Optimizing All-to-All Data Transmission in WANs

Hao Tan
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
hao.tan@uwaterloo.ca

Wojciech Golab
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
wgolab@uwaterloo.ca

***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 paper investigates the problem of optimizing all-to-all data transmission in a wide area network (WAN) using overlay multicast. We prove that in a hose 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. Upon this foundation, we 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.**

***Index Terms*—network overlays, multicast, consensus, state machine replication**

I. Introduction

Being highly available in the presence of machine failures and network partitions is crucial to today's network services. State machine replication (SMR) [1] 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 [2], [3]. Recently, blockchain [4], [5], [6], [7] 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.

Leader-centric consensus protocols like Paxos [8] and Raft [9] 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 [10], [11] 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 [12], [13], [14] 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, we argue that data dissemination should be handled in a more flexible way for consensus protocols to achieve high commit throughput and low commit latency.

In light of these challenges, we propose *ShallowForest*, an algorithm that optimizes data transmission for consensus protocols using overlay broadcast. ShallowForest computes data

Authors supported in part by a Ripple Graduate Fellowship, Ripple Faculty Fellowship, and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

978-1-7281-6680-3/20/$31.00 ©2020 IEEE

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 [15], [16], 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 backed by the sufficiency of shallow tree overlays in achieving the optimal throughput of all-to-all data transmission in a hose network model [17].

In this paper, we make the following contributions:

1) We demonstrate that the optimal throughput of all-to-all data transmission is achievable by using tree overlays with height up to two in a hose network model.
2) We formulate the data transmission overlay optimization problem as a linear programming (LP) problem.
3) We apply ShallowForest to Egalitarian Paxos (EPaxos) and conduct experiments in the Amazon Elastic Compute Cloud (EC2). The experimental 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.

II. Preliminaries

A. Network Model

In this paper, we use the hose network model [18] to represent the WAN. The hose model abstracts the network as a set of sites connected by a core network with unlimited capacity. All sites can send and receive data from each other, bottlenecked only by the edge link capacity between each site and the core network. The network topology is represented as 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 $C_u : V \rightarrow \mathbb{R}^+$ and $C_d : V \rightarrow \mathbb{R}^+$, 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 $L : E \rightarrow \mathbb{R}^+$. The uplink and downlink capacity of a site are 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. As explained in [17], this model of the network complies with empirical measurements conducted in the Amazon cloud.

B. Terminology

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. 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 b be the size of each request, the client data rate at site v is rb. We 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 $\sum_{j=1}^{n} r_{i,j} \leq R_i$. Each split of the stream is referred to as 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 $\sum_{i=1}^{n} R_i$.

C. Motivating Examples

In a network consisting of n geo-distributed sites $v_1, \ldots, v_n$, consider the case where each site has equal uplink and downlink capacities equal to B Mbps and the network latency between each pair of sites is L ms. Assume the incoming rate of client data is B Mbps at v_1 and zero elsewhere. v_1 needs to broadcast the data to all other sites.

Example 1. *Have v_1 send the data directly to each site, each site can only receive the data at the rate of $\frac{B}{n-1}$ and the latency incurred by each receiver to receive each bit of data equals to L ms.*

Example 2. *Transfer data on a path joining all sites starting at v_1 such that data transmission only happens between adjacent sites. Using a path overlay, v_1 broadcasts data at the rate of B Mbps. However, the communication latency incurred by the last site on the path equals to $(n-1)L$ ms.*

The above examples demonstrate the suboptimality of using a single overlay for data dissemination in terms of either throughput or latency. The example below shows how to achieve the optimal throughput without significantly compromising latency using multiple overlays:

Example 3. *Equally partition the incoming data stream 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*

Although the above case considers a simplified network which involves only one sender, it raises a fundamental problem: given a network comprising of a set of nodes, with 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?

III. ShallowForest Algorithm

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 collection of overlays 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.

A. Throughput-Optimal Broadcast Rate

During the first phase, ShallowForest computes the maximum achievable aggregated broadcast throughput R_{total}. We first demonstrate under what conditions R_{total} becomes achievable in a network with limited resources.

Definition 1. *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) $\forall v_i \in V,\ R_i \leq C_u(v_i)$
2) $\forall v_i \in V,\ \sum_{j \neq i} R_j \leq C_d(v_i)$
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 R_{total} will be less than $\sum_{i=1}^{n} R_i$.

Therefore, if client data streams already have sustainable rates, $R_{total} = \sum_{i=1}^{n} R_i$. Otherwise, it is impossible to broadcast all client data streams at their incoming rates. In such a case, R_{total} is computed using the following LP formulation:

$$\text{free variables:}\quad R'_i \quad 1 \leq i \leq n \tag{1}$$

$$\text{maximize:}\quad R_{total} = \sum_{i=1}^{n} R'_i \tag{2}$$

$$\text{subject to:}\quad R'_i \leq \min(C_u(v_i), R_i) \quad \forall i : 1 \leq i \leq n \tag{3}$$

$$\sum_{j \neq i} R'_j \leq C_d(v_i) \quad \forall i : 1 \leq i \leq n \tag{4}$$

$$(n-1)\sum_{i=1}^{n} R'_i \leq \sum_{i=1}^{n} C_u(v_i) \tag{5}$$

In the above LP formulation, variables $R'_1, \ldots, R'_n$ represent some set of sustainable client data rates. Equation 2 defines the optimization objective. Constraints 3–5 ensure that $R'_1, \ldots, R'_n$ meet the first three conditions of being sustainable. After computing R_{total}, ShallowForest proceeds to the next phase.

B. Latency-Optimal Overlays

The goal of the second phase is to compute a partitioning scheme for each client data stream that achieves the aggregated broadcast throughput R_{total}, and construct overlays for all sub-streams such that the network latency incurred by the data transmission is minimized.

Definition 2. *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 $C_u : V \rightarrow \mathbb{R}^+$, $C_d : V \rightarrow \mathbb{R}^+$, $L : E \rightarrow \mathbb{R}^+$, client data streams $s_1 \ldots s_n$ with rates $R_1 \ldots R_n$, and a target aggregated broadcast throughput R_{total}, 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) $\sum_{i=1}^{n} \sum_{j=1}^{n_i} r_{i,j} = R_{total}$.
3) $\sum_{i=1}^{n} \sum_{j=1}^{n_i} l(O_{i,j}) r_{i,j}$ 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 over $O_{i,j}$ 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.

1) Choosing Overlay Candidates: It is impractical to consider all possible types of overlays due to their sheer number. However, overlay candidates used by the second phase have a critical impact on the resulting aggregated latency weight.

Selected overlay candidates must not impair achieving the R_{total} computed by the first phase, and are expected to be as shallow as possible to minimize the overhead of network latency.

Definition 3. ***Base overlay*** *refers to the following types of overlays: 1-level tree, or 2-level tree with exactly one non-leaf node (excluding the root).*

Definition 4. *Let S be a set of sub-streams and $\mathbb{O}$ be the set of all overlays in a network $G(V, E)$. A one-to-one mapping $f : S \to \mathbb{O}$ is said to be* ***sustainable*** *if each sub-stream $s \in S$ can be broadcast at its rate using $f(s)$ without violating downlink and uplink capacity constraints at any site.*

Theorem 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 sustainable mapping from sub-streams to overlays such that:*

1) Each sub-stream's overlay is a base overlay.
2) The resulting aggregated throughput equals to $\sum_{i=1}^{n} R_i$

Figure 1 demonstrates these two base overlays in a network with four sites. The general idea for proving Theorem 1 is to construct a partitioning scheme for each client data stream and associate each sub-stream with a base overlay such that the resulting data transmission will not violate downlink and uplink capacity constraints at any site. Based on Theorem 1, it is sufficient to only consider base overlays to achieve the optimal aggregated broadcast throughput.

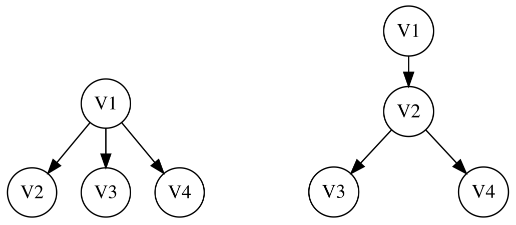

Fig. 1. Two types of base overlays in a cluster of four nodes.

2) LP formulation: We first set up a partitioning scheme for each client stream, and pair 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. We 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:

$$\text{free variables:} \quad r_{i,j} \quad \forall i, j : 1 \le i, j \le n \tag{6}$$

$$\text{minimize:} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} l(O_{i,j}) r_{i,j} \tag{7}$$

$$\text{subject to:} \quad U_i \le C_u(v_i) \quad \forall i : 1 \le i \le n \tag{8}$$

$$\sum_{j \neq i} r_{j,i} \le C_d(v_i) \quad \forall i : 1 \le i \le n \tag{9}$$

$$\sum_{j=1}^{n} r_{i,j} \le R_i \quad \forall i : 1 \le i \le n \tag{10}$$

$$r_{i,j} \ge 0 \quad \forall i, j : 1 \le i, j \le n \tag{11}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} r_{i,j} = R_{total} \tag{12}$$

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

$$U_i = (n-1) r_{i,i} + (n-2) \sum_{j \neq i} r_{j,i} + \sum_{j \neq i} r_{i,j} \tag{13}$$

Constraint 13 represents the amount of v_i's uplink capacity consumed by the data transmission with respect to the overlay setup. Constraint 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 9 characterizes the downlink capacity constraint at a specific site: the aggregated rates of data received by a site should be less than or equal to that site's downlink capacity. Constraint 10 enforces the sum of sub-stream rates being less than or equal to the rate of the original stream. Constraint 11 enforces all sub-stream rates to be non-negative. Constraint 12 enforces the sum of all sub-stream rates equals to be R_{total}, which is computed in the first phase (see Section III-A).

Note that the actual throughput achieved by the overlays computed in this phase is approximate to the optimal throughput for two reasons: (1) The LP formulations relax the integrality constraint on the rate of each sub-stream; in reality, you cannot split a data stream at a granularity finer than one bit. (2) A software LP solver may introduce rounding error.

IV. Amoeba Paxos: Workload-Aware Consensus

EPaxos [13] is a state-of-the-art decentralized consensus protocol that performs favourably 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, we build Amoeba Paxos (APaxos) on top of the publicly available EPaxos implementation [19] by applying ShallowForest to the data transmission.

A. Overview

Figure 2 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 the 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

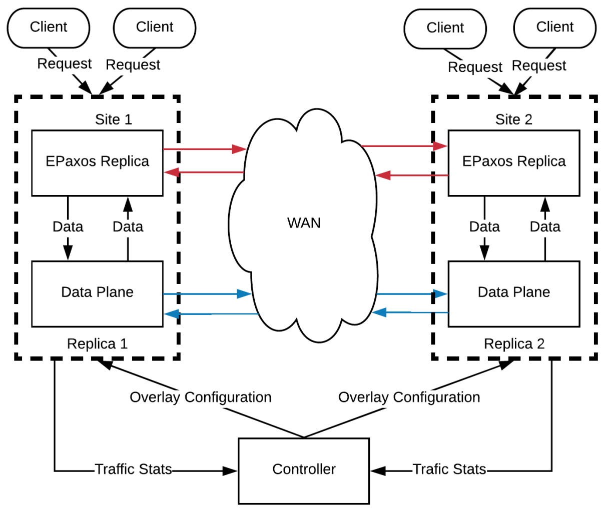

Fig. 2. The software architecture of APaxos. Red lines represent protocol messages and blue lines represent client operations.

controller. The overlay configuration determines how much data to transmit using a specific overlay. The data plane also transmits the client operations received from other replicas based on the overlay information encapsulated in 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 the prototype implementation, we hard-code the client data rates and available network resources in the controller.

B. 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.

C. The Data Plane

This section presents salient details of the data plane.

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.

2) Overlay Configuration: The data plane thread running on site v_j 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 ith overlay. That is, among w KB of data broadcast by v_j, `overlay_config[i]` KB of data should be broadcast using the ith overlay. The ith entry of the overlay configuration is also referred to as the quota of the ith overlay. In our implementation, w is set to 200KB to achieve the best performance.

3) Overlay Information: Another advantage of using only base overlay candidates is minimizing the overhead of overlay information in the data transmitted by the data plane process. To broadcast a client operation γ, 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 broadcast the message to the remaining replicas with the re-transmission bit set to 0.

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 present 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 appear in the cache.

D. 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 does not receive a response within a timeout period. 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 timeout period, 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.

V. Evaluation

This section presents the evaluation of ShallowForest, particularly the benefit of the ShallowForest optimization in terms of commit throughput and commit latency.

A. Experiment Setup

TABLE I
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	-

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 I 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 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. `APaxos+SF` denotes APaxos optimized using ShallowForest in all results.

B. The Effect of Using Multiple TCP flows

Besides the ShallowForest optimization, APaxos differs from original EPaxos by using multiple flows between each replica for data transmission. This experiment evaluates the effectiveness of using multiple TCP flows by comparing the performance of original EPaxos and APaxos using a different number of flows between each pair replicas. We compare the performance of three candidates: EPaxos, APaxos using 5 TCP connections and APaxos using 20 TCP connections. For each candidate, the experiment uses five replicas located in IR, CA, VA, TK, and OR. The client loads on all replicas are equivalent.

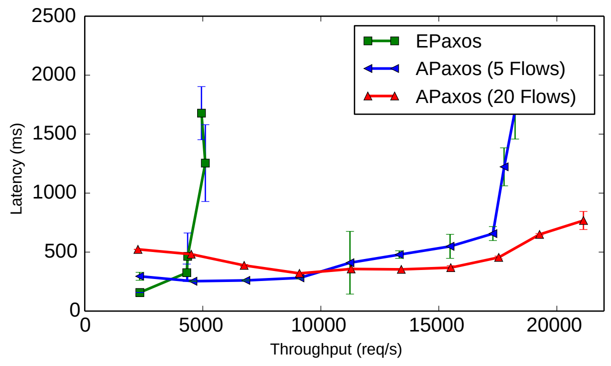

Fig. 3. Latency vs. throughput for 5 replicas with different flow numbers.

Figure 3 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 the VM is warmed up through executing the workload for 20 seconds before each run. The result demonstrates that using a larger number of flows leads to a higher commit throughput. APaxos using 20 TCP connections achieves approximately 4.2X throughput compared to EPaxos, which uses only one TCP connection between each pair of replicas. We also note that the candidate using a smaller number of TCP connections can achieve lower commit latency for low client load in Figure 3. The reason behind it is that the candidate using a smaller number of TCP connections has higher load per connection. When the client load is low, the congestion control window of a TCP connection grows and recovers faster, which results in lower transmission delay.

C. *Different Skewness Levels*

TABLE II
LOAD ON REPLICAS 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%

Workloads in the real world can be highly skewed across different geo-areas, and continuously changing over time. For this experiment, we use five geo-distributed replicas to evaluate the effectiveness of ShallowForest in dealing with skewed client load across replicas. We 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 II demonstrates the load on each replica at different skewness levels. For each skewness level, we increase the aggregated client request rate and measure the commit throughput as well as the corresponding average commit latency. Figure 4 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 4, 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 4 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 load.

D. *Effect of Replication Factor on Performance*

We also compare the effectiveness of ShallowForest for various replication factors. For this experiment, we measure the commit throughput of APaxos with five, seven and nine geo-distributed replicas. For all replication factors, we set s to

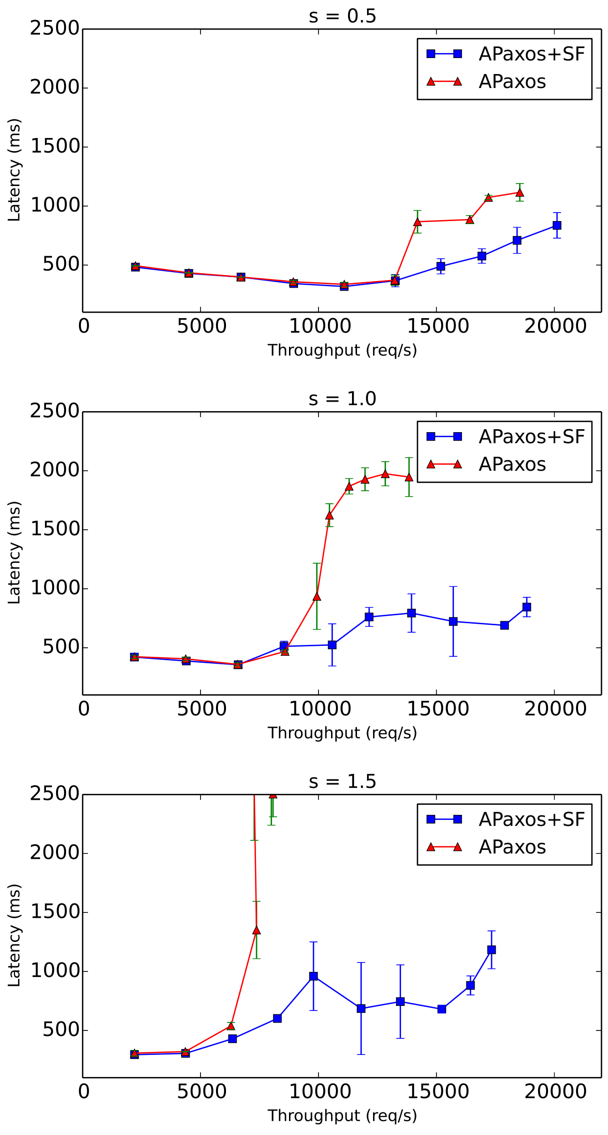

Fig. 4. Latency vs. throughput for 5 replicas with Zipfian workloads.

1 and the aggregated data rate of incoming client requests to 750 Mbps. Table III summarizes the load on each replica for various replication factors.

TABLE III
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 5 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

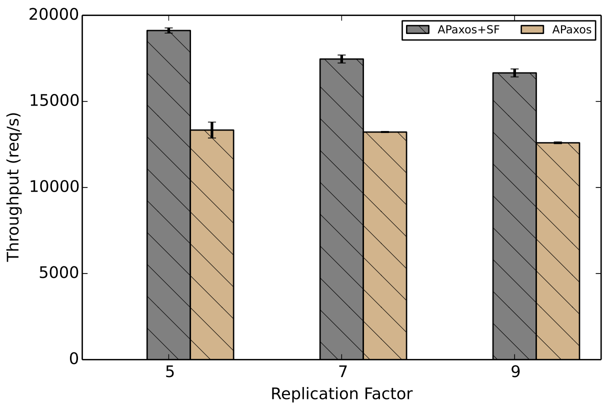

Fig. 5. Throughput of different numbers of replicas.

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, 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.

VI. Related Work

Consensus Over WAN Mencius [12] 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 [13] 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. However, unlike APaxos, EPaxos does not consider the availability of network capacity and unbalanced client loads across replicas. Canopus [20] 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. 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.

Decoupling Data Transmission From Ordering Decoupling data transmission from ordering is a common technique used by many consensus protocols [14], [10], [11] and blockchain systems. To separate the ordering plane from data transmission, S-Paxos [14] 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 [10], [11] 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 that does not change adaptively to various client loads across replicas. Some permissionless blockchain systems [4], [5], [21] use gossip protocols [22] for high-throughput data dissemination. ShallowForest does not apply to those systems as it requires the location and identity of each participant to be known a priori.

Application-Level Multicast Application level multicast [15], [16], [23] has been studied extensively for content distribution in P2P networks. Among those systems, ShallowForest is most similar to SplitStream and Bullet network [15], [16]. SplitStream [15] 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 [16] 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. 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.

VII. Conclusion

In this paper, we presented a method of optimizing data transmission in a WAN environment, called ShallowForest, and applied to the widely-cited 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 ShallowForest can make a consensus protocol more resilient to skewed load by handling the data transmission in a more workload-aware and network-aware manner. In future work, we plan to build a fully autonomous controller that can automatically estimate client load and available network capacity.

References

[1] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems." *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 2, pp. 254–280, Apr. 1984.

[2] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 335–350.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. of the 2010 USENIX Annual Technical Conference*, 2010.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[5] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[6] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. of the 13th ACM EuroSys Conference*, 2018, pp. 30:1–30:15.

[7] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121)*, pp. 51–58, December 2001.

[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of the 2014 USENIX Annual Technical Conference*, 2014, pp. 305–319.

[10] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. of the 2010 IEEE/IFIP International Conference on Dependable Systems Networks*, 2010, pp. 527–536.

[11] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring Paxos," in *Proc. of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012, pp. 1–12.

[12] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for wans," in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 369–384.

[13] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

[14] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-Paxos: offloading the leader for high throughput state machine replication," in *Proc. of the 31st IEEE Symposium on Reliable Distributed Systems*, 2012, pp. 111–120.

[15] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: High-bandwidth multicast in cooperative environments," in *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 298–313.

[16] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 282–297, Oct. 2003.

[17] Tan, Hao, "Shallowforest: Optimizing all-to-all data transmission in wans," 2019. [Online]. Available: http://hdl.handle.net/10012/14690

[18] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 95–108, Aug. 1999.

[19] I. Moraru, D. G. Andersen, and M. Kaminsky, "Epaxos source code," https://github.com/efficient/epaxos, 2014.

[20] S. Rizvi, B. Wong, and S. Keshav, "Canopus: A scalable and massively parallel consensus protocol," in *Proc. of the 13th International Conference on Emerging Networking Experiments and Technologies*, 2017, pp. 426–438.

[21] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation*, 2019, pp. 95–112.

[22] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987, pp. 1–12.

[23] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, J. Jannotti, and Y. Yildirim, "Extensible optimization in overlay dissemination trees," in *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006, pp. 611–622.