PAPER

D-Paxos: Building Hierarchical Replicated State Machine for Cloud Environments

Fagui LIU[†], ***Nonmember and*** **Yingyi YANG**[†a)], ***Member***

SUMMARY We present a hierarchical replicated state machine (H-RSM) and its corresponding consensus protocol D-Paxos for replication across multiple data centers in the cloud. Our H-RSM is based on the idea of parallel processing and aims to improve resource utilization. We detail D-Paxos and theoretically prove that D-Paxos implements an H-RSM. With batching and logical pipelining, D-Paxos efficiently utilizes the idle time caused by high-latency message transmission in a wide-area network and available bandwidth in a local-area network. Experiments show that D-Paxos provides higher throughput and better scalability than other Paxos variants for replication across multiple data centers. To predict the optimal batch sizes when D-Paxos reaches its maximum throughput, an analytical model is developed theoretically and validated experimentally.

key words: *replication, replicated state machine, consensus protocol, batching, logical pipelining*

1. Introduction

Replication is necessary if cloud storage systems are to guarantee fault tolerance and to increase the level of availability. In planet-scale services and applications, it is insufficient to provide fault tolerance with redundant replicas on commodity machines within a single data center. For example, tolerance to the outage of a single data center due to problems such as network partitions or facility-wide outages, is now considered essential for big-data applications which process massive datasets across geographically distributed data centers. Many industrial cloud storage systems, such as Google's Spanner [1], Yahoo!'s PNUTS [2] and Facebook's Cassandra [3], are deployed worldwide and replicate data across multiple continents to serve a very large number of clients nowadays. However, replication at wide scale is known to be very expensive in terms of performance.

To achieve fault-tolerant and highly available cloud storage services across multiple data centers, the replicated state machine (RSM) approach [4] is widely used. Paxos [5] is one of the best known protocols for implementing a RSM. Despite their simplicity, traditional RSM and Paxos are still not ideal for replication among servers distributed across multiple data centers, each holding a large number of redundant replicas. This is due to two main problems. The first is the high latency of a wide-area network (WAN). Paxos's performance is affected by many factors, such as network bandwidth and latency, especially when it is executed among multiple data centers. Due to the high latency of wide-area channels, the unique leader in Paxos must spend a great deal of time idle waiting for responses from a quorum of acceptors during execution among data centers. The second problem is the unbalanced link dependency pattern. When executing Paxos among data centers, only replicas that are located on the same data center as the leader communicate with it exclusively via a low-latency local-area network (LAN), whereas those located on other data centers rely mainly on a higher-latency wide-area network to establish a connection to the leader, which leaves the local-area network within every other data centers idle for most of the time.

To mitigate the problems mentioned above, we define a hierarchical replicated state machine (H-RSM) and design a corresponding protocol D-Paxos ('D' stands for delegator) to improve the utilization of idle resources. The H-RSM is designed based on the idea of parallel processing and aims to improve the resource utilization. It provides consistency guarantees similar to the one-copy serializability used in databases. To execute D-Paxos among multiple data centers, a distinguished replica server called delegator is elected for H-RSM in each data center. The advantages that D-Paxos improves resource utilization and provides high throughput mainly consist in two aspects. On one hand, all delegators generate pre-ordered sequences of requests by efficiently utilizing idle local-area bandwidth. Sequences are then proposed as batches to improve the overall throughput among data centers (batching). On the other, all delegators totally order sequences of requests pre-ordered in a round-robin manner (logical pipelining), which further improves the throughput and balances the workload among data centers.

The contributions of this paper are as follows: (1) We design an H-RSM and its corresponding consensus protocol D-Paxos for replication among servers distributed across multiple data centers. We also theoretically prove that D-Paxos satisfies the safety and liveness properties of our H-RSM. (2) We develop an analytical model for D-Paxos, which provides a good approximation to the optimal batch sizes for pre-ordered requests when D-Paxos reaches its maximum throughput. (3) We validate the analytical model experimentally and present the experimental evaluation to show performance advantages compared to other Paxos variants for replication among servers distributed across multiple data centers.

Manuscript received January 20, 2016.
Manuscript publicized March 22, 2016.

[†]The authors are with the School of Computer Science and Engineering, South China University of Technology (SCUT), Higher Education Mega Centre, Panyu District, Guangzhou, 510006, China.
a) E-mail: yangyingyi10@126.com (Corresponding author)

DOI: 10.1587/transinf.2016EDP7036

The rest of paper is organized as follows. Section 2 presents related definitions and introduces our H-RSM. Section 3 reviews Paxos and Multi-Paxos. In Sect. 4, D-Paxos is detailed and related proofs are given. Section 5 presents our analytical model of D-Paxos for predicting how many requests must be pre-ordered to reach D-Paxos's maximum throughput. The experimental evaluation is given in Sect. 6 along with the validation of the analytical model. Section 7 presents related work; and Sect. 8 concludes the paper.

2. A Hierarchical Replication State Machine

2.1 System Model

We model a cloud-based system as $m(m \in \mathbb{N}, m \geq 3)$ data centers interconnected by a wide area network, each of which is composed of a set of servers and a set of clients. The wide-area network among data centers is modeled as a set of links pairwise and independent connecting data centers, and the local-area network among servers within the same data center is modeled as a set of links shared among them. The communication between data centers is usually latency-unbounded, while the communication within each data center is bandwidth-bound.

We assume that the replicas and clients in each data center run on separate servers and are connected through a LAN. Servers in each data center only receive requests from clients located in the same data center, and those clients in turn only connect and send the requests to the nearby servers in the same data center. This is reasonable, since clients always seek for lower-latency responses.

2.2 Definition

An H-RSM is a RSM that provides efficient use of resources and good overall performance. If the task for totally ordering all requests in a RSM can be partitioned into subtasks for partially ordering some requests in parallel and if the former is more costly than the latter, then a possible solution is to partition the task among all replicas (the servers) into parallel subtasks among some of replicas and then combine all resulting sequences partially ordered into a totally ordered one in some way. Note that, parts of replicas can implement a local RSM (by executing a local consensus protocol, or local protocol for short, to pre-order some requests and to form pre-ordered sequences of requests). Local RSMs can be viewed as logical entities and further implement a global RSM (by executing a global consensus protocol, or global protocol for short, which totally orders sequences of requests pre-ordered from local RSMs). In this way, when waiting for messages from other local RSMs during executing a global protocol, a local RSM can keep pre-ordering requests instead of idle waiting, which increases resource utilization.

On the basis of the above idea, all servers can be divided into $g(g \in \mathbb{N}, g \geq 3)$ components.

Definition 1. *A component is a set of servers which implement a local replicated state machine and a logical entity that involves in the implementation of a global replicated state machine. Let $\boldsymbol{S}$ be the complete set of all servers, a component $\boldsymbol{C}_i(\boldsymbol{C}_i \subseteq \boldsymbol{S}, i \in \{1, \ldots, g\})$ is a subset composed of $n_i(3 \leq n_i \leq |\boldsymbol{S}|)$ servers in $\boldsymbol{S}$.*

According to our system model and definition 1, all data centers can be considered as non-overlapping components. Let $\boldsymbol{D}_i = \boldsymbol{C}_i(|\boldsymbol{D}_i| \geq 3,\ i \in \{1, \ldots, m\})$ be a set of all servers within a data center, we have $\bigcup \boldsymbol{D}_i = \boldsymbol{S}, \boldsymbol{D}_i \cap \boldsymbol{D}_j = \emptyset, \sum(i = 1)^m = |\boldsymbol{D}_i| = |\boldsymbol{S}|(i \neq j, i, j \in \{1, \ldots, m\})$. Therefore, we refer to 'data center' as 'component' hereafter.

In an H-RSM, a global consensus protocol is required for local RSMs to implement a global RSM. For executing such a global protocol, there are two points to notice:

First of all, a local RSM is an agent performing some role in a global protocol, like the concept used in ordinary Paxos [5]. However, a global protocol is actually executed by some physical object(s) (e.g. servers or a component composed of multiple servers). Therefore, we have

Definition 2. *In an H-RSM, physical object(s) on behalf of a local RSM to participant a global protocol is (are) referred to as delegator(s).*

Secondly, instead of any single request from some client, pre-ordered sequences of requests generated in the execution of local protocol are proposed in global protocol. In this paper, a finite, non-empty and pre-ordered sequence of requests is denoted as $\langle r_1, \ldots, r_k \rangle$, with $r_i, i \in \{1, \ldots, k\}$ stands for the ith request issued by some client, where $k(k \in \mathbb{N})$ is the number of elements. An empty sequence is denoted as $\langle \rangle$. In this paper, we do not consider *commutable* requests and the history with commutable requests. Instead, we check for duplicate requests by recording committed requests.

To precisely specify actions performed by a component in an H-RSM, we introduce some related definitions as follows.

Definition 3. *A server is correct if it does not crash. All correct servers eventually agree on a unique request and this request must have been proposed. A server that is not correct is faulty.*

Definition 4. *A quorum refers to a set of participants (servers or components) that is large enough to ensure the liveness of a (local or global) consensus protocol.*

Definition 5. *A component is correct if there exist a quorum of servers within the component that are correct. Otherwise, it is faulty.*

Definition 6. *A component proposes a proposal, which means a delegator(s) elected on behalf of this component proposes a proposal to other components.*

Definition 7. *A component accepts a proposal, which* *means a delegator(s) acting on behalf of this component receives a proposal sent by another component and accepts it if it does not violate any of the safety properties of the global protocol.*

Definition 8. *Once a component learns a sequence agreed upon by the components, delegator(s) in this component then broadcasts the sequence to all servers within the same component, each of which eventually commit this sequence in the predetermined order.*

2.3 Properties

Consensus is a fundamental coordination problem that requires a group of agents to agree on a common value, based on values proposed. In our H-RSM, consensus is divided into local consensus and global consensus. Here, we refer to a server in a local consensus or a component in a global consensus as an *agent* and refer to a request in a local consensus or a sequence of requests in a global consensus as a *value*. Each agent in a consensus starts with an initial value to *propose* and *decides on* some proposed value. A local or global consensus implementation satisfies the following four properties.

Termination: Every correct agent eventually decides some value.

Validity: If all agents propose the same value v, then every correct agent decides v.

Integrity: Every correct agent decides at most one value.

Agreement: If a correct agent decides value v, then every correct agent decides v.

In an H-RSM, a server *submits* requests, a component *submits* pre-ordered sequences of requests, and each server eventually *commits* a totally ordered sequence of requests that are submitted by the servers. An H-RSM implementation should satisfy the liveness properties H-RSM Validity and H-RSM Agreement, and the safety properties H-RSM Integrity and H-RSM Total Order.

H-RSM Validity: If a correct server in a correct component submits a request r, then all correct servers in all components will eventually commit r.

H-RSM Agreement: If a correct server in a correct component commits a request r, then all correct servers in all correct components will eventually commit request r as well.

H-RSM Integrity: Any given request r is committed by each correct server in each correct component at most once, and only if r was previously submitted.

H-RSM Total Order: If two correct servers p and q from any correct component both commit request r_1 and r_2, then server p commits request r_1 before r_2 if and only if server q commits request r_1 before r_2.

3. Paxos and Multi-Paxos in a Nutshell

Paxos and Multi-Paxos are efficient consensus protocols commonly used for replicated state machines. Both of them are proved to satisfy the safety properties Validity, Integrity and Agreement and the liveness property Termination [5]. In this section, we briefly describe Paxos and Multi-Paxos.

Paxos requires $2f+1$ replicas to tolerate f faults. Paxos is usually described in terms of three roles: proposers that can propose values, acceptors that choose a single value and learners that learn what value has been decided. A single replica can execute multiple roles simultaneously. The execution of a Paxos instance proceeds in a sequence of *rounds*. For each round, one replica among the proposers plays the role of *coordinator* of the round. To propose a value, proposers send the value to the coordinator. The coordinator tries to get the others to agree on a value proposed by it. The round may succeed, in which case the value proposed is decided, or it may fail, in which case some other replica (or the same) starts a new round. Rounds are numbered with increasing numbers, with higher-numbered rounds superseding lower-numbered rounds.

Each round consists of two phases. In the first, the coordinator sends a PREPARE message to the acceptors (Phase 1a message) asking them to abandon all lower numbered rounds and to reply with the last value they accepted and the round number where they accepted it, or null if they did not accept any value (Phase 1b message). The acceptors answer to this message only if they have not participated in any higher numbered round. Once the coordinator receives a majority of replies, it enters the second phase. The coordinator chooses a value to propose based on the messages received from the acceptors. If some Phase 1b messages contain a value, the coordinator takes the value associated with the highest round number. Otherwise, it is free to choose any value. Once the value is chosen, the coordinator proposes this value with an ACCEPT message sent to all acceptors (Phase 2a message). The acceptors will once again answer only if they have not participated in a higher numbered round, in which case they send a Phase 2b message to the coordinator, informing it that they have accepted the proposal. Once receiving a quorum of Phase 2b messages from acceptors, the coordinator knows that a value has been decided and sends the decision to the learners.

Multi-Paxos is based on the observation that when executing a series of Paxos instances, the coordinator can execute Phase 1 for an arbitrary number of instances using a single prepare phase [6]. Afterwards, it only needs to execute Phase 2 of each instance, therefore reducing the number of communication delays for instances by almost half. In our implementation of D-Paxos, Multi-Paxos advances through a series of views, which play a similar role as rounds in a Paxos instance. A delegator elected in a component acts as the local coordinator. Once a new pre-ordering phase begins, the delegator becomes the local coordinator to coordinate a new view with a view number, which is higher than any view number previously observed by the coordinator. The coordinator then executes Phase 1 for the specified number of instances that, according to the local knowledge of the coordinator, were not yet decided. Unlike the PRE-

PARE message used in Paxos, the PREPARE message used in Multi-Paxos contains the view number and the list of k instances numbers with which instances do not know the decision. Once receiving such a message, an acceptor answers with a Phase 1b message for k instances of Paxos. For every instance, the Phase 1b message contains the last value it accepted and the corresponding view number, or null if it has not accepted any value for this instance. After completing the Phase 1, the local coordinator can then execute Phase 2 for k instances simultaneously. In this way, Multi-Paxos can actually improve the utilization of resources in a pipelining manner.

4. D-Paxos

In this section, we detail D-Paxos, a protocol which is designed to implement our H-RSM. Its core idea is to improve the utilization of idle time and available local-area bandwidth by ingeniously using batching and logical pipelining, which leads to better throughput.

4.1 Assumptions

To ensure the correct execution of D-Paxos, further assumptions are needed.

Benign failure We assume that all servers in the system may fail by crashing and can later recover their states through stable storage that survives crashes. Servers never exhibit Byzantine behaviors [7] even if they have failed, that is, all failures are *benign*.

Partial synchrony Servers communicate via exchanging messages in bidirectional channels. Message interactions are unreliable: messages can be lost or duplicated, but they are not corrupted. There is no upper bound on message transmission delays. There are bounds on relative process speeds and on message transmission times, but these bounds are not known and they hold only after some unknown but finite time.

The FLP impossibility result [8] asserts that it is not possible to solve the consensus problem in asynchronous systems when even a single process can fail. Like many other asynchronous consensus protocol, D-Paxos utilizes a failure detector to circumvent the impossibility result. To satisfy H-RSM Validity property (see the proof of Lemma 4 in Sect. 4.4 for more detail), D-Paxos requires a failure detector of eventually perfect class $\diamond\mathcal{P}$, which is allowed to be implemented under partial synchrony assumption. For brevity, we simply adopt two properties *completeness* and *accuracy*[†] introduced in [9] to give an informal definition of $\diamond\mathcal{P}$ as below.

Definition 9. *A failure detector of eventually perfect class $\diamond\mathcal{P}$ is a failure detector which satisfies* Strong Completeness *and* Eventual Strong Accuracy. *That is to say, a failure detector of class $\diamond\mathcal{P}$ guarantees that eventually every server that crashes is permanently suspected by every correct server and there is a time after which correct servers are not suspected by any correct server.*

[†]Completeness characterizes the failure detector's capability of suspecting incorrect processes, while accuracy characterizes the failure detector's capability of not suspecting correct processes.

4.2 The Execution of D-Paxos

To describe D-Paxos more precisely, we number all m components from 1 to m. As mentioned in Sect. 2.2, delegators are physical objects on behalf of a local RSM to participant a global protocol. Before the execution of D-Paxos, an initialization should has been completed within each component.

Initialization: A distinguished server is elected (through, e.g. a leader election) to be a delegator within each component.

These delegators elected play important roles in the execution of D-Paxos. The logical relationships among components and servers are shown in Fig. 1.

Intuitively, D-Paxos consists of two phases, local preordering phase (also known as pre-ordering phase) and global total ordering phase (also known as total ordering phase), which are used for implementing a local RSM and a global RSM respectively. Although their relationships appear to be intuitive, all participants in D-Paxos are required to follow some rules, which are critical for D-Paxos to satisfy the safety and liveness properties of an H-RSM.

Rule 1. Each delegator elected from each component has three roles. First, it acts as a local coordinator, which executes a specified number (say k) of instances of local protocol (also referred to as *local instances*) in each preordering phase. Second, the elected delegator acts as a global acceptor, which participates instances of global protocol (also referred to as *global instances*) in total ordering phases. Last, it acts as a global leader in a round-robin manner, which coordinates global instances among delegators in total ordering phases. A bounded number of global instances $p \times m + i$ is assigned to delegator d_i, where $p \in \mathbb{N}$ and $d_i \in \boldsymbol{D}_i, i \in \{1, \ldots, m\}$. Once a delegator becomes the global leader of global instances assigned to it, all other delegators become global acceptors in those global instances.

Rule 2. Once a delegator has become the global leader and initiated a global instance, all delegators agree that the leader is the default one for this instance and start from the state in which the leader had run Phase 1 (just like the one

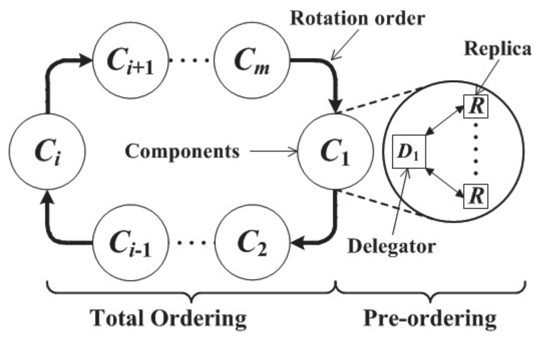


Fig. 1 Logical relationships among components and servers.

in ordinary Paxos) for some initial round r; that is, they promise not to accept any proposal for any round smaller than r.

Rule 3. Two types of proposals are defined for global instances. One is the finite, non-empty and pre-ordered sequence of requests $\langle r_1, \ldots, r_k \rangle$ and the other is the empty sequence $\langle\rangle$ which means a proposal that leaves the state unchanged and that generates no response. Only the default delegator who is assigned to a global instance can propose a pre-ordered sequence of requests or an empty sequence $\langle\rangle$ (a global instance can be skipped by its default delegator with an empty sequence $\langle\rangle$ if there is not any request pre-ordered and buffered) in that global instance, while others can only propose an empty sequence $\langle\rangle$.

Based on the above rules, the execution of each D-Paxos instance proceeds in two phases:

Pre-ordering phase within each component: (1) Each delegator, working as a local coordinator, continuously receives requests from clients. (2) Each delegator executes multiple local instances (here we use Multi-Paxos [5], which means it simultaneously coordinates k instances of Phase 2 of the ordinary Paxos). The resulting pre-ordered sequence is written to a local stable storage and used as a proposal for the next global instance that the delegator will coordinate.

Total ordering phase among components: (1) By Rule 1, the delegator turns to be the global leader. (2) By Rules 2 and 3, the global leader proposes a proposal by sending a GLOBAL_ACCEPT message with the pre-ordered sequence of requests obtained in its pre-ordering phase to all other components and waiting for GLOBAL_ACCEPTED messages from a quorum of global acceptors. (3) Once receiving a proposal from the global leader, a global acceptor records the sequence into the stable storage and checks whether the following constraints are satisfied: this component has not yet accepted any proposal with greater round number for this instance and the delegator who forwarded the proposal with a LOCAL_FORWARD message to all servers within this component has received LOCAL_REPLY messages from a quorum of servers. If these constraints are satisfied, the global acceptor sends a GLOBAL_ACCEPTED message back to the global leader; otherwise, it does nothing. (4) Once receiving GLOBAL_ACCEPTED messages from a quorum of global acceptors, the global leader learns that the proposal has been chosen and notifies all other components by broadcasting a GLOBAL_LEARN message. All delegators which learn the result notify all other servers in their component by broadcasting a local LOCAL_LEARN message.

From rules 2, 3 and the description of a total ordering phase, the global protocol is actually the Phase 2 of ordinary Paxos in which its proposals and actions are restricted. That is to say, a global instance is a special case of an instance of Phase 2 of ordinary Paxos. Therefore, a global instance has the same quorum size as an ordinary Paxos instance, namely, a majority ($\lfloor \frac{m}{2} \rfloor + 1$) of delegators.

4.3 Failure Detection and Failover

In this section, we consider how to detect and handle failures happened in both two phases of D-Paxos.

In D-Paxos, the failure detection relies on a failure detector of class $\diamond\mathcal{P}$, which is strictly stronger than the weakest failure detector Ω [10]. Informally, it guarantees that all faulty servers and eventually only faulty servers are suspected. As described earlier, the execution of a D-Paxos instance proceeds in two phases, i.e. the pre-ordering phase and the total ordering phase. The pre-ordering phase is executed before the total ordering phase. Since there are differences between these two phases in terms of execution conditions (mainly the message latency), it would not be appropriate to implement a failure detector of class $\diamond\mathcal{P}$ among servers distributed across multiple components. Actually, two failure detectors for participants involved in these two phases are implemented by a similar design idea but with slight differences and set with different parameters due to their execution conditions. As a consequence, failure detectors of eventually perfect class $\diamond\mathcal{P}$ (which is required for D-Paxos to satisfy H-RSM Validity, see Sect. 4.4 for more detail) are used in both phases, even though the weakest failure detector Ω is enough for the pre-ordering phase.

We take the implementation for delegators in total ordering phase as an example to briefly describe the algorithm we use to implement a failure detector of class $\diamond\mathcal{P}$. In this case, a faulty delegator in effect means a faulty component. This is because a faulty delegator can be replaced through a re-election mechanism for its component, which is discussed later. A component fails in case more than a majority of servers in this component fail. A faulty component should be detected with a failure detector. However, a failure detector can only be implemented among delegators, since they participate in global instances on behalf of components. For simplicity, we implement a failure detector of class $\diamond\mathcal{P}$ based on the heartbeat, ring-based algorithm presented in [11]. In this algorithm, each delegator d_i periodically sends a heartbeat message to its successor in the ring. Every delegator also waits for periodical heartbeats from its predecessor in the ring. If delegator d_i does not receive such a heartbeat on a specific time-out interval, it suspects that its predecessor has crashed, include this predecessor in its local list of suspected delegators L_{d_i}, and set its predecessor to it current predecessor's predecessor. If delegator d_i later on receives a heartbeat message from a delegator d_j it is erroneously suspecting, delegator d_i correct its local list of suspected delegators L_{d_i}, increments the time-out interval, and changes the perception of its correct predecessor in the ring to d_j. Besides its local list of suspected delegators L_{d_i}, every delegators d_i has a global list G_{d_i}, which provides the properties of eventually perfect class $\diamond\mathcal{P}$. Delegators propagate the global lists around the ring. Whenever d_i includes a delegator d_j in its L_{d_i}, it also includes d_j in G_{d_i}. Finally, each time d_i receives a heartbeat message from its supposed correct predecessor in the ring, it builds a new global list of suspected delegators by merging the global list carried by the heartbeat with its own local list L_{d_i}. Delegator d_i also sets its supposed correct successor in the ring to its nearest delegator following the ring but not belonging to its new global list.

With a failure detector of eventually perfect class $\diamond\mathcal{P}$, a delegator re-election mechanism can be implemented in a component. When a delegator fails or is falsely suspected during the pre-ordering phase, such a delegator re-election mechanism prevents the liveness of local instances from being affected. In total ordering phase, a delegator re-election mechanism can also provide fault tolerance for components. Therefore, a delegator re-election mechanism is very important for D-Paxos. In D-Paxos, using the algorithm for eventually perfect failure detector class $\diamond\mathcal{P}$ described above makes the implementation of a delegator re-election mechanism more easier. Specifically, all servers in a component $\boldsymbol{C}_i$ are arranged in a logical ring. With the implementation of a failure detector of class $\diamond\mathcal{P}$, each correct server can receive the set of suspected servers returned by the failure detector. Consequently, it can also get the set of non-suspected servers, which is just the complement of the set of suspected servers. On this basis, each server applies a same deterministic function `delegator` to the set of non-suspected servers to choose one from it, eventually all correct servers will permanently agree on the same correct server. Since all servers in a component are arranged in a logical ring, function `delegator` can simply choose the server with the lowest numbered to be the distinguished one. Because delegators participate in global instances, they should be known to each other. In our implementation, each server maintains a local list which records the related information about delegators of components. Whenever a server is elected to be a delegator, it is responsible for updating the list and informing all other servers of the update.

With the implementation of the failure detector of class $\diamond\mathcal{P}$ and the delegator re-election mechanism, we specify the following failover mechanism for faulty delegators.

Failover mechanism 1. If a delegator fails or is falsely suspected, a new delegator is re-elected from the same component where the previous one locates and it restores the state by inquiring for other servers about the state and receiving responses from them.

A component can handle a delegator failure by failover mechanism 1. However, a component cannot commit requests in case more than a quorum of servers within this component failed, e.g. due to power outage, or falsely suspected, e.g. due to a temporary network breakdown. Once a component failure did happen, other delegators cannot commit requests even if instances which they coordinate to choose those requests succeeded. The protocol is blocked. As a consequence, we provide a solution as follows.

Failover mechanism 2. If a component fails, another component is permitted to finish those global instances which are assigned to the failed component. The delegator from that alternative component acts as the alternative leader and requires responses from other delegators by sending a GLOBAL_PREPARE message. If it receives GLOBAL_PROMISE messages from a quorum of delegators and gets a non-empty, pre-ordered sequence of requests, it proposes a proposal with this sequence. Otherwise, it proposes a proposal with empty sequence $\langle\rangle$ by Rule 3.

Based on failover mechanism 2, in a global instance i, only the non-empty, pre-ordered sequence of requests proposed by the default delegator can be chosen, any other non-empty, pre-ordered sequence cannot be proposed and chosen. In other words, the decision for a global instance i is either a non-empty, pre-ordered sequence of requests proposed by the default delegator or an empty sequence $\langle\rangle$.

Besides the cases in which a component failure did happen, the cases in which a component is falsely suspected should also be handled. Accordingly, we specify another failover mechanism as follows.

Failover mechanism 3. In case a component is falsely suspected, if it proposes a pre-ordered sequence $h \neq \langle\rangle$ to global instance i and learns that $\langle\rangle$ is chosen, then it is permitted to propose h again.

4.4 Proof

We show that D-Paxos with above failover mechanisms implements consensus and satisfies the requirements of an H-RSM.

Theorem 1. *Assuming a failure detector of eventually perfect class $\diamond\mathcal{P}$, D-Paxos implements consensus.*

Proof. In the pre-ordering phase, the safety properties of local protocol Multi-Paxos obviously can be guaranteed. With a failure detector of class $\diamond\mathcal{P}$, a failure detector stronger than Ω, the failed delegator will be detected and a new delegator will be elected by failover mechanism 1 and takes over to coordinate local instances for its component, once the previous delegator has failed. So the liveness properties of the local protocol Multi-Paxos are not affected. This phase does not involve committing a request. Therefore, the safety and liveness properties of the global protocol are not affected either.

In the total ordering phase, once a delegator takes turns to be a global leader, it proposes the pre-ordered sequence achieved in its pre-ordering phase. The global protocol starts from a specific (and safe) state. In case the delegator fails, with failure detector $\diamond\mathcal{P}$ and failover mechanism 1, the global protocol still makes progress. In the worst case, when the component fails, with failure detector $\diamond\mathcal{P}$ and failover mechanisms 2 and 3, the safety and liveness properties of the global protocol are not affected. Since the global protocol is actually the Phase 2 of ordinary Paxos with its proposals and actions restricted, it obviously implements consensus. Therefore, D-Paxos implements consensus and satisfies the following properties: Termination, Validity, Integrity and Agreement. □

Lemma 1. *D-Paxos satisfies H-RSM Agreement.*

Proof. If a request r is committed by a correct server p, then (1) component $\boldsymbol{C}$ where p locates must commit a pre-ordered sequence h of requests containing r; (2) the pre-ordered sequence h must have been decided by $\boldsymbol{C}$ in some global instance i; (3) component $\boldsymbol{C}$ must have learned all global instances smaller than i; and (4) component $\boldsymbol{C}$ does not learn h in any global instance smaller than i. For any given correct component $\boldsymbol{Q}$, the Termination property of consensus guarantees that $\boldsymbol{Q}$ will eventually learn all instances smaller than or equal to i as well. By the Agreement property of consensus, the pre-ordered sequence h is learned in global instance i and h has never been learned in instances smaller than i. Therefore, once all instances smaller than or equal to i are learned and committed by component $\boldsymbol{Q}$, all servers in $\boldsymbol{Q}$ will commit pre-ordered sequence h containing r, which happens eventually. □

Lemma 2. *D-Paxos satisfies H-RSM Integrity.*

Proof. As mentioned in Sect. 2.2, a committed request is recorded and then used for duplicates checking. If a request r is committed by a correct server p, then a pre-ordered sequence h containing r is committed by component $\boldsymbol{C}$ containing p. The pre-ordered sequence h of requests must have been chosen in some global instance i. The Validity property of consensus guarantees that h was proposed by some component $\boldsymbol{Q}$ in instance i. Since request r is an element of h, request r must have been proposed by the delegator from component $\boldsymbol{Q}$ in some local instance. □

Lemma 3. *D-Paxos satisfies H-RSM Total Order.*

Proof. Given that two correct servers p and q from two correct components $\boldsymbol{C}_p$ and $\boldsymbol{C}_q$ respectively, both of which commit request r_1 and r_2, we show that the lemma holds in two cases.

If request r_1 and r_2 belong to the same pre-ordered sequence h of requests, then component $\boldsymbol{C}_p$ and $\boldsymbol{C}_q$ must commit requests in the same order as they appear in the sequence. Therefore, the lemma holds in this case.

If request r_1 and r_2 respectively belong to different pre-ordered sequences, assuming request r_1 belongs to pre-ordered sequence h_1 and request r_2 belongs to pre-ordered sequence h_2. If component $\boldsymbol{C}_p$ commits r_1 before r_2, that is, component $\boldsymbol{C}_p$ commits h_1 before h_2, then there must exist global instances i_1 and $i_2 (i_1 < i_2)$ such that (1) component $\boldsymbol{C}_p$ has learned all instances smaller than or equal to i_2; (2) component $\boldsymbol{C}_p$ learns h_1 in i_1 and does not learn h_1 in instances smaller than i_1; and (3) component $\boldsymbol{C}_p$ learns h_2 in i_2 and does not learn h_2 in instances smaller than i_2. If component $\boldsymbol{C}_q$ commits r_2 before r_1, which means $\boldsymbol{C}_q$ commit h_2 before h_1, then there must exist global instances j_1 and $j_2 (j_1 < j_2)$ such that (1) component $\boldsymbol{C}_q$ has learned all instances smaller than or equal to j_2; (2) component $\boldsymbol{C}_q$ learns h_2 in j_1 and does not learn h_2 in instances smaller than j_1; and (3) component $\boldsymbol{C}_q$ learns h_1 in j_2 and does not learn h_1 in instance smaller than j_2. Without loss of generality, we assume $i_2 \leq j_2$. Since component $\boldsymbol{C}_p$ learns h_1 in i_1, by the Agreement property of consensus, component $\boldsymbol{C}_q$ must also learn h_1 in i_1. Since $i_1 < i_2 \leq j_2$, this contradicts with $\boldsymbol{C}_q$ does not learn h_1 in any instance smaller than j_2. Therefore, the lemma holds in this case. □

Lemma 4. *Assuming a failure detector of eventually perfect class $\diamond\mathcal{P}$, D-Paxos satisfies H-RSM Validity.*

Proof. Given that a correct server p from correct component $\boldsymbol{C}_p$ submits a request r, the failure detector $\diamond\mathcal{P}$ guarantees that request r will eventually be placed in certain position in a pre-ordered sequence h of requests, then pre-ordered sequence h will be proposed to a global instance i. After that, (1) If component $\boldsymbol{C}_p$ are not falsely suspected, all correct components will eventually propose sequences to global instances smaller than i, and all non-faulty components will eventually learn sequences decided for those instances. For those global instances which are assigned to faulty components, they will be handled by failover mechanism 2, and all non-faulty components will eventually learn sequences decided. So request r will eventually be decided in i. (2) If component $\boldsymbol{C}_p$ is falsely suspected, e.g. transient network partition, the failure detector $\diamond\mathcal{P}$ guarantees that eventually all faulty servers and only faulty servers are suspected. Therefore, component $\boldsymbol{C}_p$ will recover, say, at time t. By failover mechanism 3, component $\boldsymbol{C}_p$ will continue to submit the pre-ordered sequence h upon false suspicion until h is chosen. If the pre-ordered sequence h hasn't been chosen by t, by case (1), the pre-ordered sequence h will be chosen once component $\boldsymbol{C}_p$ re-proposes h after t. Therefore, D-Paxos satisfies H-RSM Validity. □

Theorem 2. *Assuming a failure detector of eventually perfect class $\diamond\mathcal{P}$, D-Paxos implements hierarchical replicated state machines.*

Proof. From Lemma 1 to Lemma 4. □

D-Paxos does not necessarily guarantee the sequential consistency. As one of consistency models used in the domain of concurrent computing, sequential consistency requires that the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [12]. Using D-Paxos, the execution order of requests at all correct server is not necessarily the same as the order in which those requests are issued. D-Paxos only guarantees that all requests ordered are executed by every correct server in the same order, as proved earlier. In terms of correctness, one-copy serializability used in database protocols adapts to replicated scenarios. The basis for correctness is data dependencies. Sequential consistency allows to read old values under some conditions. In this respect, one-copy serializability has similarities with sequential consistency, but strictly speaking, the two consistency criteria are different [13]. In some sense, D-Paxos provides consistency guarantees similar to one-copy serializability.

5. The Size of Pre-Ordered Batch of Requests

Both batching and pipelining are effective optimizations for improving system's performance by making full use of the resources. When replicating data across multiple data centers in the cloud, a replicated system's throughput will be improved as long as idle resources in data centers are effectively used. D-Paxos achieves this goal with batching (by utilizing LAN bandwidth available to coordinate local instances in pre-ordering phases and to form pre-ordered batches of requests that are used for global instances) and logical pipelining (by coordinating multiple global instances in a round-robin manner and amortizing workload over delegators).

In D-Paxos, although a delegator coordinates not only global instances but also local instances, the workload for coordination is shared among delegators through rotating leader scheme. Due to high wide-area latency, a delegator actually coordinates several local instances to form its pre-ordered batches of requests as long as it is idle waiting for messages for global instances. Therefore, delegators' workloads are dominated by local instances. For replication among servers distributed multiple components, D-Paxos may reach its maximum throughput due to high wide-area latency before delegators saturate other resources (e.g. CPU or bandwidth), which is demonstrated by our measurement results for a particular setting shown in Sect. 6. Therefore, it is critical to coordinate a moderate number of local instances to form a large enough pre-ordered batch of requests in each pre-ordering phase. In this section, in the case of latency bounded, we only consider how large a pre-ordered batch of requests should be formed (i.e. how many local instances should be coordinated) by a delegator in order to gain the potentially maximum throughput.

5.1 Parameters

For simplicity, we only concern about the settings where all components have the same number of servers and focus on the best case, that is, all delegators execute local instances and global instances at approximately the same rate, in the case of no message loss and failures. We assume that there is no other application competing for bandwidth and CPU time and mechanisms internal to the protocol, such as delegator re-election, should have a minimal impact on D-Paxos's performance. In the system we concern, the wide-area network latency is the main potential bottleneck which affects D-Paxos's throughput. As mentioned earlier, the local protocol is Multi-Paxos whereas the global protocol is actually the Phase 2 of ordinary Paxos with its proposals and actions restricted, therefore, both phases in D-Paxos have similar message pattern to that of Phase 2 of ordinary Paxos.

To better understand the relationship among D-Paxos's performance and related properties (e.g. latency, request sizes, the number of components and replicas) and how batch sizes impact D-Paxos's throughput, we develop an analytical model. The parameters used in the rest of this paper are shown in Table 1. Note that, all payloads or data used in D-Paxos are capsulated in protocol messages. Since the protocol header is relatively much smaller than the payload, we simply use the payload to indicate the size of a protocol message. Taking a LOCAL_ACCEPT message used in pre-ordering phase for example, we have $S_{local\_accept} = S_{req} + h$, where h is the size of the protocol headers. Since $h \ll S_{req}$, we assume that $S_{local\_accept} \approx S_{req}$ for simplicity and readability. Besides, since sizes S_{local_accept}, S_{local_reply} and $S_{local\_confirm}$ of acknowledge messages which are propagated within components are similar, we just use S_{ack} to denote their sizes.

Table 1 Parameters.

Parameter	Description
m	Number of components
n	Number of replicas per component
L_w	One-way delay between components
L_l	One-way delay between replicas in each component
B_w	Wide-area network bandwidth available
B_l	Local-area network bandwidth available
k	Size of a pre-ordered batch (sequence) of requests
S_{global_accept}	Size of GLOBAL_ACCEPT message in total ordering phase
$S_{global\_accepted}$	Size of GLOBAL_ACCEPTED message in total ordering phase
S_{global_learn}	Size of GLOBAL_LEARN message in total ordering phase
S_{local_accept}	Size of LOCAL_ACCEPT message in pre-ordering phase
$S_{local\_forward}$	Size of LOCAL_FORWARD message used for forwarding a global proposal
S_{local_learn}	Size of LOCAL_LEARN message used for forwarding global decision
S_{local_ack}	Size of LOCAL_ACCEPTED, LOCAL_REPLY and LOCAL_CONFIRM message
S_{req}	Size of a request

5.2 Quantitative Analytical Model

To get the optimal number k of batch size, we consider how many requests can a delegator orders (i.e. how many Phase 2 instances of ordinary Paxos can be executed in parallel when executing Multi-Paxos) in its interval between the time when it turns to be a global leader in case of stable network and components with equal processing capacity. Let the time spent by a global leader in sending out a proposal (i.e. a pre-ordered batch of requests) in the total ordering phase of a D-Paxos instance be T^{WAN}_{maj}, the time spent by a global acceptor to receive a proposal be $T^{WAN}_{receive}$ and the one-way latency from a global leader to a global acceptor be L_w. A delegator can enter into pre-ordering phase of the next D-Paxos instance assigned to it and act as a local coordinator once it has proposed its global proposal in total ordering phase of the current D-Paxos instance. A potential leader can end its current pre-ordering phase of the current D-Paxos instance and start its next total ordering phase once it receives its predecessor's proposal. There are m rotations in a rotating cycle. Therefore, the rotating cycle of this potential leader can be represented as $(m-1)\times(T^{WAN}_{receive}+T^{WAN}_{maj})+m\times L_w$. We notice that, during this period of time, if the delegator (i.e. this potential leader) can receive enough requests, it can stay busy and takes full advantage of resources to prepare a large enough batch for its next global instance. However, besides working as a local coordinator to coordinate local instances in pre-ordering phase of current D-Paxos instance (the time spent is denoted as T^{LAN}_{inst}) and send out the resulting sequence as a proposal in subsequent total ordering phase (the time spent is T^{WAN}_{maj}), the delegator also need to act as a global acceptor to accept proposals from other global leaders in global instances assigned to other delegators (the time spent is denoted as T^{WAN}_{accept}) and forward results learned to all other replicas within its component (the time spent is denoted as T^{WAN}_{decide}). The message pattern during this interval is shown in Fig. 2.

From the description above and Fig. 2, we get

$$\begin{aligned}(m-1)&\times(T^{WAN}_{receive}+T^{WAN}_{maj})+m\times L_w\\&=k\times T^{LAN}_{inst}+T^{WAN}_{maj}+(m-1)\times T^{WAN}_{accept}+m\times T^{WAN}_{decide}\end{aligned}\tag{1}$$

Here we consider T^{WAN}_{maj}, $T^{WAN}_{receive}$, T^{LAN}_{inst}, T^{WAN}_{accept} and T^{WAN}_{decide} respectively.

Once a delegator becomes the next global leader, it proposes its batch (k requests) as a proposal. Wide-area networks latency $L_w(L_w \gg L_l)$ is the main bottleneck. The inter-component section of the connection between the delegators will likely be different for each pair of delegators, so that after leaving the component, the messages from a delegator will follow independent paths to other delegators. Every message sent by the delegator uses a separate logical channel of bandwidth B_w in this case. With proposals of the same size, we can obtain the time T^{WAN}_{maj} required for a global leader to propose a global proposal with GLOBAL_ACCEPT message in total ordering phase and the time $T^{WAN}_{receive}$ required for a global acceptor to receive such a global proposal is

$$T^{WAN}_{maj}=T^{WAN}_{receive}=\frac{S_{global\_accept}}{B_w}=\frac{k\times S_{req}}{B_w}\tag{2}$$

When a delegator acts as a local coordinator, it coordinates Multi-Paxos instances in a LAN. In this case, the delegator has a total bandwidth of B_l to share among all other replicas in the same component [14]. As mentioned above, Phase 2 instances of ordinary Paxos are the dominant workload for Multi-Paxos instances. With the assumption that there is no delay from the time an acceptor receives a message until it answers, the time for one Phase 2 instance is the time from the sending of the first LOCAL_ACCEPT message to the reception of enough LOCAL_ACCEPTED messages. To finish an instance, the local coordinator must receive a majority of LOCAL_ACCEPTED messages. This happens only if the LOCAL_ACCEPT message was received by a majority of local acceptors. Since the local coordinator in Multi-Paxos coordinates multiple Phase 2 instances simultaneously and all instances have the same message pattern (see Fig. 3 (a)), it behaves similarly to the execution of instances with pipelining. In this scenario, the only exception is that the local coordinator caches all pre-ordered requests to a queue so as to form a batch rather than informing the client of the result. According to Fig. 3 (a) and analysis above, we have the time $k\times T^{LAN}_{inst}$ required for a local coordinator to per-order k local instances in pre-ordering phase is

$$\begin{aligned}k\times T^{LAN}_{inst}&=k\times\left\lfloor\frac{n}{2}\right\rfloor\times\frac{S_{local\_accept}}{B_l}+2\times L_l+\frac{S_{local\_ack}}{B_l}\\&=\left\lfloor\frac{n}{2}\right\rfloor\times\frac{k\times S_{req}}{B_l}+2\times L_l+\frac{S_{local\_ack}}{B_l}\end{aligned}\tag{3}$$

After receiving a global proposal from a global leader, a global acceptor will determine on whether it accepts the proposal or not. As mentioned in Sect. 4.2, the decision is made according to replies from a quorum of repli-

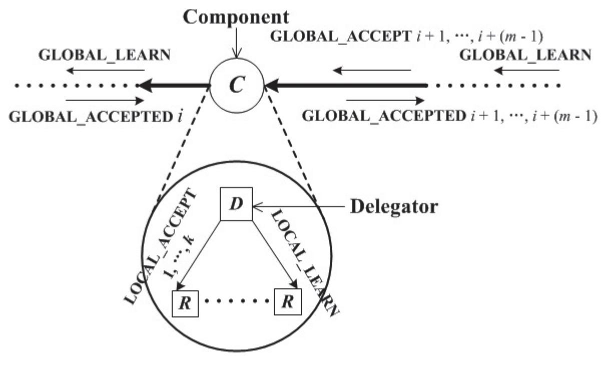


Fig. 2 Message pattern when a delegator acts as a local coordinator, a global leader or a global acceptor.

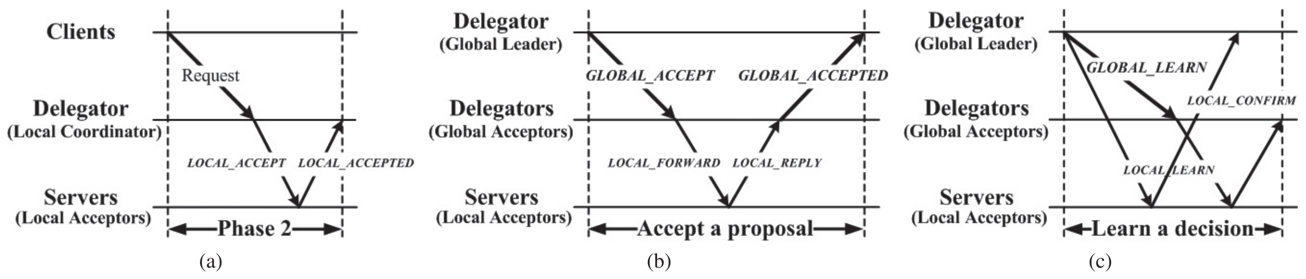

Clients
Request
Delegator (Local Coordinator)
LOCAL_ACCEPT
LOCAL_ACCEPTED
Servers (Local Acceptors)
Phase 2
(a)
Delegator (Global Leader)
GLOBAL_ACCEPT
GLOBAL_ACCEPTED
Delegators (Global Acceptors)
LOCAL_FORWARD
LOCAL_REPLY
Servers (Local Acceptors)
Accept a proposal
(b)
Delegator (Global Leader)
GLOBAL_LEARN
LOCAL_CONFIRM
Delegators (Global Acceptors)
LOCAL_LEARN
Servers (Local Acceptors)
Learn a decision
(c)

Fig. 3 Message pattern when a delegator (a) as a local coordinator coordinates an instance, (b) as a global acceptor accepts a global proposal, and (c) as a global acceptor learns the decision.

cas within its component. In this way, a new delegator can restore the state by inquiring for other replicas about the state in case the previous delegator fails. Therefore, the time T^{WAN}_{accept} required for a global acceptor to accept a global proposal includes the time used by a global acceptor for receiving the GLOBAL_ACCEPT message and sending the GLOBAL_ACCEPTED message back and the time used by a global acceptor for broadcasting the proposal with LOCAL_FORWARD messages and collecting LOCAL_REPLY messages from a majority of replicas within its component (see Eq. (4)). The message pattern is shown in Fig. 3 (b). The time T^{WAN}_{accept} required for a global acceptor to accept a global proposal in total ordering phase is given in Eq. (4).

$$\begin{aligned} T^{WAN}_{accept} &= \frac{S_{global\_accept}}{B_w} + \left\lfloor \frac{n}{2} \right\rfloor \times \frac{S_{local\_forward}}{B_l} + 2 \times L_l \\ &\quad + \frac{S_{local\_ack}}{B_l} + \frac{S_{global\_accepted}}{B_w} \\ &= \frac{k \times S_{req}}{B_w} + \left\lfloor \frac{n}{2} \right\rfloor \times \frac{k \times S_{req}}{B_l} \\ &\quad + 2 \times L_l + \frac{S_{local\_ack}}{B_l} + \frac{S_{global\_accepted}}{B_w} \end{aligned} \tag{4}$$

Once receiving GLOBAL_ACCEPTED messages from a quorum of global acceptors, the global leader learns that its proposal for a D-Paxos instance has been chosen. Besides broadcasting a GLOBAL_LEARN message to other delegators, it also broadcast a LOCAL_LEARN message to inform all replicas within its component of the decision. After receiving a global GLOBAL_LEARN message from a global leader, a global acceptor learns the decision and simply forward the decision with a LOCAL_LEARN message to all other replicas within its component. All replicas which receive a LOCAL_LEARN message will acknowledge it with a LOCAL_CONFIRM message. The message pattern is shown in Fig. 3 (c). Regardless of its role, a delegator should handle the same number of messages in both cases. Therefore, the time T^{LAN}_{decide} required for a global leader or an acceptor to learn a decision of a D-Paxos instance is given by Eq. (5).

$$\begin{aligned} T^{WAN}_{decide} &= \frac{S_{global\_learn}}{B_w} + \left\lfloor \frac{n}{2} \right\rfloor \times \frac{S_{local\_learn}}{B_l} + 2 \times L_l \\ &\quad + \frac{S_{local\_ack}}{B_l} \\ &= \frac{k \times S_{req}}{B_w} + \left\lfloor \frac{n}{2} \right\rfloor \times \frac{k \times S_{req}}{B_l} + 2 \times L_l \\ &\quad + \frac{S_{local\_ack}}{B_l} \end{aligned} \tag{5}$$

Introducing Eqs. (2), (3), (4), (5) into Eq. (1), we obtain the calculation formula for the optimal number k of batch size. Since sizes of acknowledge messages S_{ack} and $S_{global\_accepted}$ are much smaller compared to sizes of messages carrying request load, such as S_{global_accept}, they would be negligible. By simplifying the resulting formula, we present the approximate formula as below in Eq. (6).

$$k \approx \left\lfloor \frac{L_w - 4 \times L_l}{(\frac{n}{B_l} + \frac{2}{m \times B_w}) \times S_{req}} \right\rfloor \tag{6}$$

6. Experimental Study

In this section, we study the performance of D-Paxos experimentally and validate our analytical model obtained. To facilitate performance evaluation, we put enough pressure on these protocols, which is achieved by providing adequate client requests and constructing a simple service adopting these protocols. The simple service receives requests and simply replies, but without state changed.

In order to study D-Paxos's performance improvement for replication among servers distributed across multiple components, we compare it with Multi-Paxos and Mencius [15], an efficient consensus protocol for WANs, in terms of throughput, latency, scalability and fault tolerance in the same settings. Note that neither Multi-Paxos nor Mencius is designed to implement a H-RSM. When running among servers distributed across multiple data centers, there is no concept of 'component' for both of them. However, for comparison's sake, we still refer to 'data center' as 'component' in the discussion involving Multi-Paxos and Mencius.

6.1 Experimental Settings

We run our experiments over Emulab [16]. Emulab provides an experimentation facility that integrates different experimental environments: emulation, simulation and live-Internet into a common framework. The integration brings the control and ease of use usually associated with simulation to emulation and live network experimentation without sacrificing realism.

To run our experiments over Emulab, we specified a network topology desired via an *ns* script first. The virtual topology includes links and LANs, with associated characteristics such as bandwidth, latency. The *ns* specification is then parsed into an intermediate representation that is stored in a database and later allocated and loaded onto hardware. Mechanisms such as interposing Dummynet nodes can be used to regulate bandwidth, latency in our experiments. During experiment execution, Emulab provides interfaces and tools for experiment control and interaction.

Appropriate settings are emulated for specific experimental purposes. Due to limited conditions, up to five components are emulated and each of them is composed of 10 to 15 replicas. Nevertheless, we believe such settings are sufficient for evaluating our D-Paxos. An extra server in each component is used to simulate multiple clients sending requests to this component. All servers within the same component are connected by an emulated local-area network with 0.25 ms delay and 920 Mb/s of effective bandwidth while all components are connected by an emulated wide-area network with 150 ms delay and 9.45 Mb/s of effective bandwidth. In each experiment, the time-out interval $\triangle s$ and $\triangle d$ for failure detectors implemented separately for servers in a component and for delegators among components are set to 2 ms and 900 ms respectively.

In a setting, where five components are built and each of them is composed of 10 replicas, the optimal size k for pre-ordered batches is about 2,200 with request size of 256 B (the minimum size used in our experiments), which is far less than the limits of the delegator's ability to process 256 B requests (the test under the same setting indicates that a delegator is capability of processing more than 5,000 requests/second). Therefore, CPU is not the main bottleneck in our experiments.

6.2 Throughput and Latency

In this section, we study D-Paxos's improvement in throughput and latency compared to Multi-Paxos and Mencius. Experiments with request sizes of 4 KB, 1 KB and 256 B were run on three components, each of which has 10 replicas.

As shown in Fig. 4 (a), no matter what request size adopted, D-Paxos outperforms the others with respect to the throughput. With 256 B requests and batch size of 930, the peak throughput of D-Paxos reaches around 2,300 requests/second, which is about 3.5 times and 1.25 times higher than Multi-Paxos (660 requests/second) and Mencius (1,830 requests/second) respectively. Multi-Paxos has the lowest throughput. Although many instances can be executed simultaneously in Multi-Paxos, the unique leader is the bottleneck, which leads to unbalanced link dependency. By making all replicas sharing coordination workload of instances, Mencius presents higher throughput. However, more coordinators mean more messages transmitted simultaneously in the whole system, which restricts Mencius to achieve a better possible performance. In contrast, only delegators in D-Paxos involve in both global instances and local instances, which decreases the number of messages and the amount of data transmitted over networks and reduces the potential uncertainty about the time spent in making a decision. Delegators efficiently exploit idle time left by inherent higher latency among data centers to pre-order requests and form batches, which improves the throughput through batching in total ordering phases. Moreover, the rotating leader scheme, a logical pipelining manner, is used to further improve D-Paxos's throughput.

Figure 4 (b) shows the average latency of Multi-Paxos, Mencius and D-Paxos for ordering 1,000 requests respectively. Since the leader is unique in Multi-Paxos, client requests from other components should be sent to this unique leader over wide-area networks, which results in an additional round trip delay for committing those requests. Therefore, Multi-Paxos has the highest latency. In Mencius, every replica is given a chance to coordinate instances as a leader. Client requests can be handled by replicas locally, rather than being forwarded to a unique leader which may far away from them. Therefore, Mencius has lower average latency. In D-Paxos, requests from each component can be coordinated directly by the corresponding delegator due to rotating leader scheme, rather than being sent to a remote leader. Moreover, each pre-ordering phase is executed by a delegator when total ordering phases are executed, there is no extra latency caused. In contrast, batches formed in pre-ordering phases can be used for improving D-Paxos's throughput and decreasing latency. Therefore, we can see that D-Paxos has the lowest latency.

6.3 Scalability

High scalability is one of pursuing goals to cloud storage systems. Systems' fault tolerance is expected to be improved by scaling data centers or replicas without significant impact on performance. We evaluated the scalabil-

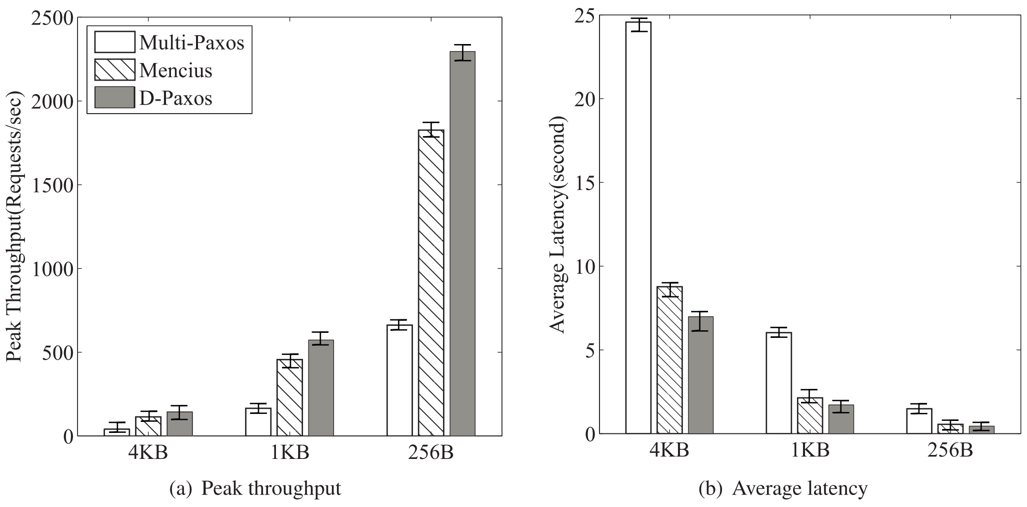
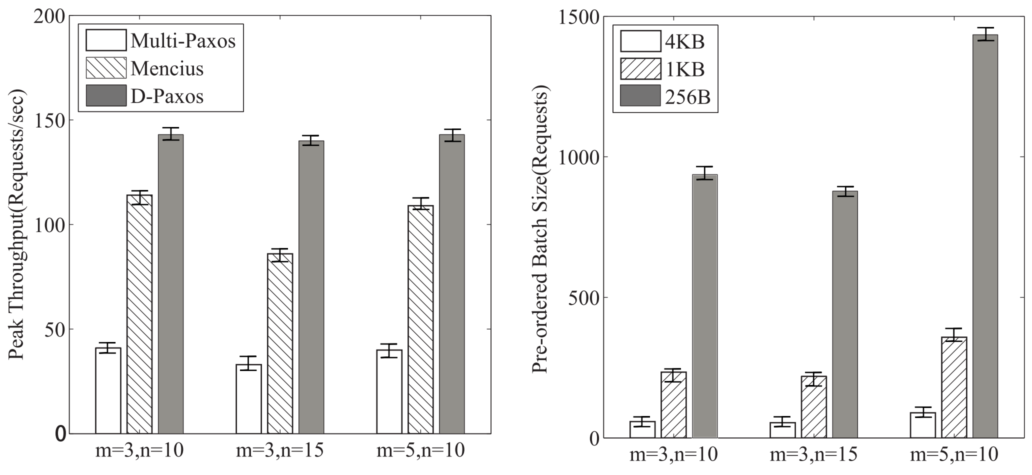


Fig. 4 (a) Peak throughput and (b) average latency of Multi-Paxos, Mencius and D-Paxos for request sizes of 4 KB, 1 KB and 256 B, in the experimental setting where three components are emulated and each one consists of 10 replicas. Each result in the figure represents the average of 10 measured values.



(a) Peak throughput

(b) Pre-ordered batch sizes of D-Paxos

Fig. 5 (a) Peak throughput of Multi-Paxos, Mencius and D-Paxos for request size of 4 KB in different settings. (b) Pre-ordered batch sizes of D-Paxos for request sizes of 4 KB, 1 KB and 256 B, when reaching maximum throughput in different settings. Each result in the figure represents the average of 10 measured values.

ity of Multi-Paxos, Mencius and D-Paxos by running them with 4 KB requests in different settings. As we can see in Fig. 5 (a), all protocols are negatively affected by the increase in the total number of replicas (i.e. either by increasing the number of replicas per component (n) from 10 to 15 or by increasing the number of components (m) from three to five). This is reasonable, because the increase in the number of replicas leads to larger quorums and thus causes reduced performance of each leader-centric instance in each protocol. Note that increasing the number of replicas per component has more negative impact on performance. The reason is subtle. Taking Mencius for instance, increasing the number of replicas per component causes more contention for LAN bandwidth shared among all replicas in each component, resulting in reduced throughput. Whereas increasing the number of components increases the available WAN bandwidth among components, which in turn offsets the impact caused by the increase in the total number of replicas.

Among all protocols, D-Paxos is the least affected one by the increase in the number of components. To better understand its reason, we present the experimental results for different pre-ordered batch sizes when D-Paxos reaches the maximum throughput in different settings. We run the experiments with request sizes of 4 KB, 1 KB and 256 B. As shown in Fig. 5 (b), regardless of request sizes, if the number of components m remains unchanged while the number of replicas n per component increases, the number of requests which can be pre-ordered in a pre-ordering phase decreases. This can be explained by the fact that more messages are required in pre-ordering phase with increasing number of replicas, which negatively impacts the batch size formed and thus D-Paxos's throughput. On the other hand, if the number of replicas per component remains unchanged while the number of components increases, the number of requests which can be pre-ordered in pre-ordering phase increases. The increase in the number of components means more delegators involve in total ordering phase, resulting in an extended rotation cycle. If the number of replicas is unchanged, an extended rotation cycle leads to a longer pre-ordering phase and a larger batch formed in each component, which offsets the impact caused by the increase in the total number of replicas. From another point of view, since delegators take turns to coordinate global instances, it can also be regarded as the increase of the logical pipelining window size. As a consequence, D-Paxos outperforms Multi-Paxos and Coordinated Paxos in scalability.

Table 2 Comparison of analytical and experimental results of the optimal batch size k for different settings.

Setting(m, n)	**S_{req}(Bytes)**	**Model(predictions)**	**Experiments**
	4K	58	57-59
3×10	1K	234	230-236
	256	937	923-947
	4K	54	54-55
3×15	1K	219	216-222
	256	878	864-888
	4K	89	88-90
5×10	1K	358	352-362
	256	1434	1410-1451

6.4 The Effect of Request Sizes and Batch Sizes

To validate the analytical model obtained in Sect. 5.2 and give an insight into the effect of request sizes and batch sizes on throughput, we run D-Paxos in different settings and study the changes of pre-ordered batch sizes and throughput as the request size increases. Table 2 shows a summary of the analytical and experimental results of the optimal batch sizes for different request sizes when D-Paxos reaches maximum throughput in different settings.

Table 2 shows that, in all cases the prediction for optimal batch size k is inside the range where the experiments reach maximum throughput, indicating that the model pro-

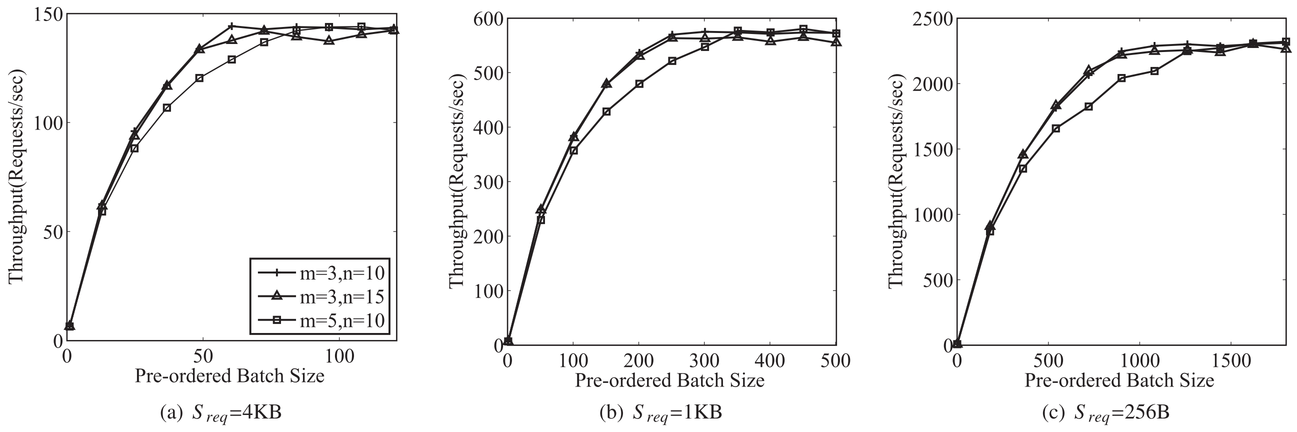


Fig. 6 Throughput with increasing pre-ordered batch sizes for request sizes of 4KB, 1KB and 256B in different settings.

vides a good approximation. In any experimental setting, the smaller request size D-Paxos uses, the larger optimal batch size it forms when reaching maximum throughput. This is because the time left for each delegator to pre-order and process requests of different sizes is stable if the rotation cycle is relatively stable. The smaller the request size, the shorter the instance latency and the more the requests can be pre-ordered. Consequently, the throughput in requests per second is higher with smaller request size. As shown in Fig. 6, in the setting where three components are emulated and each of them is composed of 10 replicas, the maximum throughput in requests per second for request size of 4 KB is about 140 requests/second, whereas that for request size of 256 B is about 2,300 requests/second.

Figure 6 also shows the throughput with increasing pre-ordered batch sizes for request sizes of 4 KB, 1 KB and 256 B in different settings. In all cases, all curves show similar trends. With smaller batch sizes, the idle time and the available link bandwidth caused by high wide-area latency are underutilized. Thus, gradually increasing batch size will make better use of resources and the throughput increases accordingly. After reaching its maximum, the throughput will be stable, even if further increasing the batch size. The reason is that the idle time due to wide-area latency has been fully taken advantage of, the increased batch size also result in the growth of time spent in receiving, sending and handling messages, which offsets the performance improvement brought by batching and logical pipelining. We also notice that curves for five components change slowly than those for three components before D-Paxos reaches the maximum throughput with request sizes of 4 KB, 1 KB and 256 B. This can be explained by the fact that the rotation cycle is extended as the number of delegators involved in D-Paxos grows and the time left for pre-ordering requests is extended accordingly. With the same request size and the same batch size specified, more time left due to more delegators involved leads to lower throughput in requests per second.

6.5 The Effect of Failures

In this section, we compare the effect of failures on Multi-Paxos, Mencius and D-Paxos and discuss how many crashes of replicas each protocol can tolerate. In this set of experiments, D-Paxos is run with 4 KB requests in a setting where five components are emulated and each of them is composed of 10 replicas. In each experiment, we run one of these three protocols over a period of time and see how failures affect it. Specifically, each experiment is run for 120 seconds, with the first 20 seconds ignored before the system becomes stable. About 60 seconds into each experiment, we crash the relevant server or component and observe the change of the throughput. Note that, since suspicion happened and relevant failover mechanisms were invoked quickly, which makes it very difficult to see what happened in each case. Therefore, the failure report is deliberately delayed for another 10 seconds so that we can see what occurs during the interval when the crash happens.

We first examine how Multi-Paxos and Mencius behave under failure. In Multi-Paxos, we discuss respectively how a crashed leader or a crashed non-leader replica affects the throughput. Figures 7 (a) and 7 (b) show the results we measured in both cases. As shown in Fig. 7 (a), the throughput quickly drops to zero when the leader crashes. Since there is only one leader in Multi-Paxos, throughput remains zero for 10 seconds until the failure is detected and a new leader is elected. The new leader then starts recovering previously unfinished instances and Multi-Paxos's throughput goes back to the level before the failure happens. Figure 7 (b) shows the throughput observed when a non-leader replica crashes. As we can see, a failed non-leader replica has little or no effect on throughput. This is because Multi-Paxos, just like Paxos, requires $2f + 1$ replicas to tolerate f faults [5]. Any single faulty, non-leader replica won't hinder from forming a quorum to make progress. As for Mencius, every replica has the opportunity to coordinate instances as a leader. All replicas in Mencius are equivalent. In this case, we discuss how a faulty replica affects Mencius's through-

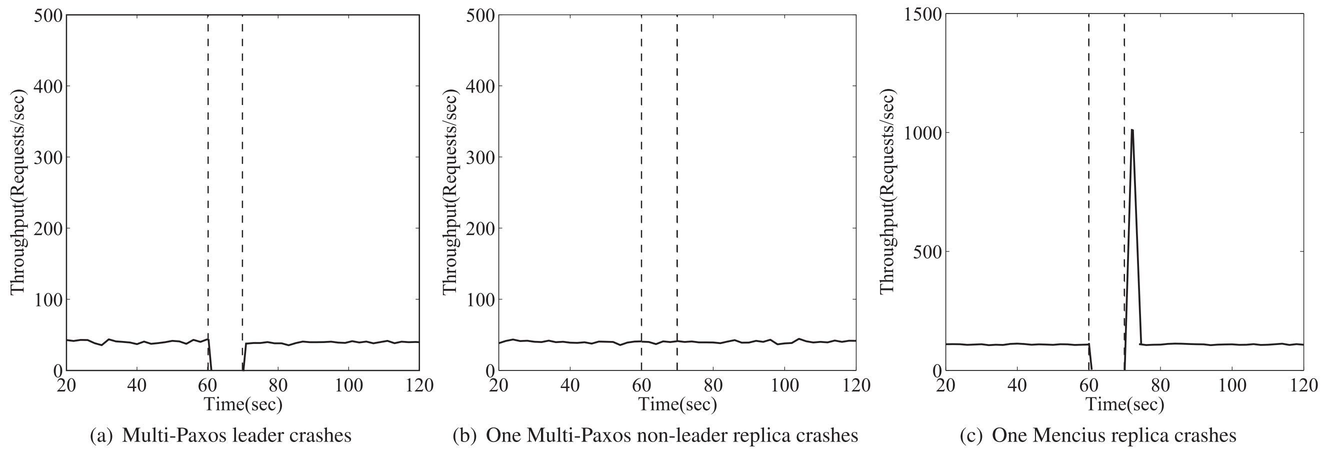


Fig. 7 Throughput of Multi-Paxos and Mencius under failure.

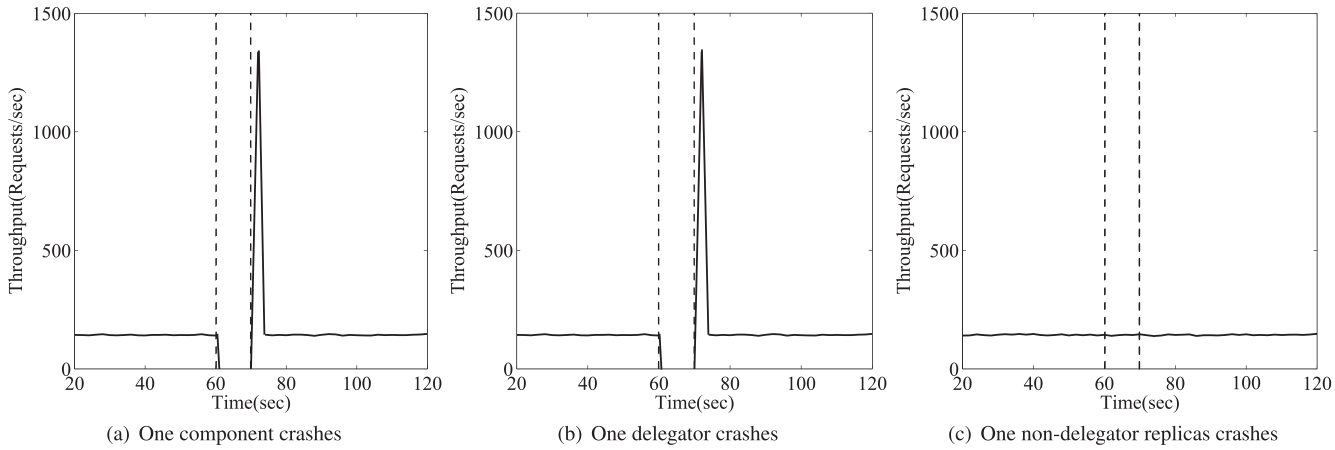


Fig. 8 Throughput of D-Paxos under failure.

put. As shown in Fig. 7 (c), the throughput quickly drops to zero when a replica crashes. When the failure detector reports the failure, a sharp spike is observed. This is because all other replicas are still able to make progress and learn instances they coordinate during the period the failure remains unreported. However, they cannot commit these instances because they have to wait for the decision of the missing instances coordinated by the faulty replica. The cumulative sequences due to the failure can be committed, resulting in a sharp spike of roughly 1,000 requests. Finally, Mencius's throughput quickly recovers.

We now discuss how failures affect D-Paxos's throughput. Because D-Paxos implements our H-RSM, replica organization in D-Paxos is different from that in Multi-Paxos and Mencius. In this scenario, experiments for D-Paxos are run for evaluating (1) the effect of a crashed component, (2) the effect of a crashed delegator in a component, and (3) the effect of a crashed replica in a component. The results in these cases are shown in Fig. 8.

Figure 8 (a) shows the instantaneous throughput when a component fails as a whole. The throughput is roughly 120 requests/second in the beginning. It quickly drops to 0 when a component fails, because the crash leads to a gap in the total ordered sequence and thus no more subsequent sequences can be committed by replicas in other components. About 10 seconds later, some delegator notices the crash and reacts by failover mechanism 2. All sequences delayed due to the gap now can be committed, which results in a sharp spike of roughly 1,200 requests. The throughput level returns to normal subsequently. The resultant curve in Fig. 8 (b) is similar to that in Fig. 8 (a). When a crash occurs, the failed delegator cannot proceed to coordinate global instances, which also result in a gap in the total ordered sequence. The overall throughput quickly drops to 0 because replicas in other components cannot commit their sequences at the moment. When the failure is reported, a re-election is initiated within the component by failover mechanism 1 and a new delegator is elected. The component is then recovered and proceeds to handle the requests. There are nine replicas left in the component after recovery. It has little effect on the overall throughput, because any quorum required for local instances and global instances to make progress is not affected by any single faulty replica. The result in Fig. 8 (c) also shows that a single replica failure does not prevent all other replicas from committing subsequent requests and has no or little effect on the overall throughput of D-Paxos.

To summarize, Multi-Paxos temporarily stalls only when the leader fails. Mencius temporarily stalls when any of the replicas crashes, while D-Paxos temporarily when any of delegators or components crashes.

Finally, we analyze how many crashes of replicas each protocol can tolerate. In the setting where all components are composed of the same number of replicas, the total number of replicas is $m \times n$, where m is the number of components and n is the number of replicas per component. As mentioned earlier, Multi-Paxos requires $2f + 1$ replicas to tolerate f faults. That is to say, Multi-Paxos can tolerate $\lceil \frac{m \times n}{2} \rceil - 1$ crashes in this setting. Mencius is a variant of Paxos. In Mencius, any single instance coordinated by a server is actually a Paxos instance with its proposals and actions restricted. Therefore, Mencius can also tolerate $\lceil \frac{m \times n}{2} \rceil - 1$ crashes.

Compared to that of Multi-Paxos and Mencius, the analysis of D-Paxos's fault tolerance is more complicated. This is because that the execution of a D-Paxos instance, especially its total ordering phase, involves interaction among servers both from components and within each components. According to the description of D-Paxos in Sect. 4.2, a global instance requires that at least a quorum ($\lceil \frac{m}{2} \rceil$ in this case) of components are correct. In other words, at most $\lceil \frac{m}{2} \rceil - 1$ components can be faulty. According to definition 5 introduced in Sect. 2.2, a component is correct if there exist a quorum ($\lceil \frac{n}{2} \rceil$ in this case) of replicas within the component that are correct. Otherwise, it is faulty. Therefore, a component is correct even it has $\lceil \frac{n}{2} \rceil - 1$ crashed replicas. Based on the above description, we discuss how many crashes of replicas D-Paxos can tolerate, under the condition that its liveness is guaranteed. In the base case, (1) there are $\lceil \frac{m}{2} \rceil$ correct components and $\lceil \frac{m}{2} \rceil - 1$ faulty ones, (2) every correct component has $\lceil \frac{n}{2} \rceil - 1$ faulty replicas, and (3) every faulty component has n faulty replicas. In this case, D-Paxos can tolerate total $\lceil \frac{m}{2} \rceil \times (\lceil \frac{n}{2} \rceil - 1) + (\lceil \frac{m}{2} \rceil - 1) \times n$ replica crashes. In the worst case, (1) there are $\lceil \frac{m}{2} \rceil$ correct components and $\lceil \frac{m}{2} \rceil - 1$ faulty ones, (2) every correct component has n correct replicas, and (3) every faulty component has $\lceil \frac{n}{2} \rceil$ faulty replicas. In this case, D-Paxos can tolerate $(\lceil \frac{m}{2} \rceil - 1) \times \lceil \frac{n}{2} \rceil$ replica crashes at most.

Taking the setting we used in this set of experiments for example, Multi-Paxos and Mencius can tolerate $\lceil \frac{5 \times 10}{2} \rceil - 1 = 24$ replica crashes. As for D-Paxos, it can tolerates total $\lceil \frac{5}{2} \rceil \times (\lceil \frac{10}{2} \rceil - 1) + (\lceil \frac{5}{2} \rceil - 1) \times 10 = 32$ replica crashes at best and, at worst, tolerate $(\lceil \frac{5}{2} \rceil - 1) \times \lceil \frac{10}{2} \rceil = 10$ replica crashes.

Thus it can be seen that D-Paxos has better fault tolerance compared to that of Multi-Paxos and Mencius at its best, but has much worse fault tolerance at its worst. The uncertainty of D-Paxos's fault tolerance is due to its organization of replicas. Besides that, the fact that D-Paxos cannot tolerate as many crashes as Multi-Paxos and Mencius at its worst is the drawback in compensation for its high performance.

7. Related Work

Our H-RSM draws inspiration from the scalable agreement protocol presented by Kapritsos et al. [17]. Scalable agreement protocol is not a specific protocol dependent upon a particular failure model. Instead, it is a generic strategy. In its architecture, multiple overlapping ordering clusters and independent execution clusters are employed to improve the overall scalability and to balance loads. All partial orders generated by each ordering clusters should be combined into a single total order for requests before being committed to execution clusters.

Given related definitions and properties of H-RSM, D-Paxos, a consensus protocol for replication across multiple data centers in the cloud, is proved to implement H-RSM. Unlike the scalable agreement protocol with overlapping virtual clusters, D-Paxos distributes pre-ordering clusters over non-overlapping data centers geographically dispersed. All partial orders for requests are combined and executed by all replicas in the system. Specifically, D-Paxos concentrates on how to tackle challenges caused by high transmission latency and the unbalanced link dependency among data centers by efficiently utilizing idle time and available bandwidth. In addition, D-Paxos provides better fault tolerance with its failover mechanisms.

Mencius [15] is a multi-leader state machine replication protocol for WANs. It has high throughput under high client load and low latency under low client load. From cluster partition point of view, Mencius can be regarded as a specific case of scalable agreement protocol with its M-N configuration being set to $M = N \geq 3$. From optimization point of view, it can be regarded as a logical extension of pipelining [14]. However, Mencius is not designed for replication across multiple data centers, each of which has a large number of redundant replicas.

Many state-of-the-art cloud storage systems, such as Megastore [18] and Spanner [1] and some commit protocols, such as MDCC [19], use Multi-Paxos [5] as an important optimization to achieve high throughput. Multi-Paxos allows the leader to execute Phase 2 of several instances in parallel. The execution of Multi-Paxos requires the leader to be relatively stable, that is, the leader won't be replaced in a long enough period of time. In D-Paxos, a delegator is elected in each data center, which can be used as a stable coordinator to coordinate Multi-Paxos instances in pre-ordering phases. Despite of a unique coordinator in each data center, D-Paxos use logical pipelining (i.e. rotating leader scheme) among delegators for further improving throughput and balancing load.

Multi-Ring Paxos [20] uses multiple instances of Ring Paxos [21] to order all requests in which those instances have an interest and thus scale throughput without sacrificing response time. Multi-Ring Paxos behaves similarly to D-Paxos, because learners subscribing to multiple groups of Ring Paxos instances should also combine multiple pre-ordered sequences of requests from them. However, Ring

Paxos use techniques, e.g. IP multicast, that are only available on a LAN. Therefore, Multi-Ring Paxos does not apply to replication across multiple data centers in the cloud. Besides, neither batching nor pipelining is adopted in Multi-Ring Paxos.

Steward [22] is a hierarchical replicated state machine architecture for WAN. A group of servers in a site is converted into a logical entity that works as a single participant in a wide-area protocol. However, the logical entity is not achieved through state machine replication approach. Consequently, requests from a site are transmitted directly to the upper WAN protocol rather than being pre-ordered within the site. Steward can withstand Byzantine failure but subject to a single site compromise.

8. Conclusion

In a wide-area cloud environment where data is replicated across multiple data centers, an efficient way to improve the throughput is to make full use of idle resources (such as idle time and available bandwidth). In this paper, we defined an H-RSM based on the idea of parallel processing. D-Paxos was designed for replication among replicas distributed across multiple data centers. We proved that D-Paxos implemented consensus and satisfied the safety and liveness properties of our H-RSM, so that it can be used to implement an H-RSM.

D-Paxos provides high throughput with batching (delegators can pre-order requests by effectively utilizing idle time and available bandwidth to form pre-ordered batches and propose batches in global instances) and logical pipelining (delegators can utilize more wide-area bandwidth provided by components and share coordination load by using a rotating-leader scheme). Our experimental results also demonstrate that D-Paxos has better throughput and scalability than other Paxos variants for replication across multiple components. D-Paxos provides good fault tolerance, even to the outage of a single component due to network partitions, facility-wide outages, etc.

The way in which we build our H-RSM upon two separate levels provides a good deal of flexibility to customize D-Paxos's local protocol and global protocol for matching environment change. Our future work includes evaluating our D-Paxos's performance on a real world platform and introducing more parameters so as to provide a more general analytical model and improves its prediction accuracy.

Acknowledgments

We would like to thank the anonymous reviewers whose comments and suggestions have dramatically improved the quality of the paper. This work was partially supported by the Second Batch of Strategic Emerging Industrial Core Technology Research Project in Guangdong Province under Grant No.2012A010701005, the Key Lab of cloud computing and big data in Guangzhou under Grant No. SITGZ[2013]268-6, Engineering and Technology Research Center of Guangdong Province for Big Data Intelligent Processing under Grant No. GDDST[2013]1513-1-11, Key Project of the Combination of "Production, Education and Research" supported by Guangdong province and Ministry of Education, under Grant No.2012B091000109 and Science and Technology Program in Guangzhou, China (International Science and Technology Cooperation Program) under Grant No. 2012J5100018.

References

[1] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," ACM Trans. Comput. Syst., vol.31, no.3, pp.251–264, 2013.
[2] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," Proc. VLDB Endow., vol.1, no.2, pp.1277–1288, 2008.
[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," ACM SIGOPS Oper. Syst. Rev. (ACM), vol.44, no.2, pp.35–40, 2010.
[4] F.B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," ACM Comput. Surv., vol.22, no.4, pp.299–319, 1990.
[5] L. Lamport, "Paxos made simple," ACM SIGACT News, vol.32, no.4, pp.18–25, 2001.
[6] J. Kończak, N. Santos, T. Żurkowski, et al., "JPaxos: State machine replication based on the Paxos protocol," Tech. Rep., EPFL-REPORT-167765, Faculté Informatique et Communications, Switzerland, 2011.
[7] L. Lamport, "Byzantizing Paxos by refinement," Distributed Computing, Lect. Notes Comput. Sci., vol.6950, pp.211–224, Springer Verlag, Berlin, Heidelberg, 2011.
[8] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of distributed consensus with one faulty process," J. Assoc. Comput. Mach., vol.32, no.2, pp.374–382, 1985.
[9] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," J. Assoc. Comput. Mach., vol.43, no.2, pp.225–267, 1993.
[10] T.D. Chandra, V. Hadzilacos, and S. Toueg, "Weakest failure detector for solving consensus," J. Assoc. Comput. Mach., vol.43, no.4, pp.685–722, 1996.
[11] M. Larrea, A. Lafuente, I. Soraluze, R. Cortiñas, and J. Wieland, "Designing efficient algorithms for the eventually perfect failure detector class," J. Software, vol.2, no.4, pp.1–11, 2007.
[12] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," IEEE Trans. Comput., vol.C-28, no.9, pp.690–691, 1979.
[13] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," 2000 Proc. Int. Conf. on Distributed Computing Systems, pp.464–474, IEEE Computer Society, Washington, DC, USA, 2000.
[14] N. Santos and A. Schiper, "Optimizing Paxos with batching and pipelining," Theor. Comput. Sci., vol.496, pp.170–183, 2013.
[15] Y. Mao, F.P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," Proc. 7th USENIX Symp. on Oper. Syst. Des. Implement., pp.369–384, USENIX Association, Berkeley, 2008.
[16] M. Hibler, R. Ricci, L. Stoller, et al., "Large-scale virtualization in the Emulab network testbed," 2008 USENIX Annu. Tech. Conf., pp.113–128, USENIX Association, Berkeley, 2008.

[17] M. Kapritsos and F.P. Junqueira, “Scalable agreement: Toward ordering as a service,” Proc. Sixth Work. on Hot Top. Syst. Dependability, pp.1–8, USENIX Association, Berkeley, 2010.
[18] J. Baker, C. Bond, J.C. Corbett, et al., “Megastore: Providing scalable, highly available storage for interactive services,” Proc. CIDR 2011, 5th Bienn. Conf. Innovative Data Syst. Res. (CIDR), pp.223–234, 2011.
[19] T. Kraska, G. Pang, M.J. Franklin, S. Madden, and A. Fekete, “MDCC: Multi-data center consistency,” Proc. 8th ACM European Conf. on Computer Syst., EuroSys 2013, pp.113–126, Association for Computing Machinery, New York, 2013.
[20] P.J. Marandi, M. Primi, and F. Pedone, “Multi-ring Paxos,” Proc. Int. Conf. Dependable Syst. Networks, pp.1–12, IEEE Computer Society, Washington, 2012.
[21] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone, “Ring Paxos: A high-throughput atomic broadcast protocol,” Proc. Int. Conf. Dependable Syst. Networks, pp.527–536, IEEE Computer Society, Washington, 2010.
[22] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, “Steward: Scaling Byzantine fault-tolerant replication to wide area networks,” IEEE Trans. Dependable Secure Comput., vol.7, no.1, pp.80–93, 2010.

Fagui Liu was born in 1963. She received her M.A. degree from Beihang University (BUUA) and Ph.D. degree from South China University of Technology (SCUT). Currently she is a professor at the School of Computer Science and Engineering, South China University of Technology. She has worked in computer systems and software engineering for over 25 years both in industry and academia. Her recent research interests include cloud computing and big data, cloud-based software factory.

Yingyi Yang was born in 1982. He received his B.S. and M.A. degrees from University of Electronic Science and Technology of China (UESTC). Currently he is a Ph.D. candidate at the School of Computer Science and Engineering, South China University of Technology (SCUT). He is a member of The Institute of Electronics, Information and Communication Engineers (IEICE). His current research interests include cloud computing, distributed computing, replicated cloud storages, big data and consensus protocols.