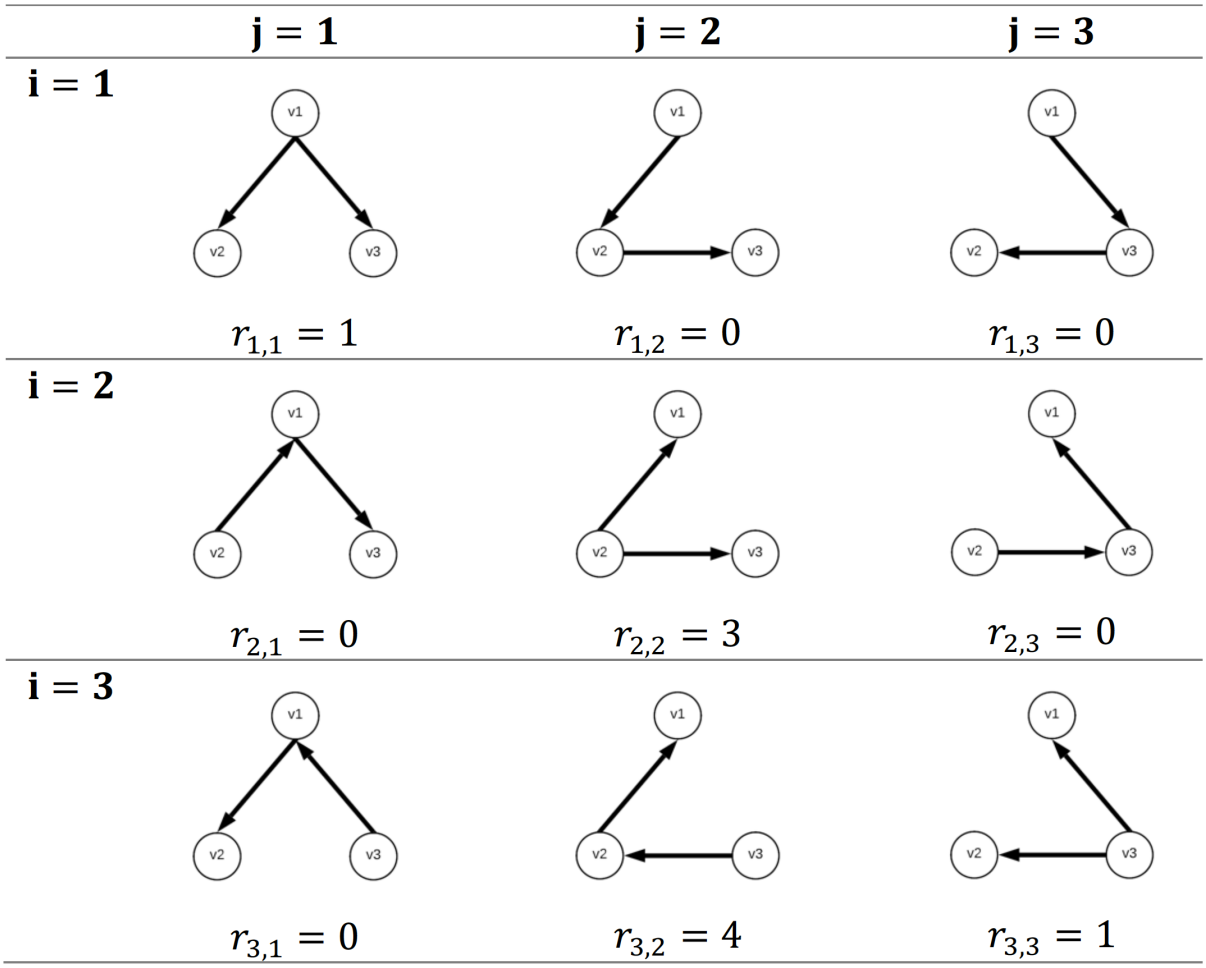


Figure 4.2: Visualization of Algorithm 1.

### 4.3.4 Correctness of Algorithm 1

Let $U_i[\alpha]$ represent the value of $U_i$ at the start of iteration $\alpha$ of the outer loop.

**Proposition 4.3.1.** *For all $\alpha$ such that $1 \leq \alpha \leq n$, if $\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)R_\alpha$, then $\sum_{i=1}^{n} r_{\alpha,i} = R_\alpha$ at the end of iteration $\alpha$ of outer loop.*

*Proof.* This proposition can be proved by contradiction. Assume:

$$\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)R_\alpha \tag{4.1}$$

$$\sum_{i=1}^{n} r_{\alpha,i} \neq R_\alpha \tag{4.2}$$

at the end of iteration $\alpha$ of the outer loop. If that is the case, line 9 is never executed during iteration $\alpha$, otherwise, $R'_\alpha$ becomes 0 at line 11 and the inner loop terminates with $\sum_{i=1}^{n} r_{\alpha,i} = R_\alpha$. At the start of the inner loop's last iteration, we have:

$$R'_\alpha = R_\alpha - \sum_{i=1}^{n-1} r_{\alpha,i} \tag{4.3}$$

Since line 7 is executed at every iteration of the inner loop, at the start of the inner loop's last iteration, we have:

$$R'_\alpha = R_\alpha - \frac{1}{n-2} \sum_{i=1}^{n-1} U_i[\alpha] \tag{4.4}$$

$$(n-2)R'_\alpha > U_n[\alpha] \tag{4.5}$$

Equation 4.4 implies that

$$(n-2)R'_\alpha = (n-2)R_\alpha - \sum_{i=1}^{n-1} U_i[\alpha] \tag{4.6}$$

Therefore, we have:

$$(n-2)R'_\alpha > U_n[\alpha] \implies (n-2)R_\alpha > \sum_{i=1}^{n} U_i[\alpha] \tag{4.7}$$

contradicts with 4.1. □

**Proposition 4.3.2.** $\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)\sum_{i=\alpha}^{n} R_i$, *for all* $\alpha$ *such that* $1 \leq \alpha \leq n$.

*Proof.* This proposition can be proved by induction on $\alpha$.
Base case $\alpha = 1$:

$$\sum_{i=1}^{n} U_i[1] = \sum_{i=1}^{n} (C_u(v_i) - R_i) \tag{4.8}$$

According to condition (3) of sustainable rates:

$$\sum_{i=1}^{n} (C_u(v_i) - R_i) \geq (n-2)\sum_{i=1}^{n} R_i \tag{4.9}$$

This proves the base case.

Induction step: choose an arbitrary $\alpha > 1$ and assume the equation below is true.

$$\sum_{i=1}^{n} U_i[\alpha - 1] \geq (n-2)\sum_{i=\alpha-1}^{n} R_i \tag{4.10}$$

As a result of proposition 4.3.1,

$$\sum_{i=1}^{n} r_{\alpha-1,i} = R_{\alpha-1} \tag{4.11}$$

at the end of the outer loop's iteration $\alpha - 1$. Since line 10 is executed at every iteration of the inner loop, we have:

$$U_i[\alpha] = U_i[\alpha - 1] - (n-2)r_{\alpha-1,i}, \forall i : 1 \leq i \leq n \tag{4.12}$$

According to equation 4.11, by summing over $i$, we have:

$$\sum_{i=1}^{n} U_i[\alpha] = \sum_{i=1}^{n} U_i[\alpha-1] - (n-2)\sum_{i=1}^{n} r_{\alpha-1,i} \tag{4.13}$$

$$= \sum_{i=1}^{n} U_i[\alpha-1] - (n-2)R_{\alpha-1} \tag{4.14}$$

According to equation 4.14, by subtracting $(n-2)R_{\alpha-1}$ from both sides of inequality 4.10, we have:

$$\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)\sum_{i=\alpha}^{n} R_i \tag{4.15}$$

This completes the induction step. □

**Proposition 4.3.3.** *The output of Algorithm 1 satisfies the Valid Partition Constraint.*

*Proof.* This proposition holds as the direct outcome of Proposition 4.3.1 and Proposition 4.3.2.

According to Proposition 4.3.2:

$$\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)\sum_{i=\alpha}^{n} R_i \geq (n-2)R_\alpha, \ \forall \alpha : 1 \leq \alpha \leq n \tag{4.16}$$

and Proposition 4.3.1:

$$\sum_{i=1}^{n} U_i[\alpha] \geq (n-2)R_\alpha \implies \sum_{i=1}^{n} r_{\alpha,i} = R_\alpha \tag{4.17}$$

We have:

$$\sum_{i=1}^{n} r_{\alpha,i} = R_\alpha, \ \forall \alpha : 1 \leq \alpha \leq n \tag{4.18}$$

□

Equation 4.18 implies the Valid Partition Constraint.

**Lemma 4.3.2.** $U_i[\alpha] \geq 0$ *for all* $i, \alpha$ *such that* $1 \leq i, \alpha \leq n$

*Proof.* I prove this lemma by induction on $\alpha$.
Base case $\alpha = 1$: By line 2, we have: $U_i[1] = C_u(v_i) - R_i$ for all $i$ such that $1 \leq i \leq n$. According to condition (1) of sustainable rates, $U_i[1] \geq 0$ holds for all $i$ such that $1 \leq i \leq n$.
Induction step: For an arbitrary number $\alpha$ such that $1 < \alpha \leq n$, assume $U_i[\alpha - 1] \geq 0$. Line 6 and line 7 guarantee $r_{\alpha,i} \leq \frac{U_i[\alpha-1]}{n-2}$ and $U_i[\alpha] = U_i[\alpha - 1] - (n-2)r_{\alpha,i}$ according to line 10. As a result, $U_i[\alpha] \geq 0$. □

**Proposition 4.3.4.** *The output of Algorithm 1 satisfies the Uplink Capacity Constraint.*

*Proof.* From Lemma 4.3.2, we have $U_i \geq 0$ throughout the execution of Algorithm 1 for all $i$ such that $1 \leq i \leq n$. According to the overlay defined in the previous section 4.3.1, sending $s_{i,j}$ consumes $r_{i,j}$ of $U_i'$ and $U_i'$ is consumed only by sending $v_i$'s sub-streams. By Proposition 4.3.3, $\sum_{j=1}^{n} r_{i,j} = R_i$ for all $i$ such that $1 \leq i \leq n$. Since we reserve $R_i$ for $U_i'$, sending all of $v_i$'s sub-streams will consume exactly the amount of its reserved uplink capacity. Because $C_u(v_i) = U_i + U_i'$, no uplink capacity constraint is violated. □

**Proposition 4.3.5.** *The output of Algorithm 1 satisfies the Downlink Capacity Constraint.*

*Proof.* Since every sub-stream is broadcast by a tree overlay, each site receives all other site's data exactly once and the data is received at the same rate as the source is trying to send. According to the condition (2) of sustainable rates, there is also no violation of downlink capacity constraint. □

## 4.4 Discussion

For the sake of simplicity, I define the shallow tree overlay approach as the techniques of splitting up the client data stream and broadcasting each sub-stream with a shallow tree overlay. According to Theorem 4.2.1, any sustainable rates in all-to-all data transmission is achievable by using shallow tree overlays. Comparing to directly broadcasting to other sites, the equation below captures the throughput improvement achieved by the shallow tree overlay approach for sustainable client data rates $R_1, \ldots, R_n$:

$$\frac{\sum_{i=1}^{n} R_i}{\sum_{i=1}^{n} \min(R_i, \frac{C_u(v_i)}{n-1})} \tag{4.19}$$

The numerator is the aggregated throughput achieved by the shallow tree overlay approach while the denominator is the aggregated throughput achieved by direct broadcasting. If some site $v_i$ does not have enough uplink capacity to broadcast its data directly at

the rate $R_i$, using the shallow tree overlay approach results in higher aggregated throughput. Such a case can happen when there is a mismatch between the distribution of client load and the distribution of available uplink capacity. Examples of this situation include but not limited to:

1. All sites have similar uplink capacity and the client load at some site is much higher than the client load at other sites.
2. The client loads at each site are similar and some site has limited uplink capacity compared to other sites.

Equation 4.19 implies that one can use the shallow tree overlay approach to improve the aggregated throughput of all-to-all data transmission for the above scenarios.

## 4.5 Conclusion

According to Theorem 4.2.1, by leveraging overlay multicast, all-to-all data transmission can achieve the best possible throughput by using two-hop paths for data transmission. It provides a theoretical foundation for ruling out deep overlay trees with height greater than two when optimizing data transmissions for applications such as blockchains and consensus protocols. Based on those results, I develop a two-phase optimization algorithm that optimizes data transmission for consensus protocols. At a high level, the first phase computes the optimal data rates based on available network resources. The second phase computes the optimal combinations of overlay trees that yield the lowest latency and provide the data rates obtained in the first phase. The next chapter will describe the algorithm in detail.

# Chapter 5

# ShallowForest: Optimizing All-to-All Data Transmission in WANs

ShallowForest is a two-phase algorithm that optimizes all-to-all data transmission in a WAN environment for consensus protocols and blockchain systems. We assume that network capacity is the critical performance-limiting resource for such systems in a WAN environment. The primary optimization goal of ShallowForest is to maximize the aggregated data transmission throughput while the secondary goal is to minimize the network latency subject to the data transmission rate. As a result, the first phase computes the maximum achievable data transmission throughput constrained by the network capacity and client load across all sites. In the second phase, ShallowForest computes the optimal way to partition each client data stream and associates each sub-stream with an overlay such that the resulting overlay combination achieves the optimal throughput obtained from the first phase. As network delay is not negligible in a WAN environment, the second phase also minimizes the aggregated latency weight of the resulting overlays. In the sections below, we describe the ShallowForest algorithm in detail.

## 5.1 Throughput-Optimal Data Rates

For client data streams with sustainable rates, ShallowForest starts the next phase directly. Otherwise, ShallowForest computes a set of sustainable rates that result in the maximum

aggregated data transmission throughput by using the following LP formulation:

$$\text{maximize:} \quad \sum_{i=1}^{n} R'_i \tag{5.1}$$

$$\text{free variables:} \quad R'_i \qquad \forall i : 1 \le i \le n \tag{5.2}$$

$$\text{subject to:} \quad R'_i \le \min(C_u(v_i), R_i) \qquad \forall i : 1 \le i \le n \tag{5.3}$$

$$\sum_{j \ne i} R'_j \le C_d(v_i) \qquad \forall i : 1 \le i \le n \tag{5.4}$$

$$(n-1) \sum_{i=1}^{n} R'_i \le \sum_{i=1}^{n} C_u(v_i) \tag{5.5}$$

Constraints 5.3–5.5 ensure that the resulting rates meet the three of four conditions of being sustainable. The output of the above formulation is a set of sustainable rates $R'_1, \ldots, R'_n$ with the maximum sum that will be passed to the next phase.

## 5.2 Latency-Optimal Overlays

The goal of this phase is to compute a partitioning scheme for each client data stream and construct overlays for all sub-streams such that the network latency incurred by the data transmission is minimized.

**Latency Weight.** The latency weight $l(O)$ of an overlay $O = (V, E)$ rooted at $v \in V$ is the aggregated network latency incurred by all receivers to receive $v$'s data. Let $P_i \subseteq E$ be the path in $O$ from $v$ to some $v_i \in V$, we have $l(O) = \sum_{v_i \in V} \sum_{e \in P_i} L(e)$.

**Problem Statement.** Given a network $G = (V, E)$ with $C_u : V \to \mathbb{R}^+$, $C_d : V \to \mathbb{R}^+$, $L : E \to \mathbb{R}^+$ and client data streams $s_1 \ldots s_n$ with sustainable rates $R_1 \ldots R_n$, for each client data stream $s_i$, find a partitioning scheme $P(s_i, n_i) = \{s_{i,1}, \ldots, s_{i,n_i}\}$ and the corresponding overlay of each sub-stream $O_{i,1}, \ldots, O_{i,n_i}$ such that:

1. Each sub-stream $s_{i,j}$ can be broadcast at its rate $r_{i,j}$ without violating downlink and uplink capacity constraints at any site.
2. $\sum_{i=1}^{n} \sum_{j=1}^{n_i} l(O_{i,j}) r_{i,j}$ is minimized.

The term $l(O_{i,j})r_{i,j}$ is the product of sub-stream rate and its overlay's latency weight, which represents the network latency subject to the data transmission on $O_{i,j}$ at rate $r_{i,j}$.

Minimizing the sum of this term over all overlays will promote transmitting more data on overlays with low network latency to reduce the average network latency incurred by the entire all-to-all data transmission.

### 5.2.1 Choosing Overlay Candidates

Overlay candidates are expected to be as shallow as possible to minimize the overhead of network latency. As a result of the main property of sustainable rates, it is sufficient to consider overlays with height less than or equal to two in the second phase. I further reduce overlay candidates to base overlays as illustrated in Figure 4.1. The intuition behind choosing these two types is that 1-level trees yield the minimum network latency when the uplink capacity at the root is abundant, while 2-level trees are the most bandwidth efficient when the uplink capacity at the root is scarce. Also, according to Theorem 4.3.1, base overlays are sufficient for achieving any sustainable rates. As a result, each site has $n$ overlay candidates, one 1-level tree and $n-1$ 2-level trees.

### 5.2.2 LP formulation

I start by setting up a partitioning scheme for each client stream and pairing each sub-stream with an overlay. Since there are $n$ overlay candidates for each site, each client data stream $s_i$ will be split into $n$ sub-streams $s_{i,1}, \ldots, s_{i,n}$ with rates $r_{i,1}, \ldots, r_{i,n}$. Those sub-stream rates are the variables to be optimized. I assign an overlay $O_{i,j} = (V, E_{i,j})$ to a sub-stream $s_{i,j}$ such that the data transmission is handled in the following way: (1) the data of $s_{i,i}$ is sent directly from $v_i$ to all the remaining sites; (2) the data of $s_{i,j}$ for $i \neq j$ is sent from $v_i$ to $v_j$ first, and then $v_j$ broadcasts the data to the rest of the sites. The resulting LP formulation is as follows:

$$\text{minimize:} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} l(O_{i,j}) r_{i,j} \tag{5.6}$$

$$\text{free variables:} \quad r_{i,j} \qquad \forall i, j : 1 \leq i, j \leq n \tag{5.7}$$

$$\text{subject to:} \quad U_i \leq C_u(v_i) \qquad \forall i : 1 \leq i \leq n \tag{5.8}$$

$$\sum_{j=1}^{n} r_{i,j} = R_i \qquad \forall i : 1 \leq i \leq n \tag{5.9}$$

$$r_{i,j} \geq 0 \qquad \forall i, j : 1 \leq i, j \leq n \tag{5.10}$$

For all $i, j$ such that $1 \leq i, j \leq n$:

$$U_i = (n-1)r_{i,i} + (n-2)\sum_{j \neq i} r_{j,i} + \sum_{j \neq i} r_{i,j} \tag{5.11}$$

Constraint 5.11 represents the amount of $v_i$'s uplink capacity consumed by the data transmission with respect to the overlay setup. Constraint 5.8 characterizes the uplink capacity constraint at a specific site: the aggregated rates of data sent out of a site should be less than or equal to that site's uplink capacity. Constraint 5.9 enforces the sum of all sub-stream rates being equal to the rate of the original stream. Constraint 5.10 enforces the all sub-stream rates to be non-negative. Since the client data rates output by the first phase are sustainable, and each site receives the data from other sites exactly once according to the overlay setup, we do not need the downlink capacity constraint in the formulation.

Note that, the actual throughput achieved by the overlays computed in this phase is approximate to the optimal throughput due to two reasons: (1) the LP formulations relax the integrality constraint on the rate of each sub-stream. In contrast to the assumption I made in Section 4.1, you cannot split a data stream at a granularity finer than one bit; (2) there could be rounding error when solving the LP problem.

# Chapter 6

# Amoeba Paxos: Making EPaxos Workload-Aware

EPaxos [31] is a state of the art decentralized consensus protocol that achieves equal or better performance than many other protocols in a WAN environment. However, its workload agnostic approach to handle data transmission will lead to sub-optimal performance when dealing with skewed load across replicas. To make EPaxos workload-aware, I build Amoeba Paxos (APaxos) on top of the publicly available EPaxos implementation [32] by applying ShallowForest to the data transmission. This section provides the design and implementation of Amoeba Paxos.

## 6.1 Overview

Figure 6.1 depicts the software architecture. There are three major components in APaxos: the ordering plane, the data plane and a centralized controller. The ordering plane receives incoming client requests and orders them using original EPaxos protocol. Instead of broadcasting messages with client operations directly to other replicas, the ordering plane replaces actual client operations in protocol messages with client operation IDs and offloads the job of broadcasting client operations to a co-located data plane thread.

The data plane broadcasts client operations with specific overlays according to its overlay configuration updated by the controller. The overlay configuration determines how much data to transmit with a specific overlay. The data plane also transmits the client operations received from other replicas based on the overlay information encapsulated in

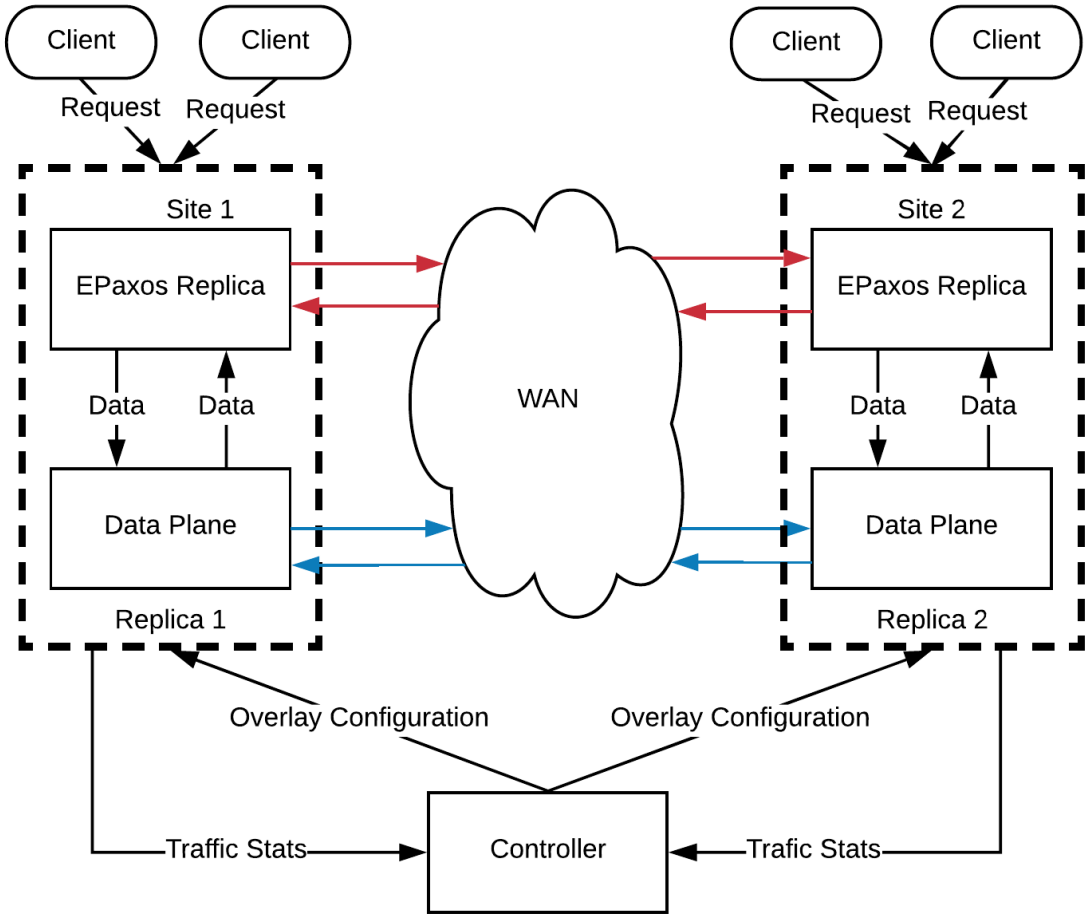


Figure 6.1: The software architecture of APaxos. Red lines represent the transmission of protocol messages and blue lines represent the transmission of client operations.

the received data. Besides handling data transmission, the data plane also buffers the received client operations and reassembles them into protocol messages required by the ordering plane.

The controller applies ShallowForest to compute the optimal partitioning scheme and overlays for each site and updates each site's overlay configuration through RPC calls. In my prototype implementation, I hard-code the client data rates and available network resources in the controller.

## 6.2 The Ordering Plane

There are three types of messages in EPaxos that enclose client operations: `PreAccept`, `TryPreAccept` and `PrepareReply`. The ordering plane replaces client operations in `PreAccept` messages with client operation IDs. The resulting message is referred to as `PreAcceptLight` to distinguish it from the original `PreAccept` message. The ordering plane only separates client operations from `PreAccept` messages because broadcasting `PreAccept` messages consumes the greatest amount of bandwidth in normal operation while the latter two messages are only involved in the EPaxos's recovery process. The ordering plane broadcasts `PreAcceptLight` messages and handles the protocol messages from other replicas in the same way as an EPaxos replica does.

## 6.3 The Data Plane

This section provides further implementation details regarding the data plane.

### 6.3.1 Overcoming the Per-flow Rate Limit

To overcome the per-flow rate limit enforced by public cloud providers, APaxos sets up multiple TCP connections between each pair of replicas located in different areas. For a specific recipient, the data plane picks the TCP connection from the pool in a round-robin fashion and transmits one client operation using a selected TCP connection in a separate thread. The purpose of letting each TCP connection have equal chances to transmit client operations is to make each TCP connection have a similar congestion control window size. With a high incoming rate of client operations, there could be multiple TCP connections concurrently sending client operations to the same recipient.

### 6.3.2 Overlay Configuration

For the data plane thread running on site $v_j$, it sends client operations based on a local overlay configuration `overlay_config`, an array with the same size as the number of overlay candidates. Each entry `overlay_config[i]` is the amount of data out of a configurable window size $w$ KB that should be broadcast using the $i_{th}$ overlay. That is, among $w$ KB of data broadcast by $v_j$, `overlay_config[i]` KB of data should be broadcast using the $i_{th}$ overlay. The $i_{th}$ entry of the overlay configuration is also referred to as the quota of the $i_{th}$ overlay. In my implementation, $w$ is set to 200KB to achieve the best performance.

After computing an optimal solution for the ShallowForest optimization, the controller sets `overlay_config[i]` to $\lfloor \frac{r_{j,i}}{R_j} \rfloor w$. When sending out a client operation $\gamma$ with size $x$ KB, the data plane picks a random overlay with enough quota to send out $\gamma$. If the $i_{th}$ overlay is picked, the data plane subtracts $x$ from `overlay_config[i]`. If no overlay has enough quota to send out $\gamma$ , the data plane will update each entry of `overlay_config` with the latest partitioning scheme computed by the controller. With such an approach, the client data stream is partitioned at the granularity of the size of a client operation.

### 6.3.3 Overlay Information

Another advantage of using only the base overlay candidates demonstrated in Figure 4.1 is minimizing the overhead of overlay information in the data transmitted by the data plane process. To broadcast a client operation $\gamma$, the data plane process simply piggybacks a re-transmission bit to the original message based on its transmission overlay. The bit is set to 0 for a 1-level tree overlay and 1 otherwise. When a data plane process receives data from other replicas, it checks the piggybacked re-transmission bit. If the bit equals 1, the data plane process will multicast the message to the remaining replicas with the re-transmission bit set to 0.

The code snippet below shows the `send` and `recv` procedures implemented in the data plane. The original data plane code is written in golang, which is directly integrated with the EPaxos source code. However, for the clarity of the demonstration, here I use the equivalent Python code instead.

Listing 6.1: Code for sending and receving client operations at the data plane

```
1 #the procedure for sending client operations
2 def send(self):
3     '''
```

```
4       :param self: the data plane thread
5       :return:
6       '''
7       # loop forever until the data plane shuts down
8       for not self.shut_down:
9           # obtain a client operation not yet broadcast
10          client_operation = self.get_client_operation()
11          # get the index of the overlay with enough quota to send
                client_operation
12          i = get_overlay_id(client_operation)
13          # if i equals to the replica ID, the overlay is a 1-level tree
14          if i == self.id:
15              # broadcast client_operation
16              broadcast(client_operation)
17          # otherwise, the overlay is a 2-level tree and replica i is the
                non-leaf node
18          else:
19              # set retransmission bit to 1
20              client_operation.do_retrans = 1
21              # send to replica i only
22              send_to_replica(i, client_operation)
23
24  #the procedure to handle received client operations
25  def recv(self):
26      '''
27      :param self: the data plane thread
28      :return: nothing
29      '''
30      #loop forever until the data plane shuts down
31      for not self.shut_down:
32          # go through all replicas
33          for i in range(self.replica_num):
34              # receive client operation from replica i
35              client_operation = self.recv_client_operation(i)
36              # store the client operation in data plane's local cache
37              self.cache[client_operation.id] = client_operation
38              # the client operation need to be send to other replicas
39              if client_operation.do_retrans == 1:
40                  # set retransmission bit to 0
41                  client_operation.do_retrans = 0
```

```
                # send client_operation
                for i in range(self.replica_num):
                    """
                    send the client operation to replicas
                    other than the source replica
                    """
                    if not i == client_operation.source:
                        send_to_replica(i, client_operation)
```

### 6.3.4 Assemble Ordering Plane Messages

Upon receiving a `PreAcceptLight` message, the data plane assembles a `PreAccept` message by retrieving all client operations referenced by the `PreAcceptLight` message and feeds it to the ordering plane. As client operations are transmitted with different overlays, it is possible that some referenced client operations are not presented in the cache at the time the `PreAcceptLight` message arrives. In such a case, the data plane waits for a configurable period of time for the missing client operations to present in the cache.

The code snippet below shows the `handle_light_preaccept` procedure for handling received `PreAcceptLight` messages. The procedure is triggered and executed in a separate thread whenever the data plane receives a `PreAcceptLight` message.

Listing 6.2: Code for reassembling PreAccept messages

```
#The procedure for handling PreAcceptLight messages
def handle_light_preaccept(self, pre_accept_light):
    '''
    :param self: the data plane thread
    :param pre_accept_light: the PreAcceptLight message object to be processed
    :return: nothing
    '''
    # creating a PreAccept message object from the PreAcceptLight message
    # by copying all fields of PreAcceptLight message
    pre_accept = PreAccept(preaccept_light)
    # go through all operation ids referenced by the PreAcceptLight message
    for operation_id in pre_accept_light.id_list:
        #check if the corresponding client operation is in local cache
        if operation_id in self.cache:
            #add the client operation to PreAccept message
```

```
            pre_accept.command_list.append(self.cache[operation_id])
        else:
            #wait for a period of time for the missing client operation
            time.sleep(WAIT_TIME)
    #once all client operations have been added to the PreAccept message,
    #send it to the ordering plane
    self.feed_to_order_plane(pre_accept)
```

### 6.3.5 Handling Failures

EPaxos does not rely on a controller to determine data transmission overlays. To avoid the single point of failure, APaxos can deploy multiple controllers and each controller can independently compute the optimal partitioning scheme and overlay mapping for each site. When the current controller fails, a backup controller will continue to update each site's overlay configurations.

The other difference between EPaxos and APaxos lies in the way `PreAccept` messages are sent out. EPaxos assumes message passing is asynchronous between replicas and introducing a data plane does not break this assumption. As a result, APaxos inherits the $safety$ property from EPaxos. For the *liveness* property, EPaxos guarantees the client operation will eventually be committed if there are $f+1$ non-faulty replicas and the client retries (possibly with another replica) if it doesn't receive a response within a certain period of time. As a result, the data plane should guarantee all non-faulty replicas finally receive both `PreAcceptLight` and all client operations it references.

Since direct broadcast will guarantee that all non-faulty replicas receive the data, client operations transmitted using a 1-level tree overlay and `PreAcceptLight` require no additional mechanism to ensure data delivery to all non-faulty replicas. However, if a message is transmitted using a 2-level tree overlay, all leaf nodes will not receive the message when the node in the middle crashes. To preserve the property that the protocol can make progress with $f+1$ non-faulty replicas, the middle node sends ACK to the root replica after it completes sending the message to the remaining replicas. If the ACK from the middle node is not received after a period of time, the root node broadcasts the message directly to other replicas and marks the middle node as a potentially crashed node, which needs to be avoided in future data transmissions.

# Chapter 7

# Evaluation

This section presents the evaluation of ShallowForest. The focus of the evaluation is to compare the commit throughput and commit latency of APaxos with and without the ShallowForest optimization. This evaluation omits the performance of original EPaxos because APaxos has superior performance even without the ShallowForest optimization due to other optimizations, such as decoupling the data plane from the ordering plane, parallelizing message serialization, and using multiple TCP connections between each pair of replicas.

## 7.1 Experiment Setup

For the experiment, APaxos is deployed across nine Amazon EC2 regions: Tokyo (**TK**), Singapore (**SG**), Sydney (**SY**), Frankfurt (**FF**), Ireland (**IR**), Oregon (**OR**), Virginia (**VA**), London (**LD**), and California (**CA**). Table 7.1 summarizes the network latency measured using ping between each pair of regions. In each region, the experiment uses an m4.xlarge VM instance with four 2.4 GHz Intel Xeon E5-2676v3 processors and 16 GB main memory. The OS version on each VM is Ubuntu 16.04 and the golang version used to compile APaxos is 1.9.4. A client process and an APaxos replica are executed on each VM, as well as a controller process on a single VM chosen at random. The client process sends requests only to its co-located APaxos replica.

The purpose of the experiment is to evaluate the effectiveness of the ShallowForest optimization when the network is the bottleneck. Therefore, the workload consists of only write requests as they involve broadcasting a significant amount of payload data. All

Table 7.1: Network latency (ms) between each pair of sites used in the experiment.

| | **IR** | **CA** | **VA** | **TK** | **OR** | **SY** | **FF** | **LD** | **SG** |
|---|---|---|---|---|---|---|---|---|---|
| **CA** | 146 | - | | | | | | | |
| **VA** | 74 | 64 | - | | | | | | |
| **TK** | 228 | 113 | 166 | - | | | | | |
| **OR** | 135 | 22 | 79 | 101 | - | | | | |
| **SY** | 275 | 152 | 203 | 116 | 143 | - | | | |
| **FF** | 25 | 149 | 90 | 245 | 165 | 288 | - | | |
| **LD** | 13 | 140 | 80 | 236 | 144 | 274 | 15 | - | |
| **SG** | 184 | 178 | 244 | 74 | 165 | 173 | 179 | 173 | - |

requests are committed on the fast-path as each request is associated with a distinct key. For saturating the network resource provisioned to each VM, the request size is set to 4 KB and 20 TCP connections are established between each pair of VMs. Those parameter values are picked by increasing both request size and the number TCP connections until the throughput of APaxos cannot be improved further.

The client process can be configured to issue requests to an APaxos replica at a specific rate in an open loop. In the experiments, client request rates are enforced to be sustainable. The experiment uses the Zipfian distribution to model the skewed client load across replicas. For the network topology, the experiment uses the average network capacity measured in 1-minute intervals for a total of 30 minutes, and the average latency measured by 3 pings between each pair of replicas. I use `APaxos+SF` to denote APaxos optimized using ShallowForest in all results.

## 7.2 Different Skewness Levels

Workloads in the real world can be highly skewed across different geo-areas and continuously changing over time. Therefore, for the first experiment, I use five geo-distributed replicas to evaluate the effectiveness of ShallowForest in dealing with skewed client load across replicas. I vary the exponent parameter $s$ of the Zipfian distribution to tune the skewness level of client load distribution. Client load is uniformly distributed when $s$ equals to zero and increasing $s$ leads to a more skewed client load distribution. Table 7.2 demonstrates the load on each replica at different skewness levels. For each skewness level,

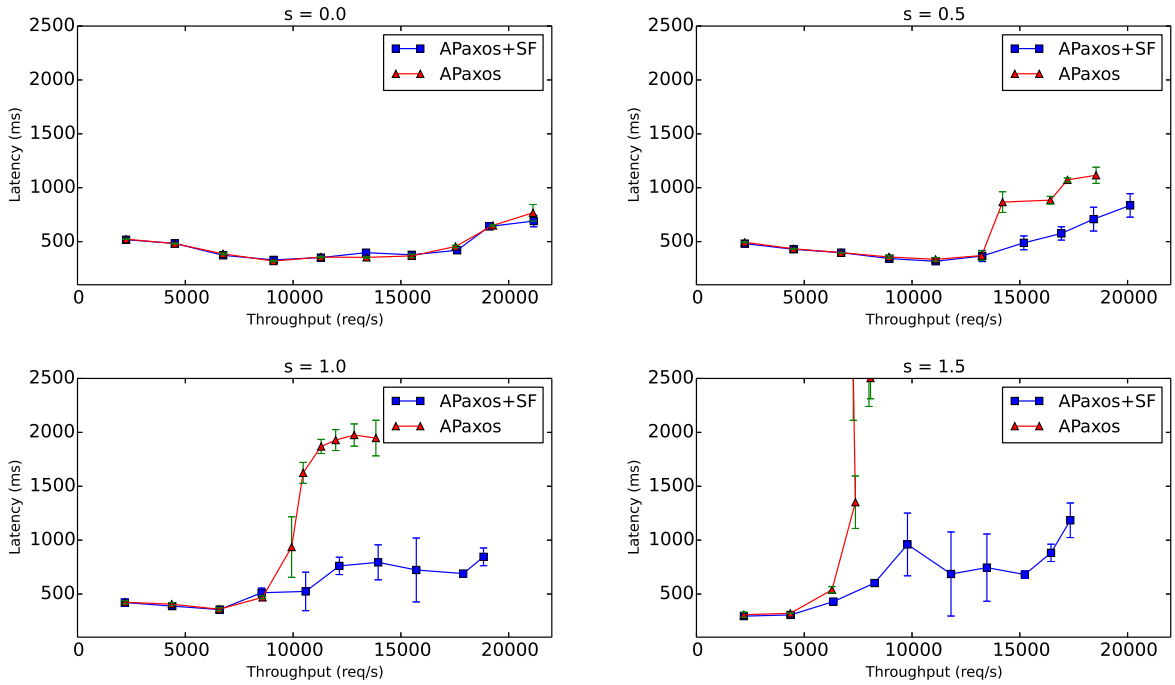


Figure 7.1: Latency vs throughput for 5 replicas under different levels of skewness of client load.

Table 7.2: Load on replicas with under different skewness levels

| $s$ | **IR** | **CA** | **VA** | **TK** | **OR** |
|---|---|---|---|---|---|
| **0.0** | 20% | 20% | 20% | 20% | 20% |
| **0.5** | 38% | 20% | 16% | 13% | 12% |
| **1.0** | 62% | 15% | 9% | 6% | 5% |
| **1.5** | 82% | 8% | 4% | 2% | 1% |

I increase the aggregated client request rate and measure the commit throughput as well as the corresponding average commit latency. Figure 7.1 presents the experimental results where each data point is the average of 4 runs and the error bar represents the standard deviation of 4 runs. Each run lasts for 20 seconds and each VM is warmed by executing the workload for 40 seconds before each run.

As shown in Figure 7.1, the commit latency of APaxos without the ShallowForest optimization grows more rapidly with the increasing commit throughput due to contention for uplink capacity at replicas with high client load. When optimized using ShallowForest, APaxos achieves higher commit throughput with lower commit latency. For instance, when $s = 1.5$, ShallowForest improves the commit throughput of APaxos by 100% with 60% reduction in commit latency.

Figure 7.1 also shows that ShallowForest only improves APaxos slightly when the client load is moderately skewed. When $s = 0.5$, ShallowForest improves the commit throughput of APaxos by 10% with 30% reduction in commit latency. As each VM instance is provisioned with similar uplink and downlink capacity, the optimal overlays become increasingly 1-level tree dominated with a more uniformly distributed client load. This also explains why ShallowForest brings no performance improvement when $s = 0$.

## 7.3 Different Cluster Sizes

I also compare the effectiveness of ShallowForest for various replication factors. For this experiment, I measure the commit throughput of APaxos with five, seven and nine geo-distributed replicas. For all replication factors, I set $s$ to 1 and the aggregated data rate of incoming client requests to 750 Mbps. Table 7.3 summarizes the load on each replica for various replication factors.

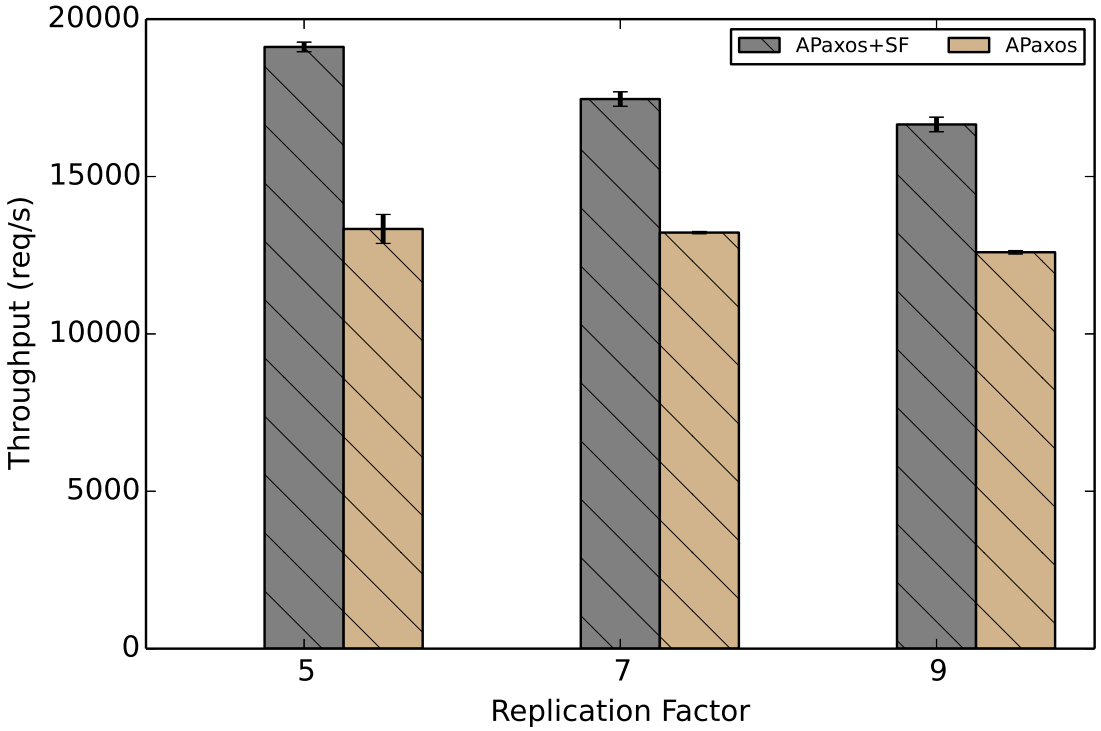


Figure 7.2: Throughput for different numbers of replicas.

Table 7.3: Load on replicas with different replication factors

| RF | **IR** | **CA** | **VA** | **TK** | **OR** | **SY** | **FF** | **LD** | **SG** |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 62% | 15% | 9% | 6% | 5% | | | | |
| 7 | 53% | 15% | 9% | 6% | 5% | 4% | 3% | | |
| 9 | 47% | 15% | 9% | 6% | 5% | 4% | 3% | 3% | 2% |

Figure 7.2 demonstrates the experimental results, where each bar is the average of 4 runs and the error bar represents the standard deviation. ShallowForest improves the commit throughput of APaxos by 43%, 32% and 32% for replication factors 5, 7, and 9. When optimized using ShallowForest, the commit throughput drops faster with the increasing number of replicas. This is due to the higher network latency yielded by 2-level tree overlays in a larger scale deployment, which contains replicas deployed in more distant regions (Sydney and Singapore). However, comparing to APaxos without the optimization, APaxos optimized using ShallowForest still achieves higher commit throughput for all replication factors resulting from more effective use of the network capacity at replicas with low client load.

# Chapter 8

# Related Work

This chapter presents the related work on consensus protocols, overlay multicast protocols, and other techniques for optimizing data flows.

## 8.1 Consensus Over WAN

Mencius [28] is a variant of Paxos that rotates the leader for each command to distribute the load evenly across replicas. Mencius also addresses the issue of unevenly distributed client load across replicas. It allows a replica with low client load to voluntarily skip its leader term to favour replicas with higher client load. Mencius is not completely leaderless and skipping a leader term cannot help a replica with high client load to utilize network resources at a replica with low client load.

E-Paxos [31] further improves scalability and reduces commit latency by removing the role of the leader. Each client is able to send the request to the nearest replica, which is referred to as the command leader. Committing non-conflicting commands requires a single round trip between the command leader and a super-majority of all replicas. A conflicting command requires an extra round of communication for replicating dependencies of conflicting commands to a majority of replicas. However, unlike APaxos, EPaxos does not consider the availability of network capacity and unbalanced client loads across replicas.

Canopus [38] is a network-aware consensus protocol that parallelizes the dissemination of messages according to a leaf only tree (LOT). LOT organizes nodes into several consensus groups based on locality. The main goal of using LOT for data transmission is to minimize the usage of highly contented links. To order commands, different consensus groups first

reach their local consensus and then communicate in parallel with other groups to reach a global consensus. Canopus assumes that all nodes in the same consensus group cannot fail together, otherwise the protocol halts until some nodes in the group recover. In terms of data transmission, LOT might incur higher network latency in a WAN environment. For $n$ nodes, LOT requires $O(\log n)$ transfers for data to reach all nodes, while ShallowForest requires at most two transfers.

## 8.2 Decoupling Data Transmission From Ordering

Decoupling data transmission from ordering is a common technique used by many consensus protocols [7, 30, 29, 12] and blockchain systems. To separate the ordering plane from data transmission, S-Paxos [7] associates each batch of client requests with a unique ID and uses Paxos as its ordering plane protocol to order batch IDs. Disseminating client requests is handled by a separate process. The data plane of S-Paxos is leaderless, meaning that a client may contact any replica to broadcast the request to other replicas. Ring Paxos [30, 29] handles data dissemination using one or more logical ring overlays. To multicast messages to a group of receivers, all servers are placed on a logical ring. The sender just sends data once to its immediate successor and all subsequent receivers store-and-forward the message until the last receiver receives it. Logical ring overlay minimizes the data replication at the sender to achieve high throughput data transmission and the optimal network utilization. However, when using a ring overlay, the network latency of the data transmission grows proportionally with the number of participants. Both Ring Paxos and S-Paxos handle the data dissemination with a static overlay which does not change adaptively to various client loads across replicas. Some permissionless blockchain systems [33, 10] use gossip protocols [18] for high-throughput data dissemination. ShallowForest is not applicable to those systems as it requires the location and identity of each participant to be known apriori.

## 8.3 Application-Level Multicast

Application level multicast [11, 24, 35] has been studied extensively for content distribution in P2P networks. Among those systems, ShallowForest is most similar to SplitStream and Bullet network [11, 24]. SplitStream [11] partitions the data to be broadcast into several disjoint sections called stripes and constructs a separate broadcast tree for each stripe. To receive the complete stream, a node must be presented in every broadcast tree. SplitStream

enforces that any two broadcast trees must be interior-node-disjoint, which means every node is an interior node in precisely one tree and a leaf node in all other trees. This property improves the robustness of the system because the failure of a node only affects the delivery of a single stripe. Bullet [24] divides the data into multiple disjoint blocks which are further divided into packet-size objects. For data dissemination, Bullet uses an epoch-based algorithm called RanSub for membership management and overlay construction. Within each epoch, RanSub collects newly joined peers and distributes a random subset of peers collected in previous epochs to each participating node. RanSub also organizes all nodes discovered in an epoch into a single overlay tree rooted at the source. For data dissemination, each participating node receives a subset of the data objects of the source from its parent and request the remaining data objects from peers in the random subsets received in each epoch.

Since both protocols are designed for P2P networks, overlays are constructed in a decentralized fashion. They also have a complicated mechanism to improve the robustness of the system when the node can join and leave at any time. Both protocols focus on optimizing data transmission throughput and do not impose any constraints on the height of overlays. ShallowForest only uses shallow tree overlays and optimizes both the throughput and the network latency subject to the data transmission.

## 8.4 Optimizing Data Flows

Varys [16] and Orchestra [15] optimize coflows [14], which are a group of collective flows sharing a common computation goal, such as shuffling in map-reduce systems. Hedera [4] uses simulated annealing to optimize flow scheduling in Intra-datacenter network. The optimization objectives of those systems are the flow completion time [16, 15] and the bisection bandwidth [4], which do not fit into the settings of a consensus protocol.

Network coding [39, 3] is a technique to reduce the bandwidth consumption of multicast by allowing end hosts to not only store and forward but also encode incoming data for transmission. The encoded data will be decoded at the destination host. On the basis of network coding, [27] uses linear programming to compute the optimal throughput of multiple concurrent multicast communications. The core idea of this paper is to introduce conceptual flows in the linear programming formulation, which are network flows that co-exist in the network without contending for link capacities.

Although network coding can provide theoretically optimal throughput, it is not an ideal technique for optimizing data transmission for consensus protocols. First, encoding

and decoding data introduce additional computation overhead. Second, a node must wait for the presence of all involved data partitions before encoding them and sending them out, which further increases the latency of the data transmission.

# Chapter 9

# Conclusion

In this thesis, I present the ShallowForest algorithm and apply it to optimize data transmission in a WAN environment for the EPaxos consensus protocol. The key idea of ShallowForest is to partition the data stream and use shallow tree overlays for data transmission. The experimental results demonstrate that the performance of consensus protocols can become more resilient to the uneven distribution of client load by handling the data transmission in a more workload-aware and network-aware manner.

ShallowForest enhances consensus protocols from the angle of optimizing data transmission overlays. The integration of ShallowForest with consensus protocols is still at its early stage. For instance, the prototype controller hard-codes the available network capacity and the client load at each replica. A fully autonomous controller should be able to estimate client load and available network capacity at each replica.

This thesis leaves open several research problems. First of all, ShallowForest relies on the hose network model, and I want to generalize the ShallowForest optimization to be applicable in other network models. Second, it is worth considering whether shallow tree overlays can also achieve the optimal throughput for other types of data transmission in a hose network model. For instance, instead of sending data to all other receivers, some sites just send their data to a subset of receivers. Last but not least, it would be interesting to know how much benefit in terms of network latency ShallowForest can gain from considering additional overlay candidates other than base overlays in the second phase.

# References


[1] Amazon Web Services (AWS). https://aws.amazon.com/.

[2] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr.

[3] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Trans. Inf. Theor.*, 46(4):1204–1216, September 2006.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, pages 19–19, 2010.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the 13th ACM EuroSys Conference*, pages 30:1–30:15, 2018.

[6] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., USA, 2004.

[7] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 31st IEEE Symposium on Reliable Distributed Systems*, pages 111–120, 2012.

[8] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[9] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[10] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014. Accessed: 2018-06-22.

[11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 298–313, 2003.

[12] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186, 1999.

[13] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, pages 407–424, 2016.

[14] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[15] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM'11 Conference*, pages 98–109, 2011.

[16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. *SIGCOMM Comput. Commun. Rev.*, 44(4):443–454, August 2014.

[17] David Clark, Steven Bauer, kc claffy, Amogh Dhamdhere, Bradley Huffaker, William Lehr, and Matthew Luckie. Measurement and Analysis of Internet Interconnection and Congestion. In *Telecommunications Policy Research Conference (TPRC)*, 2014.

[18] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[19] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. A flexible model for resource management in virtual private networks. *SIGCOMM Comput. Commun. Rev.*, 29(4):95–108, August 1999.

[20] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

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

[22] Osama Haq, Mamoon Raja, and Fahad R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *Proceedings of the 26th International Conference on World Wide Web*, pages 253–262, 2017.

[23] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 11–11, 2010.

[24] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. *SIGOPS Oper. Syst. Rev.*, 37(5):282–297, October 2003.

[25] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, April 1984.

[26] Leslie Lamport. Paxos made simple. pages 51–58, 2001.

[27] Zongpeng Li, Baochun Li, and Lap Chi Lau. On achieving maximum multicast throughput in undirected networks. *IEEE/ACM Trans. Netw.*, 14(SI):2467–2485, June 2006.

[28] 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, 2008.

[29] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2012.

[30] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 2010*

*IEEE/IFIP International Conference on Dependable Systems Networks*, pages 527–536, 2010.

[31] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[32] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Epaxos source code. https://github.com/efficient/epaxos, 2014.

[33] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[34] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of 2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[35] Olga Papaemmanouil, Yanif Ahmad, Uğur Çetintemel, John Jannotti, and Yenel Yildirim. Extensible optimization in overlay dissemination trees. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 611–622, 2006.

[36] Valerio Persico, Alessio Botta, Antonio Montieri, and Antonio Pescape. A first look at public-cloud inter-datacenter network performance. In *Proceedings of the 2016 IEEE Global Communications Conference*, pages 1–7, 2016.

[37] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 421–434, 2015.

[38] 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.

[39] Peter Sanders, Sebastian Egner, and Ludo Tolhuizen. Polynomial time algorithms for network information flow. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 286–294, 2003.

[40] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.

[41] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing deadlines for inter-datacenter transfers. In *Proceedings of the 10th European Conference on Computer Systems*, pages 20:1–20:14. ACM, 2015.