Hindawi Publishing Corporation
The Scientific World Journal
Volume 2015, Article ID 704049, 13 pages
http://dx.doi.org/10.1155/2015/704049

Research Article

HT-Paxos: High Throughput State-Machine Replication Protocol for Large Clustered Data Centers

Vinit Kumar[1] and Ajay Agarwal[2]

[1]*Krishna Engineering College, Ghaziabad 201007, India*
[2]*SRM University, Delhi-NCR Campus, Modinagar, Ghaziabad 201204, India*

Correspondence should be addressed to Vinit Kumar; vinitbaghel@gmail.com

Received 20 August 2014; Accepted 14 January 2015

Academic Editor: Rajesh Jeewon

Paxos is a prominent theory of state-machine replication. Recent data intensive systems that implement state-machine replication generally require high throughput. Earlier versions of Paxos as few of them are classical Paxos, fast Paxos, and generalized Paxos have a major focus on fault tolerance and latency but lacking in terms of throughput and scalability. A major reason for this is the heavyweight leader. Through offloading the leader, we can further increase throughput of the system. Ring Paxos, Multiring Paxos, and S-Paxos are few prominent attempts in this direction for clustered data centers. In this paper, we are proposing HT-Paxos, a variant of Paxos that is the best suitable for any large clustered data center. HT-Paxos further offloads the leader very significantly and hence increases the throughput and scalability of the system, while at the same time, among high throughput state-machine replication protocols, it provides reasonably low latency and response time.

1. Introduction

State-machine replication (SMR) is a fundamental technique for increasing availability of the system [1, 2]. It lies in the heart of the many real time applications. Replicating a service on multiple servers ensures that even if some replica fails the service is still available. State-machine replication prevalently uses the variants of Paxos. Google's Megastore [3], chubby lock service [4], and yahoo's Zab [5] are few of the popular applications that use the variant of Paxos. Since in leader based protocols, leader does most of the work, the bottleneck is found at the leader and the maximum throughput is limited by the leader's resources (such as CPU and network bandwidth), further increasing the number of client requests decreases the throughput. Since the bottleneck is at the leader, more additional replicas will not improve performance; in fact, it decreases throughput since the leader requires to process additional messages.

Ring Paxos [6] offloads the leader by adopting the concepts of (i) ordering of client IDs by the leader instead of full requests, (ii) dissemination of requests and learned-ids by the leader through IP-multicasting, (iii) a ring of acceptors, (iv) batching of requests at leader, and (v) pipelining (i.e., parallel execution of ring Paxos instances). Concept of ip-multicasting allows the leader to order the IDs instead of full requests and hence offloads the leader. Ring of acceptors reduces the number of messages sent to other acceptors and received from other acceptors by the leader. Because of the ring, learners learn the decision only from the leader. In other Paxos protocols those are optimized for latency instead for messages and learners learn the decision from any quorum of acceptors; that is, ring of acceptors reduces the load on the learners. Batching of requests at leader also significantly offloads the leader. Moreover, concept of cheap Paxos reduces the latency.

However, in ring Paxos leader still requires to handle all client communications. Moreover, it assigns unique ID to client requests and sends all client requests with their ID to all acceptors and learners. Furthermore, leader forwards ID to the first acceptor of the ring and on receiving ID from the last acceptor of the ring, leader broadcasts their decision to all the acceptors and learners. In ring Paxos, clients also require knowing about the leader; if leader fails then service will interrupt until the election of a new leader.

S-Paxos [7] offloads the leader by using the concepts as (i) distributing the work of handling all client communications among all nonfaulty replicas, (ii) disseminating client requests among replicas in a distributed fashion, (iii) ordering of IDs by the leader instead of full client requests, (iv) batching the client requests, and (v) pipelining. Receiving of client request by any replica has certain advantages as it offloads the leader and failure of the leader does not interrupt service.

However, in S-Paxos, every nonfaulty replica including leader receives all client requests either directly from clients or through other replicas. All these client requests may reach the leader in less number of messages because of the batching at various replicas (unlike ring Paxos). Moreover, leader may not require disseminating all client requests because of the aforementioned second concept of S-Paxos, (unlike ring Paxos) but partially disseminates the client requests and partially handles client communications. In addition, leader uses classical Paxos for ordering IDs (instead of full client requests), so leader and other replicas handle all messages belonging to classical Paxos. A large number of messages at leader adversely affect throughput.

Multiring Paxos [8] uses the concept of state partitioning [9] for increasing the throughput of the system. Each partition uses a different instance of ring Paxos. The performance of ring Paxos directly affects this protocol.

In this paper, we are proposing HT-Paxos (HT stands for High Throughput) a variant of Paxos that adopts all aforementioned concepts of S-Paxos for offloading the leader. In addition, HT-Paxos adopts few major concepts as (i) eliminating the work of handling client communications and request dissemination from the leader; that is, leader does not require either receiving or disseminating the client requests; instead it only receives the batch IDs (or request IDs) and orders them (unlike S-Paxos and ring Paxos). This significantly reduces acknowledgement messages at disseminators in large clustered data centers (unlike S-Paxos, where every disseminator sends acknowledgement messages to every other disseminator). In this way, leader as well as other disseminators becomes truly lightweight and hence for any large clustered data center, HT-Paxos provides significantly higher throughput.

Organization of this paper is as follows: to make the paper self-sufficient; next section revisits Paxos. Section 3 presents a system model. HT-Paxos is discussed in Section 4. Section 5 presents a comparative analysis of proposed work with other related works. Finally, concluding section discusses the advantages of HT-Paxos.

2. Revisiting Paxos

Under this section, theory of Paxos is briefly reviewed. Paxos is a family of protocols that implements a replicated state-machine and assumes a distributed system of processes communicating via messages. Processes can fail only by stopping, and messages can be lost or duplicated but not corrupted. Timely actions by nonfailed processes and timely delivery of messages among them are required for progress; safety is maintained despite arbitrary delays and any number of failures.

Any of the Paxos protocol has three types of agents: proposers, acceptors, and learners. In an implementation, a single process may act as more than one agent. Proposers propose the commands. Acceptors choose the sequence of commands and Learners learn and execute the commands. If only one proposer proposes all the commands and resolves the conflicts then it is called a leader.

2.1. Classical Paxos. In classical Paxos [10, 11] clients send their command to the leader. Leader creates a separate instance of Paxos protocol for every command and assigns an instance number to each instance sequentially. If an instance of Paxos protocol at any server communicates with another server then another server creates a new instance of Paxos protocol, with the same instance number provided the same instance number does not exist. But, if leader fails then leader election protocol elects a new leader.

Every instance of Paxos protocol takes one or more rounds to decide on a single output value. A successful round has two phases.

Phase 1a. Proposer (leader) selects a proposal number n and sends a prepare message that contains a proposal along with proposal number n to a majority of acceptors.

Phase 1b. If the proposal number n of the current proposal is larger than the proposal number of any previous proposals, then Acceptor promises not to accept proposals less than n and sends the last accepted proposal (if any) to the proposer. Otherwise, acceptor sends a denial to the proposer.

Phase 2a. If the Proposer receives a response (numbered n) from a majority of acceptors then it chooses the highest numbered proposal received from all such responses. If the proposer does not receive any accepted proposal then it chooses any one of the proposed proposals. Now proposer sends an accept message to a majority of acceptors along with a chosen proposal and proposal number n.

Phase 2b. If an acceptor receives an accept message for a proposal numbered n then it accepts the proposal unless it has already responded to a prepare message having a proposal number greater than n. After accepting the proposal, it sends an accepted message along with accepted proposal to the proposer and every learner. A round fails when multiple proposers send conflicting prepare messages or the proposer does not receive a majority of responses. In these cases, another round starts with a higher proposal number.

In addition, different proposers choose their proposal numbers from the disjoint sets of numbers. Therefore, two different proposers never issue a proposal with the same proposal number. Moreover, each proposer maintains the highest numbered proposal with proposal number in a stable storage and phase 1 always uses a higher proposal number than any it has already used. An acceptor always records its intended response in a stable storage before actually sending the response. Furthermore, every learner executes the learned commands sequentially as per the instance numbers.

2.1.1. Optimizations of Classical Paxos. If leader is relatively stable then phase one becomes quite unnecessary. Thus, it is possible to skip phase one for future instances of the protocol with the same leader. To achieve this, the instance number is included along with each value. It reduces the failure free message delay (proposal to learning) from four delays to two delays.

Another optimization reduces the number of messages, as phase 2b messages reach only the leader; if leader receives such messages for the same value from majority of acceptors then leader decides this value and sends this decision to all learners. However, this optimization increases the latency.

2.2. Fast Paxos. Fast Paxos [12] generalizes basic Paxos to reduce end-to-end message delays. In basic Paxos, the message delay from client request to learning is three message delays. Fast Paxosallows two message delays but requires the Client to send its request to multiple destinations. Intuitively, if the leader has no value to propose, then a client could send an accept message to the acceptors directly. The acceptors would respond as in basic Paxos, sending accepted messages to the leader and every learner achieving two message delays from client to learner. If the leader detects a collision, it resolves the collision by sending accept messages for a new round, which are accepted as usual. This coordinated recovery technique requires four message delays from client to learner. The final optimization occurs when the leader specifies a recovery technique in advance, allowing the acceptors to perform the collision recovery themselves. Thus, uncoordinated collision recovery can occur in three message delays (and only two message delays if all learners are also acceptors).

2.3. Generalized Paxos. Generalized Paxos [13] generalizes the classical Paxos, multi-Paxos, and fast Paxos. Moreover, it explores the relationship between the operations of a distributed state-machine for improving performance. When conflicting proposals are commutative operations of the state-machine, in such cases, coordinator accepts all such conflicting operations at once, avoiding the delays required for resolving conflicts and reproposing the rejected operation. This Paxos uses ever growing sets of commutative operations, after some reasonable time, these sets become stable and then leader accepts this set. Larger set reduces the number of messages and time taken by the state-machine.

2.4. Ring Paxos. Ring Paxos [6] has a logical ring of acceptors. One acceptor of the ring plays a role of the coordinator (leader). Coordinator accepts client requests and assigns a unique ID to each client request. Moreover, in phase 1, coordinator and majority of acceptors make an agreement about the ring of acceptors. Moreover, As soon as coordinator receives enough client requests to make a batch or timeout reaches, phase two triggers. In phase 2, coordinator ip-multicasts the client requests along with their IDs, round number, and instance number to all acceptors and learners. Ring Paxos executes consensus on IDs.

Upon receiving a phase 2 message, first acceptor in the ring creates a small message containing the round number, IDs, and its own decision and forwards it along the logical ring. Moreover, upon receiving a message from an acceptor of the ring, other than coordinator, each acceptor in the ring appends its decision to the message and forwards it along the logical ring, if it has the corresponding client requests.

Upon receiving the phase 2 message from the last acceptor of the ring, coordinator informs all the learners that some IDs have been chosen. In high load conditions, this information can be piggybacked on the next ip-multicast message. Moreover, learner delivers the corresponding client value in the appropriate instance.

2.5. Multiring Paxos. Multiring Paxos [8] uses the concept of logical partitioning for increasing the throughput of the system; proposers, acceptors, and learners subscribe to one or more logical partitions. Each partition uses a different instance of Ring Paxos.

2.6. S-Paxos. S-Paxos [7] assumes that all the replicas servers play the roles of all agents and out of them, one replica plays a role of the leader. Moreover, any client may send their request with their unique IDs to any of the replica. Replica accepts client requests and creates a batch that contains client requests and their IDs. After that, replica assigns an ID to that batch. Now replica forwards this batch and batch ID to all the replicas (including self). When a replica receives a forwarded batch with their batch ID, it records the batch and the batch ID in the requests set. It then sends an acknowledgment containing the batch ID to all replicas. Replica retransmits acknowledgement message periodically until batch stabilizes. Batch stabilizes after receiving $f + 1$ acknowledgments from different replicas for a particular batch ID (f represents an upper bound for faulty replicas). A replica records this fact by adding the batch ID to its *stable_Ids* set. If replica receives an acknowledgement for a particular batch id from any replica q but does not has corresponding batch then it requests q for resending the corresponding batch.

Moreover, the leader replica passes the batch IDs available in *stable_Ids* set to the ordering layer, which will then use the classical Paxos protocol to order it. Here it is significant that classical Paxos achieves consensus on IDs rather than full requests. Replicas execute client requests in the order as suggested by classical Paxos. After executing the request, the replica that received the request from the client sends the corresponding reply. In the low load condition, we may avoid batches but S-Paxos is designed for high throughput; therefore, batching and pipelining are quite well desirable. In high load conditions, any outgoing message that contains any batch may piggyback acknowledgement messages.

3. System Model

HT-Paxos is a variant of Paxos. However, we have divided the role of acceptors into two separate subcategories as (i) disseminators and (ii) sequencers. In this way, HT-Paxos has four classes of agents: proposers (clients), disseminators,

sequencers, and learners. One sequencer assumes a role of the leader. Proposers propose proposals (requests); disseminators accept proposals and disseminate accepted proposals to all other disseminators and learners; sequencers work for establishing an order by using classical Paxos (classical Paxos uses both sequencers and learners for determining and learning an order). Learners receive proposals from disseminators and execute them in an order as indicated by the leader. Although agents work differently for improving throughput but fundamental guarantees (nontriviality, stability, consistency, and liveness) of Paxos are the same in HT-Paxos.

It is proposed that clustered data center has two LANs (local area networks), calling them first LAN and second LAN. All disseminators and learners subscribe to both the LANs. Moreover, all sequencers subscribe to the second LAN. Furthermore, proposers either subscribe to the both the LANs or connect both the LANs via one or more routers.

Any computing node that has a disseminator will also have a learner and in such nodes, both agents can share all incoming messages and data structures. Moreover, nodes that have sequencers do not have any other agent. Furthermore, each computing node has two buffers one for incoming messages and another for outgoing messages for each LAN.

Like classic Paxos, it is assumed that agents communicate by sending messages. These messages can take arbitrarily long for reaching their destinations, can be delivered out of order, can be duplicated, and can be lost. Moreover, system detects all corrupted messages and considers such messages (corrupt in communication medium and finally detected) as lost. Furthermore, agents discard duplicate messages and learners discard duplicate proposals.

Like classic Paxos, the customary partially synchronous, distributed, and non-Byzantine model of computation is assumed where agents operate at arbitrary speed, may fail by stopping, may restart, and always perform an action correctly. Agents have access to stable storage whose state survives failures.

Further it is assumed that at least (i) $\lfloor n/2 \rfloor + 1$ disseminator will always remain nonfaulty out of the total n disseminators, (ii) $\lfloor m/2 \rfloor + 1$ sequencers will always remain nonfaulty out of the total m sequencers, and (iii) one learner will always be nonfaulty.

Optimized version of HT-Paxos with slight modification is discussed in Section 4.2.

For sending a message, two primitives (i) send ⟨*message*⟩ to one receiver (ii) multicast ⟨*message*⟩ to multiple receivers are used. Send primitive is for one to one communication and Multicast primitive represents that sender sends a single message but specified multiple receivers can receive this message. Multicasting can be implemented by using Ethernet/hardware multicasting or by IP-multicasting or by Dr. Multicast. Dr. Multicast [14] explains that IP multicast in data centers becomes disruptive in the presence of large number of groups and requires a proper administrative control. However, HT-Paxos has only few groups. In addition, use of multiple LANs further reduces the number of groups per LAN.

Like S-Paxos, all activities of HT-Paxos are divided into two layers, (i) dissemination layer and (ii) ordering layer. All work performed by classical Paxos comes under ordering layer and rest of the work (dissemination of requests) is related to the dissemination layer.

4. HT-Paxos

4.1. Basic Algorithm

4.1.1. An Overview. Any client sends their request (with a *request_value* and a unique *request_id*) to any one disseminator (randomly chosen) using first LAN. Moreover, if client does not receive a reply message ⟨*request_id*⟩ in a reasonably long time, then it periodically sends same request to any one disseminator (randomly chosen) using first LAN until it gets a reply. Furthermore, if client gets a reply message, then it replies with ⟨*request_id*⟩ message to that disseminator using second LAN.

If request is available from any client then disseminator receives a request. After that, it multicasts this request using first LAN to all disseminators and learners. Moreover, when a disseminator receives a request from any disseminator then (i) it records the request in the *requests_set*, (ii) replies back an acknowledgment message ⟨*request_id*⟩ to that disseminator using second LAN and (iii) periodically multicasts ⟨*request_id*⟩ message to all sequencers using second LAN until *request_id* become an element of decided set.

Disseminator that received the request from the client sends a reply message ⟨*request_id*⟩ to the corresponding client using second LAN in either of the two conditions (i) on receiving ⟨*request_id*⟩ message from at least a majority of disseminators (including self) or (ii) on observing that *request_id* is an element of decided set. Moreover, this disseminator periodically sends a reply to the corresponding client until it gets a reply message ⟨*request_id*⟩ or detects a failure of the client.

If disseminator does not receive sufficient desired acknowledgement messages ⟨*request_id*⟩ then it periodically multicasts ⟨*request_id*⟩ message to all disseminators using second LAN until it receives desired acknowledgment message(s) or when *request_id* becomes an element of decided set.

If any disseminator p receives ⟨*request_id*⟩ message from any disseminator q but does not has the corresponding request, then p sends a message ⟨Resend, *request_id*⟩ to q using second LAN. Moreover, on receiving ⟨Resend, *request_id*⟩ message from any disseminator p, disseminator q sends the corresponding request to the disseminator p using first LAN.

After receiving a ⟨*request_id*⟩ message from any learner, disseminator replies with the corresponding request to that learner.

After receiving same ⟨*request_id*⟩ messages from at least a majority of disseminators, sequencer inserts this *request_id* into its *stable_ids* set.

Moreover, leader repeatedly launches (up to the allowable number of instances at a time) an instance of classical Paxos for each *request_id* from the *stable_ids*. Classical Paxos uses second LAN for their all communications. After learning a *request_id*, learner inserts this *request_id* into the decided set.

Each disseminator also maintains *requests_set* at the permanent storage device, initially the value of this set is null and at every startup, and disseminator will initialize this set

through reading permanent storage device. Moreover, if learner is not at disseminator's site then learner similarly maintains this set. Furthermore, if learner is on disseminator's site then learner does not maintain this set but may read this set when required.

Each learner also maintains decided set at permanent storage device and initially the value of this set is null and at every startup, learner will initialize this set through reading permanent storage device. The disseminator on the same computing node may read this set when required.

Each sequencer also maintains *stable_ids* and decided set at permanent storage device and initially the values of both these sets are null and at every startup, sequencer will initialize these sets through reading permanent storage device.

4.1.2. Pseudocode of Dissemination Layer. See Algorithm 1.

4.1.3. Ordering Layer. Ordering layer uses classical Paxos (by adopting aforementioned optimizations) which is a well-defined theory in literature. Instead of using the *request_value* from any client, classical Paxos achieves consensus on the *request_id* available in the *stable_ids*. Every learner learns *request_id* sequentially as per the instance numbers of classical Paxos and inserts these *request_id* into the decided set.

When the leader fails, only sequencers are required to participate in leader election process, one of the nonfaulty sequencers assumes the role of the leader. Clients, disseminators, and learners are not required to know who the leader is. In HT-Paxos, leader election process does not affect request dissemination (i.e., no burden on disseminator and learner sites, unlike S-Paxos).

In HT-Paxos, all sequencers maintain only two sets (i) *stable_ids* set and (ii) decided set (unlike S-Paxos, where every replica maintains four sets). The leader sequentially proposes a *request_id* from the *stable_ids* set, in a new instance of classical Paxos (up to the allowable number of instances at a time). When leader learns a *request_id* after receiving phase 2b messages (of classical Paxos), it inserts this *request_id* into the decided set and then deletes this *request_id* from the *stable_ids* set. New leader always makes sure that before proposing new *request_id* from *stable_ids*, all the *request_ids* received in phase 1b messages (of classical Paxos) must be decided by as usual working of classical Paxos.

Unlike S-Paxos, HT-Paxos does not require proposed and reproposed sets, even though, same optimization (i.e., no duplicate *request_id* will be proposed by the new leader) as claimed in S-Paxos will be achieved here.

4.2. Optimizations of HT-Paxos. Before multicasting any request to all disseminators and learners, a disseminator can wait for a certain time for more requests from one or more clients and then group them into a batch and assign them a unique *batch_id*; after that, this disseminator multicasts $\langle batch\_id, batch\rangle$ message to all disseminators and learners. Upon receiving a $\langle batch\_id, batch\rangle$ message any disseminator replies $\langle batch\_id\rangle$ message to that disseminator and multicasts $\langle batch\_id\rangle$ to all sequencers. Rest of the procedure of HT-Paxos applies to the *batch_id*, similarly as the *request_id*. At sequencers, *stable_ids* set will contain *batch_ids* and classical Paxos will order the *batch_ids*. Since the ordering layer uses the classical Paxos, it can use the traditional optimizations of batching and pipelining, as well as any other optimization that applies to the classical Paxos.

Another optimization is to piggyback the acknowledgments on the messages used to forward batches; disseminator sends separate acknowledgment messages only in the absence of such messages. This optimization is especially effective when the system is under high load.

In proposed protocol, two LANs are used. However, one may use one or more LANs depending upon various factors. These factors may be either technological or economical. Use of multiple LANs may increase the reliability and performance of the communication network. As per [14], increasing more multicast group can degrade the performance of the communication network. We can reduce multicast groups per LAN using more LANs. This may have a positive impact on performance. If we do not have a technology for required bandwidth in a LAN then in such case, use of multiple LANs can provide the required bandwidth using the same technology.

Further, proposed optimization increases the fault tolerance of the system for any given number of total computing nodes. In this optimization, it is assumed that all disseminator sites also have a sequencer. This optimization may increase fault tolerance of the system but at the cost of comparatively (as compared to HT-Paxos without this optimization) lower throughput of the system. However, throughput under this optimization is still better than any other aforementioned high throughput protocols. We believe that increasing too much fault tolerance at the cost of performance is unnecessary for any large clustered data center, since massive failures are the rarest events.

4.3. Safety

4.3.1. Safety Criteria. For the safety of any protocol that implements state-machine replication, no two learners can learn the values in different order despite any number of (in our case, non-Byzantine) failures.

4.3.2. Proof of Safety (Sketch). Proposed protocol fulfills the safety requirement by adopting the following provisions.

Nontriviality. Learners can learn only values (client requests or batches) as indicated by classical Paxos. Nontriviality ensures that learners can learn only the proposed values (client requests). As per the proposed protocol, leader of the classical Paxos can only propose the *request_id* or *batch_id* that corresponds to client requests; therefore, learners can learn only the *request_id* or *batch_id* and, hence, corresponding request or batch of requests.

Consistency. Learners can learn the requests only in same sequence as indicated by classical Paxos. Since classical Paxos is a well-proven theory of literature that guarantees safety, therefore, no two learners can learn the values (client requests) in different order.

(1) /* ***Task of a proposer*** (***client***) */
(2) Create a new *request*
(3) Choose any disseminator d randomly.
(4) Send <*request*> to d using first LAN
(5) **Upon** not receiving any reply message <*request_id*> from any disseminator until $\Delta 1$ time,
(6) **Repeat** from step 3
(7) **Upon** receiving a reply message <*request_id*> from any disseminator d
(8) Send <*request_id*> to d using second LAN
(9) **If** (want to send more *requests*?)
(10) **Repeat** from step 2,
(11) **Else,** exit.

(12) /* ***Task of a disseminator*** */
(13) **Upon** receiving <*request*> from any client
(14) Multicast <*request*> to all disseminators and learners using first LAN
(15) **Upon** receiving <*request*> from any disseminator d
(16) $Requests\_set \leftarrow Requests\_set \cup request$
(17) Send <*request_id*> to d using second LAN
(18) Multicast <*request_id*> to all sequencers using second LAN,
(19) **Repeat** from step 18 after every $\Delta 2$ time, **until** ($request\_id \in decided$)
(20) **Upon** receiving <*request_id*> message from at least a majority of disseminators or $request\_id \in decided$
(21) **If** (received the corresponding *request* from the client)
(22) **Then**
(23) Send <*request_id*> to the corresponding client using second LAN
(24) **Repeat** from step 23 after every $\Delta 3$ time, **until** it receives a reply message <*request_id*> from the corresponding client or client's failure is detected
(25) **Upon** receiving <*request_id*> from any disseminator q, $\begin{pmatrix} \forall request : request \in requests\_set \\ \wedge request\_id \notin request \end{pmatrix}$ and after $\Delta 4$ time
(26) Send <Resend, *request_id*> to q using second LAN
(27) **Upon** receiving <Resend, *request_id*> from any disseminator p
(28) Send <*request*> to p using first LAN
(29) **Upon** receiving <Resend, *request_id* > from a learner l
(30) **If** $\begin{pmatrix} \exists request : request \in requests\_set \\ \wedge request\_id \in request \end{pmatrix}$
(31) Send < *request* > to l using first LAN
(32) **If** $\left(request\_id \in decided \wedge \begin{pmatrix} request \notin requests\_set : \\ request\_id \in request \end{pmatrix} \right)$
(33) Send <Resend, *request_id*> to any other disseminator using second LAN
(34) **Upon** not receiving corresponding *request* after $\Delta 5$ time **Repeat** from step 32

(35) /* ***Task of a sequencer*** */
(36) **Upon** receivingsame <*request_id*> from at least a majority of disseminators
(37) $Stable\_ids \leftarrow Stable\_ids \cup request\_id$

(38) /* ***Task of a learner*** */
(39) **If** (learner is not at disseminator's site)
(40) **Then**
(41) **Upon** receiving <*request*> from any disseminator
(42) $Requests\_set \leftarrow Requests\_set \cup request$
(43) **If**

$$\left(\left(\begin{array}{l} \forall request\_id : \\ \text{learner has learned } request\_id \wedge \\ (request \notin requests\_set : request\_id \in request) \end{array} \right) \right)$$

(44) Send <Resend, *request_id*> to any disseminator using second LAN
(45) **Upon** not receiving corresponding *request* after $\Delta 6$ time **Repeat** from step 43
(46) **Execute** *requests* in an order as provided by ordering layer.

Algorithm 1: Dissemination layer of HT-Paxos.

4.4. Progress. HT-Paxos ensures that if any client receives a reply for their request or request becomes an element of *stable_id* set at any disseminator then all available learners will surely learn that request. Moreover, protocol also ensures that if client does not crash for an enough time then client will definitely receive a reply for their request.

4.4.1. Requirements for Ensuring Progress. At least $\lfloor m/2 \rfloor + 1$ sequencers out of total m, $\lfloor n/2 \rfloor + 1$ disseminators out of total n and one learner are always required to remain nonfaulty for the progress of the proposed protocol (these requirements are only for ensuring progress. Safety does not require these conditions).

4.4.2. Proof of Progress (Sketch). As per the protocol, if any client sends a request to any disseminator, there could be two cases; it may be faulty or nonfaulty. If disseminator is faulty then client will not receive a reply for this request. Therefore, client will resend the request to any randomly chosen disseminator. Since system always has at least a majority of nonfaulty disseminators, therefore, there is a fair chance that one of the nonfaulty disseminators will receive the client request.

If nonfaulty disseminator receives a request from any client, after that this disseminator may or may not fail before forwarding the request to all disseminators and learners. If disseminator fails then client will not receive a reply and therefore will resend the request to any randomly chosen disseminator. This phenomenon may get repeated up to maximum f times, where $f = \lfloor n/2 \rfloor$, because system may have only maximum f faulty disseminators.

If disseminator does not fail and forwards the request to all disseminators and learners then some or all disseminators and learners may or may not receive request due to the message loss. If no disseminators receive request due to message loss and sender disseminator fails then client will not receive a reply; in this case, client will resend the request. This phenomenon may repeat up to maximum f times, because system may have only maximum f faulty disseminators.

If some or all disseminators receive request from any disseminator, then all such disseminators reply $\langle request\_id \rangle$ message. If disseminator does not receive replies from at least a majority of disseminators in a certain time limit for the request and also observes that *request_id* is not an element of decision set, then it multicasts *request_id* to all disseminators. If *request_id* is an element of decision set, it means at least $(f + 1)$ disseminators have the *request*, since, ordering layer can decide *request_id* only when it is an element of *stable_ids* set. *Request_id* can become an element of *stable_ids* set only when at least $(f + 1)$ disseminators have the request.

Moreover, on receiving a *request_id* by any disseminator indicating that corresponding request is not available at this disseminator, then this disseminator sends a $\langle$Resend, *request_id*$\rangle$ message to a disseminator from where it has received *request_id*. In reply of this message, disseminator receives the corresponding request. Furthermore, if disseminator observes that the *request_id* is an element of decision set then it periodically sends a $\langle$Resend, *request_id*$\rangle$ message to any other disseminator. If learner is not on the disseminator's node, then on learning a *request_id*, if corresponding request is not available then it periodically sends $\langle$Resend, *request_id*$\rangle$ message to any disseminator until it receives the request.

Statements in the above two paragraphs ensure that all nonfaulty disseminators and learners will receive the request. Nonfaulty disseminator that received the client request either receives a majority of reply messages or observes that *request_id* is an element of decided set. Hence, client will receive a reply if it does not fail, because disseminator will periodically send reply to the client until it receives a reply or detects a failure.

Leader will receive same $\langle request\_id \rangle$ messages from at least $(f + 1)$ disseminators, because at least $(f + 1)$ disseminators are always nonfaulty as per assumption and all nonfaulty disseminators have the request as per the above paragraph and all disseminators that have request periodically multicasts $\langle request\_id \rangle$ to all sequencers. Hence, *request_id* will definitely become an element of *stable_id* set at the leader and then classical Paxos will order all the elements of *stable_id* set. As we already know, classical Paxos guarantees progress under aforementioned requirements, therefore, at least one nonfaulty learner will definitely learn the *request_id*. Since, all nonfaulty learners have the corresponding request as per statements of above paragraph. Therefore, all nonfaulty learners will learn the corresponding request.

Hence, we can say that under aforementioned specific requirements, HT-Paxos ensures progress.

5. Comparative Analysis

As we observe, the workload of most of the real time applications that use state-machine replication for increasing availability is increasing day by day. Therefore, requirement of high throughput is also increasing accordingly. We can increase throughput by increasing the processing power of computers and increasing the bandwidth of communication network. Every time this solution for higher throughput may not be practical for either technological or economical reasons because replacement of existing computers and communication network with higher processing power computers and higher bandwidth communication network may be a costly affair and may not be practical every time. Moreover, there may be the case that higher technology of computers and communication network may not be available every time.

Alternatively, we can adopt a more scalable and throughput efficient protocol, that is, a protocol that requires comparatively less computation at individual computers and less traffic at individual LANs. In addition, it may increase throughput by increasing more computers and more LANs, although after a certain limit, we cannot scale up the system because of coordination overload; instead, it may start reducing the throughput after certain limit. This limit depends on the protocol that we use.

Earlier versions of Paxos (as classical Paxos, fast Paxos and generalized Paxos) lack in terms of scalability and throughput, because particularly leader has more processing and bandwidth requirements. Other variants of Paxos

like ring Paxos, multiring Paxos, and S-Paxos increase the scalability and throughput by reducing the processing and bandwidth requirements, especially at the leader (Figure 3). We are going to compare the processing and bandwidth requirements among various Paxos protocols that affected system scalability and throughput.

5.1. Processing Requirements. In general, for state-machine replication protocols, processing requirements at any individual computer reduce, if computer processes less number of messages. Therefore, we require analyzing the number of incoming and outgoing messages. For the analysis, we are considering here the case of normal operations.

Moreover, we also require some processing for the transmission of the data. If any individual computing node requires higher data transmission then it requires higher processing requirement for the data transmission. We will discuss this requirement in the next bandwidth requirements section.

5.1.1. Message Analysis of HT-Paxos. Assume clients issue total number of n requests per unit time to all m disseminators then on an average; each disseminator receives n/m requests per unit time. Further, assume that each disseminator makes a batch of n/m requests per unit time and leader makes a batch of m *batch_ids* with total s sequencers.

For processing client requests of one unit time, number of messages at various sites is determined as follows.

(1) At Any Disseminator Site

$$\text{Total incoming messages} = ((n/m) + 2m).$$

Since disseminator will receive m/n requests directly from the clients, m batches from all disseminators (including self), m reply messages $\langle batch\_id \rangle$ from all disseminators (including self), and one decision message containing m *batch_ids* from the leader (since, learner is also on disseminator' site).

$$\text{Total outgoing messages} = (m + 3).$$

Since disseminator multicasts their own batch to all disseminators and learners and per batch there is one reply $\langle batch\ id \rangle$ message. Moreover, disseminator multicasts one multicast $\langle batch\ id \rangle$ message to all sequencers and sends a reply message to the client.

$$\text{Total messages at a disseminator's site} = (3m + (n/m) + 3).$$

(2) At the Leader Site

$$\text{Total incoming messages} = (m + \lfloor s/2 \rfloor).$$

Since leader receives m *batch_ids* and $\lfloor s/2 \rfloor$ phase 2b messages of classical Paxos, as leader is also a one of the acceptors of classical Paxos, so $\lfloor s/2 \rfloor + 1$ sequences (acceptors of classical Paxos) create a required majority.

$$\text{Total outgoing messages} = 2.$$

Since leader multicasts one phase 2a message to majority of sequencers (acceptors of classical Paxos). In addition, it also multicasts a decision message to all the sequencers, disseminators and learners.

$$\text{Total messages at the leader's site} = (m + \lfloor s/2 \rfloor + 2).$$

(3) At Any Sequencer Site (Other Than the Leader)

$$\text{Total incoming messages} = m + 2.$$

Since, sequencer receives m batch IDs, one phase 2a message of classical Paxos and one decision message from the leader.

$$\text{Total outgoing messages} = 1.$$

Since, sequencer sends only one phase 2b message of classical Paxos.

$$\text{Total messages at a sequencer} = m + 3.$$

(4) At Any Learner Site (without Disseminator)

$$\text{Total incoming messages} = m + 1.$$

Since learner receives m batches and one decision message from the leader.

In normal operations, there is no outgoing message.

Therefore, total messages $= m + 1$.

5.1.2. Message Analysis of Ring Paxos. Because of the processing, bottleneck may be at the leader. Therefore, total number of messages at the leader is calculated. Just like HT-Paxos, it is assumed that out of total n requests leader makes m batches of n/m requests each.

$$\text{Total incoming messages} = n + m.$$

Since leader will receive n requests directly from the clients and for m batches leader will receive m messages from the last acceptor of the ring.

$$\text{Total outgoing messages} = n + m + 1.$$

Since leader will send n reply messages to the clients, for m batches leader will ip-multicast m messages to all acceptors and learners and ip-multicast one decision message containing m *batch_ids* to all acceptors and learners.

$$\text{Total messages at the leader's site} = 2(n + m) + 1.$$

5.1.3. Message Analysis of S-Paxos. Because of the processing, bottleneck may be at the leader. Therefore, the total number of messages at the leader is calculated. Just like HT-Paxos, it is assumed that various clients issue total number of n requests per unit time and total m disseminators are there then on an average and each disseminator receives n/m requests per unit time. Moreover, it is assumed that each disseminator makes a batch of n/m requests per unit time and the leader makes a batch of m *batch_ids*.

$$\text{Total incoming messages} = (((n/m)+m+m^2)+\lfloor m/2 \rfloor + 1).$$

Leader receives n/m requests directly from the clients and m batches from all disseminators (including self). In addition, per batch it also receives m reply messages $\langle batch\_id \rangle$ from all disseminators (including self), $\lfloor m/2 \rfloor$ messages of phase 2b of classical Paxos and one decision message from itself.

$$\text{Total outgoing messages} = n/m + m + 3.$$

Leader sends n/m reply messages to the clients and per batch one multicast of reply $\langle batch\_id \rangle$ message to all replicas. In addition, it also multicasts own batch to all the replicas, one multicast of phase 2a message of classical Paxos and one multicast of decision message to all the replicas.

$$\text{Total messages at the leader's site} = (m^2 + 2(n/m) + 2m + \lfloor m/2 \rfloor + 4).$$

5.1.4. Message Analysis of Classical-Paxos. Because of the processing, bottleneck may be at the leader. Therefore, the total number of messages at the leader is calculated under batching optimization for reducing the number of messages. Just like HT-Paxos, it is assumed that out of total n requests leader makes m batches of n/m requests each. Let there be total m acceptors.

$$\text{Total incoming messages} = n + m * \lfloor m/2 \rfloor.$$

Since, leader receives n client requests as well as per batch $\lfloor m/2 \rfloor$ messages in phase 2b.

$$\text{Total outgoing messages} = n + 2m.$$

Since, leader sends n reply messages to the clients. Moreover, per batch leader multicasts a phase 2a message and one multicast of decision message.

$$\text{Total messages at the leader's site} = 2(n + m) + m * \lfloor m/2 \rfloor.$$

5.1.5. Comparative Message Analysis. As we can see in Figure 1, large number of messages in classical Paxos and in ring Paxos are because all client communications are through the leader. S-Paxos and HT-Paxos decentralize the client communication; that is, clients may approach any disseminator. Message advantage of HT-Paxos over S-Paxos is because of the fact that in S-Paxos, every disseminator is required to reply to every other disseminator. On the other hand, in proposed HT-Paxos reply goes to only one disseminator and disseminator sites are not concerned with the most of the messages of ordering layer.

It can be observed from Figure 2 that leader in HT-Paxos is very much lightweight as compared to any disseminators. It means bottleneck may not be at the leader's site in HT-Paxos (if optimized for throughput rather than fault tolerance).

In fault tolerant version of HT-Paxos, ordering layer messages also become the part of the busiest computing node (leader's site) as similar to the S-Paxos. The message advantage of this version of HT-Paxos over S-Paxos is because of the aforementioned reply mechanism of disseminators.

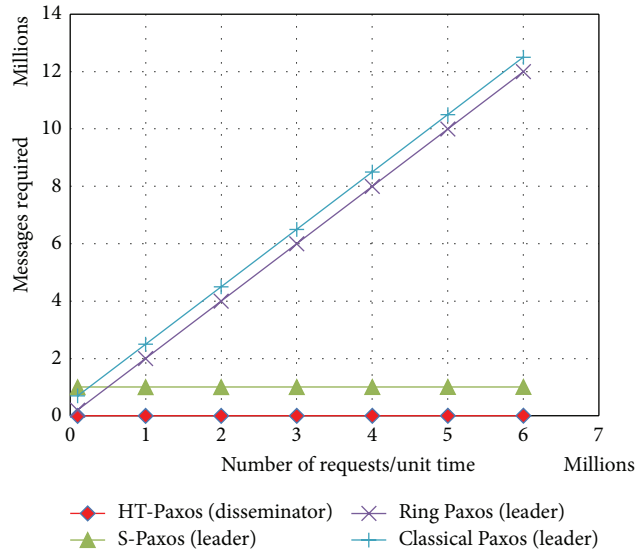

Figure 1: Comparison among mentioned variants of Paxos for the messages requirements at the busiest computing nodes, where $m = 1000$, $s = 20$.

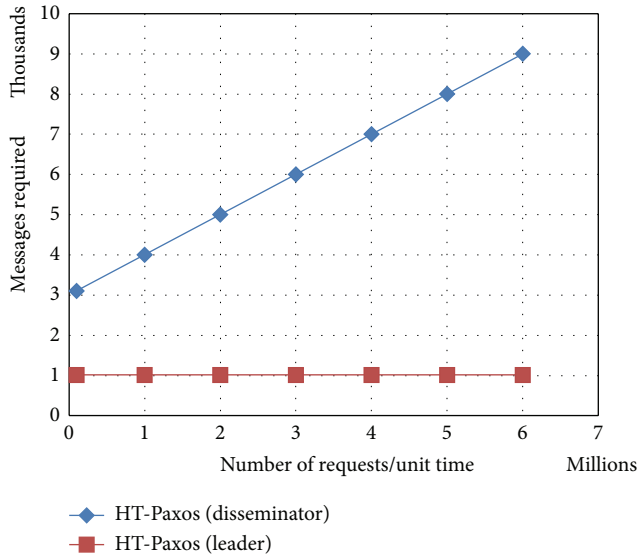

Figure 2: Comparison between any one disseminator and the leader of the HT-Paxos for the messages requirements, where $m = 1000$ and $s = 20$.

5.2. Bandwidth Requirements. Size of data and number of messages required to transmit by any computer affect the bandwidth requirements of the communication network. If any protocol requires more number of messages than due to message overhead, more data will pass through the communication network, hence requiring higher bandwidth. Bottleneck may be the bandwidth of communication network due to large data size and high number of messages. In any data center, if bandwidth is bottleneck, then there are two options either replacing the lower bandwidth LAN with higher bandwidth LAN or adopting multiple LANs of same

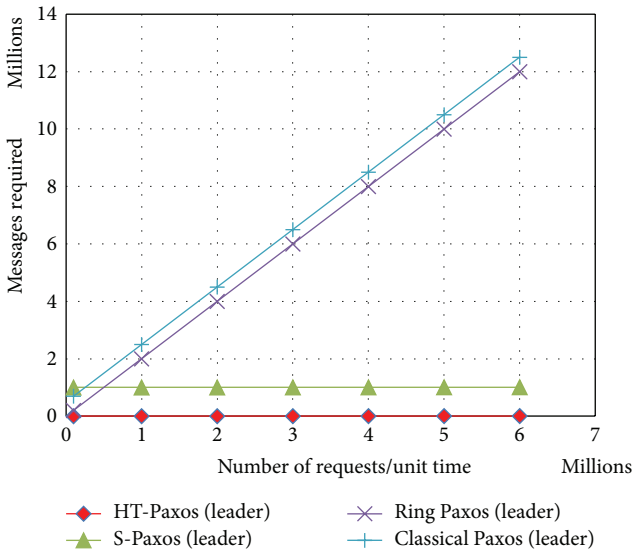

Figure 3: Comparison among mentioned variants of Paxos for the messages requirements at the busiest computing nodes, where m = 1000 and every disseminator site also has a sequencer and a learner, that is, fault tolerant version of HT-Paxos.

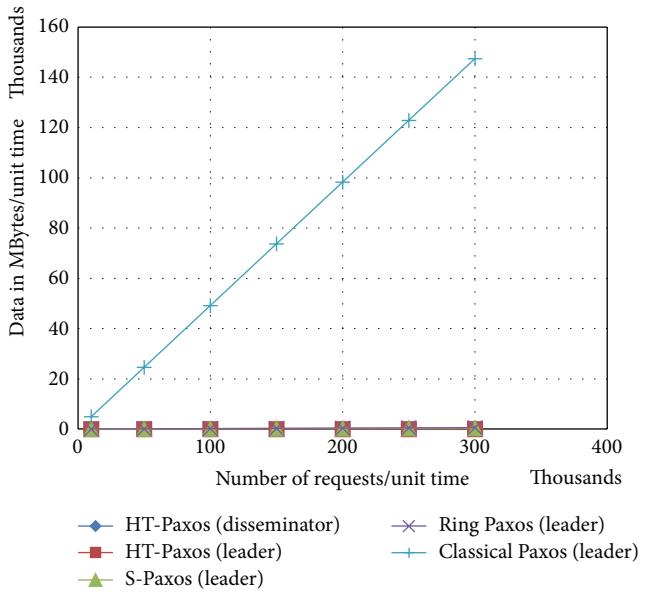

Figure 4: Comparison of bandwidth requirements at the mentioned computing nodes of the various mentioned variant of Paxos, where m = 1000, s = 20, and data size of request = 1 k byte.

bandwidth. First option may not be practical for either technological or economical reasons. As data centers do not require big cables, therefore, it is not a costly affair and hence not a big issue in any large data center.

However, if any computing node requires transmitting and receiving more data, then bottleneck may be the network subsystem of computing node that works for the transmitting and receiving of the data. Replacements of computing nodes with higher processing powers may really be a big issue, because it may be a costly affair.

Therefore, bandwidth requirements of individual computing nodes of the various variant of Paxos need to be checked. For that, same assumptions will be applicable as discussed in the previous section. Moreover, it is assumed that message overhead is 64 bytes, and *request_id*, *batch_id*, round number, and instance number are 4 bytes each.

On the basis of what incoming and outgoing messages are there, one can calculate the incoming and outgoing data per unit time.

In any clustered data center, if classical Paxos is used then leader of classical Paxos handles extremely large amount of data (as mentioned in Figure 4) just because protocol achieves consensus on request (or batch) instead of *request_id* (or *batch_id*). Other variants of Paxos for high throughput achieve consensus on *request_id* (or *batch_id*) instead of request (or batch) because, in general, *request_id* (or *batch_id*) remains very small as compared to the corresponding request (or batch).

If number of requests increases, the leader of ring Paxos handles large amount of data as compared to other high throughput Paxos (as shown in Figure 5). Major reason is that the leader handles all client communications. Moreover, in case of fewer requests, ring Paxos performs better than S-Paxos. The reason for this is the comparatively large

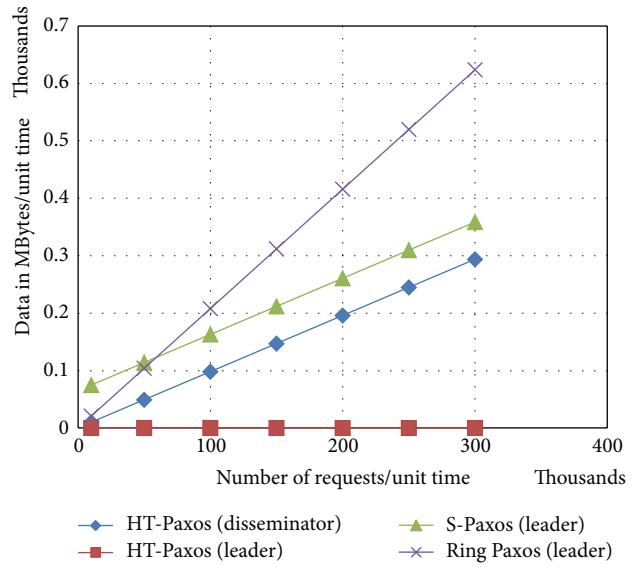

Figure 5: Comparison of bandwidth requirements at the mentioned computing nodes of the various mentioned variant of Paxos, where m = 1000, s = 20 and data size of request = 1 k byte.

number of reply messages at the disseminators. Furthermore, disseminator of HT-Paxos handles less data because of decentralized client communications like S-Paxos; in addition, it reduces the number of reply messages at the disseminators. Furthermore, leader of HT-Paxos is significantly lightweight because it handles lightweight *request_ids* or *batch_ids*.

As the data size of the client request reduces, it can be observed that the gap of S-Paxos with HT-Paxos widens as shown in Figure 6. This is because of high ratio of metadata in S-Paxos as compared to HT-Paxos. Moreover, S-Paxos

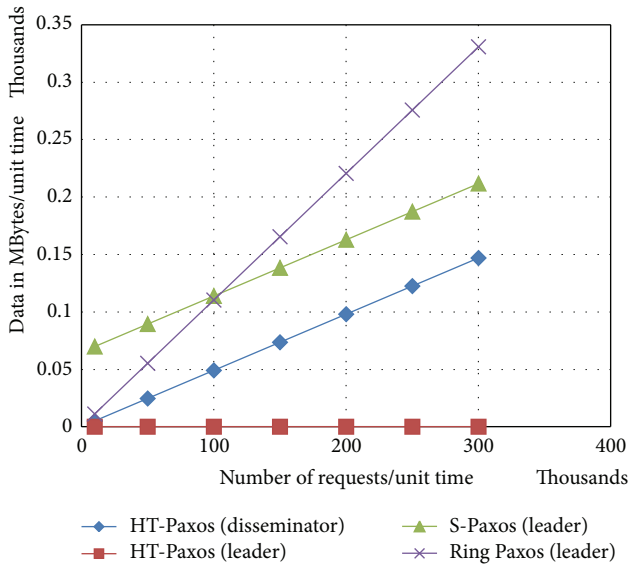

FIGURE 6: Comparison of bandwidth requirements at the mentioned computing nodes of the various mentioned variant of Paxos, where $m = 1000$, $s = 20$, and data size of request = 512 bytes.

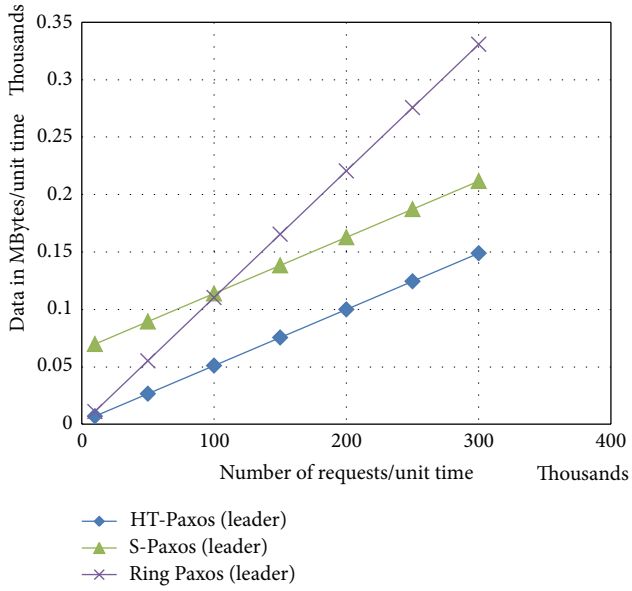

FIGURE 7: Comparison among mentioned variants of Paxos for the messages requirements at the busiest computing nodes, where $m = 1000$, every disseminator site also has a sequencer and a learner, that is, fault tolerant version of HT-Paxos, and data size of request = 512 bytes.

becomes better than ring Paxos in such case only after larger number of requests/per unit time.

In fault tolerant version of HT-Paxos, leader's site combines the dissemination and ordering layer data, but ordering layer data is too low and therefore impact in data terms at leader's site is not too much as shown in Figure 7.

5.3. Latency. HT-Paxos and S-Paxos both take six message delays for learning the client requests in the best case (if message-optimized version of classical Paxos in the ordering layer is considered). In the best case classical Paxos takes four message delays in message-optimized version and three message delays otherwise. Moreover, fast Paxos and generalized Paxos take only two message delays in the best case, while ring Paxos takes $(m + 2)$ message delays in the best case, where m represents the total number of acceptors in the ring.

5.4. Response Time. HT-Paxos takes only four message delays for responding to the client request in the best case, because a slightly optimistic approach for sending a reply to the client is chosen that is, on being sure that the request is available at any majority of disseminators, disseminator who has received the request from the client sends the corresponding reply. Under mentioned assumptions, request will definitely be executed. However, if clients want to get a reply only after the execution of requests, as in the case of S-Paxos, HT-Paxos will also take six message delays like S-Paxos. Ring Paxos takes $(m + 2)$ message delays in the best case, where m represents the total number of acceptors in the ring. In this regard classical Paxos has a comparatively good performance as it takes only four message delays.

5.5. Other Related Works. Zab [5] is a variant of the Paxos designed for the primary-backup data replication systems such as Yahoo's Zookeeper coordination service. In zookeeper, any client approaches any replica (either leader or follower) for their requests. Follower replica forwards all update requests to the primary replica for taking the services of state-machine replication protocol Zab. Zab is a centralized protocol that has one primary that disseminates the update requests to all other replicas and the leader that generally is on the same primary site works for ensuring a proper order. However, HT-Paxos is a more decentralized state-machine replication protocol that has multiple disseminators; any client for their update request may directly approach any replica that has a disseminator and after that disseminator forwards the update request to all other replicas. In case of read request client may approach any replica. Because of the centralized nature of the Zab, bottleneck may be the resources of the leader's site (or primacy's site as Zab considers both on the same site) in any large clustered data centers. Therefore, throughput and scalability will obviously be less in any large clustered data center where workload is very high.

Mencius [15] takes an alternative approach that is a moving sequencer approach [16] to prevent the leader from becoming the bottleneck. Mencius partitions the sequence of consensus protocol instances among all replicas and each replica becomes a (initial) leader of an instance in a round-robin fashion. Protocol excludes all failed replicas by adopting a reconfiguration mechanism. This protocol is a quite decentralized protocol like HT-Paxos. However, every replica failure requires a reconfiguration of the system this is not the case with HT-Paxos. Moreover, even in the case of failure free execution, leader of Mencius does the work of dissemination as well as ordering. However, in throughput-optimized version of HT-Paxos, leader is only responsible for

ordering of *request_ids* and is very much lightweight. Under a large clustered data center and heavy load environment that is the basic motivation of this paper, leader of Mencius will handle more number of messages as well as more data as compared to any disseminator or the leader. Performance of Mencius against fault-tolerant version of HT-Paxos in failure free environment may be quite comparable. However, design goal of Mencius was to provide an optimized state-machine replication protocol for WAN environment. Contrary to this HT-Paxos is for clustered environment.

LCR [17] is a high throughput state-machine replication protocol based on virtual synchrony model [18] of data replication instead of Paxos. LCR arranges replicas along a logical ring and uses vector clocks for message ordering. LCR is a high-throughput protocol, where all replicas are equally loaded, thereby utilizing all available system resources. However, latency and response times increase linearly with the number of processes in the ring. For any large clustered data center, this will be very significant. Although LCR has slightly better bandwidth efficiency, in LCR, every failure of the replica requires a view change for ensuring progress and perfect failure detection. In other words, erroneously considering a process as crashed is not tolerated which implies stronger synchrony assumptions.

State partitioning [9] is another technique that can achieve scalability. Multiring Paxos [8] uses this concept and keeps various logical groups. Each logical group has an instance of ring Paxos (in optimized version, multiple logical groups may also have a single instance of ring Paxos). Any learner may subscribe to any one or more logical groups. If a learner subscribes to multiple groups then it uses a deterministic procedure to merge messages coming from different instances of ring Paxos. However, HT-Paxos can easily adopt the concept of state partitioning by slightly changing the dissemination layer, as disseminator can multicast the request to only interested learners, while ordering layer would deliver the order to all learners (like S-Paxos). In ring Paxos or in multiring Paxos, any failure of acceptor requires a view change. Moreover, latency and response times increase linearly with the number of acceptors in the ring.

6. Conclusion and Future Work

HT-Paxos is a variant of Paxos designed for large clustered data centers that achieves significantly high throughput and scalability. It achieves all this by further offloading the leader; that is, HT-Paxos is very much decentralized protocol. As the primary focus of earlier versions of Paxos was fault tolerance and latency because of comparatively very low throughput requirement. In Paxos based protocols, the major obstacle for high throughput was bottleneck at the leader. In such systems, very soon on increasing more computing nodes fault tolerance increases rather than throughput. Practically this is highly undesirable, because massive failures could be a very rare event in the clustered data centers. Instead, it is quite more desirable in the data centers that on increasing more computing nodes, it should increase performance in terms of throughput.

Moreover, throughput may be limited because of processing power of CPU or data handling capacity of network subsystem of any computing node or bandwidth of communication networks. As commuting resources are generally very much costly as compared to data cables, in clustered data centers length of data cables required may not be too much as compared to WAN environment. Therefore, high throughput state-machine replication protocols should avoid bottleneck of CPU and network subsystems at any computing node through less computing requirements of CPU and less bandwidth requirements at any individual computing node. Proposed HT-Paxos achieves all these goals very significantly for improvement of throughput and scalability. Moreover, at the same time, latency and response times of the HT-Paxos as comparable to other high throughput state-machine replication protocols are quite less.

As future work, we plan to apply our technique to Byzantine faults and will optimize HT-Paxos for WAN.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[3] J. Baker, C. Bond, J. C. Corbett et al., "Megastore: providing scalable, highly available storage for interactive services," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pp. 223–234, January 2011.

[4] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pp. 335–350, Seattle, Wash, USA, 2006.

[5] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: high-performance broadcast for primary-backup systems," in *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN '11)*, pp. 245–256, June 2011.

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

[7] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-Paxos: offloading the leader for high throughput state machine replication," in *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems (SRDS '12)*, pp. 111–120, Irvine, Calif, USA, October 2012.

[8] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, pp. 1–12, Boston, Mass, USA, June 2012.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of the ACM SIGMOD*

International Conference on Management of Data (SIGMOD '96), pp. 173–182, Montreal, Canada, June 1996.

[10] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[11] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[12] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[13] L. Lamport, "Generalized consensus and paxos," Microsoft Research Technical Report MSR-TR-2005-33, 2005.

[14] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan et al., "Dr. Multicast: Rx for data center communication scalability," in *Proceedings of the 5th European Conference on Computer Systems*, pp. 349–362, ACM, New York, NY, USA, 2010.

[15] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 369–384, 2008.

[16] X. Défago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[17] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Transactions on Computer Systems*, vol. 28, no. 2, article 5, Article ID 1813656, 2010.

[18] K. P. Birman, "A history of the virtual synchrony replication model," in *Replication: Theory and Practice*, B. Charron-Bost, F. Pedone, and A. Schiper, Eds., vol. 5959 of *Lecture Notes in Computer Science*, chapter 6, pp. 91–120, Springer, Berlin, Germany, 2010.