Fast Mencius: Mencius with Low Commit Latency

Wei Wei*, Harry Tian Gao†, Fengyuan Xu*, Qun Li*
The College of William and Mary
*{wwei, fxu, liqun}@cs.wm.edu, †htgao@email.wm.edu

***Abstract*—Mencius is a protocol for general state machine replication that tolerates crash failures. It has high performance in wide-area networks. However, the commit latency of Mencius is limited by the slowest replica. This paper presents Fast Mencius, a crash fault-tolerant state machine replication protocol, which enhances Mencius with Active Revoke and Multi-instance Propose. Active Revoke allows the non-slow replicas to proceed without being delayed by the slowest replica, while Multi-instance Propose enables the slow replicas to have their proposals chosen by the replicated state machine. Our evaluation shows that in presence of slow replicas, Fast Mencius's commit latency is significantly lower than that of Mencius, and it also achieves high throughput.**

I. Introduction

As the reliance of our society on wide-area computing services grows, tolerating faults in these services is increasingly important. State machine replication [9] is the most general approach to implementing a highly reliable service. With this approach, a deterministic service is replicated across a set of failure-independent replicas, and the replicas consistently change their states by applying commands from an agreed sequence. Each command in the sequence is chosen by a consensus instance. State machine replication has been widely explored by previous research [3], [12], [17] and used in real systems [2], [10]. In this paper we consider the problem of building replicated state machines in wide-area networks that tolerate crash failures.

It has been proved that under pure asynchronous circumstances, no consensus protocol can ensure both safety and liveness, even when a single replica can fail [8]. However, the impossibility result can be circumvented by making weak assumptions about the synchrony of the system. The well-known Paxos protocol [12] was proposed to build replicated state machines that tolerate crash failures in an asynchronous environment, assuming the existence of an eventual leader election mechanism. The simplicity of Paxos enables it to achieve good throughput during normal execution. However, its performance is limited by the single leader. With Paxos, all the client commands need to be first forwarded to the leader. In a wide-area network where a client is not co-located with the leader, this incurs high transmission delay. Moreover, the leader in Paxos is responsible for proposing all the client commands. This one-to-many communication pattern leads to the fact that when the system is network-bound, the throughput of the system is limited by the bandwidth of the links incident upon the leader, and the available bandwidth of other links is not fully used. Also, since the leader needs to process much more messages than the other replicas, when the system is CPU-bound, the throughput of the system is limited by the leader's processing power. The variants of Paxos, like Fast Paxos [15], Generalized Paxos [14], and Hybrid Paxos [7], reduce Paxos's latency when certain conditions are met, but they still suffer from the weakness that the leader is a bottleneck of the system.

To address this problem, Mao et al. proposed Mencius [17], a multi-leader state machine replication protocol. Derived from Paxos, Mencius is designed to achieve high performance in wide-area networks. The basic idea of Mencius is to partition the sequence of consensus instances among all the replicas. By doing so, each replica proposes the received client commands using its allotted instances, and the replicas in total can service more commands. With Mencius, the network resources are more fully used, and the load of being the leader is amortized among all the replicas. Besides, clients can use their local replica to propose commands, and thus the latency for clients to receive replies is reduced.

Mencius, however, has its own weakness. As it is stated by the authors: "Mencius's commit latency is limited by the slowest server" [17], where commit latency is defined as the interval between when a replica proposes a command and when the command is committed by the replica. State machine replication requires that a replica commit a command only when it learns the command has been chosen in a consensus instance, and it has learned and committed the commands chosen in all the previous instances. With Mencius, to commit a command chosen in instance i, a replica has to wait for all the other replicas to skip, by proposing *no-op*, or to propose commands in their allotted instances smaller than i. In wide-area networks where link delays are typically large and unpredictable, it is with high probability that some slow replicas, whose transmission delay is larger than the others, exist in the system. With Mencius, the commits of commands at the fast replicas are all delayed by the slowest replica.

This paper presents Fast Mencius, a multi-leader state machine replication protocol that is derived from Mencius. With Fast Mencius, the commit latency of fast replicas is not limited by the slowest replica, while the slow replicas can still have their proposed commands chosen by the replicated state machine. To achieve this, Fast Mencius enhances Mencius with two mechanisms, Active Prepare and Multi-instance Propose. In presence of slow replicas, Active Prepare ensures that a fast replica does not need to wait for the slow ones to skip or to

This work is supported in part by US National Science Foundation grants CNS-1117412 and CAREER Award CNS-0747108.

propose commands, and Multi-instance Propose gives the slow replicas opportunities to propose their own commands. The evaluation results show that with Fast Mencius, the commit latency of the non-slow replicas is independent of the delay of the links connecting the slowest replica, and the throughput of Fast Mencius is significantly larger than that of Paxos even in presence of slow replicas.

II. System Model and Assumptions

Like Mencius, we model a system as n sites interconnected by a wide-area network. The link delays of the wide-area network are large and may have high variance. Each site contains a state machine replica and a group of clients, which communicate through links with high bandwidth and small delay. The n replicas communicate to implement a crash fault-tolerant replicated state machine. Replicas can fail by crashing, and may later recover. They have access to stable storage to record their states, which will be used after they recover from a failure. The system is asynchronous, with no bound on message transmission delay. As a crash fault-tolerant protocol, Fast Mencius requires $2f + 1$ replicas to handle up to f concurrent faulty replicas, with a quorum size of $f + 1$.

To use the service, clients send commands to their local replica. The replicas establish a total order for all the commands by running a sequence of consensus instances. Each command is chosen in one instance. Replicas commit the commands in the decided order. Once a command is committed, the reply is sent from the local replica back to the client generating the command.

Fast Mencius shares the same assumption with Mencius that the communication channels are FIFO, and messages between two correct replicas are eventually delivered, but there is no upper-bound on message delivery time. In practice, the communication channels can be implemented by UDP with retransmission and flow control or TCP. To circumvent the FLP impossibility result [8], we assume that each replica has access to a failure detector $\diamondsuit\mathbb{P}$, which guarantees eventually all faulty servers and only faulty servers are suspected [4]. Like Mencius, the failure detector is only used by Fast Mencius to guarantee liveness. Fast Mencius is safe even if the failure detector makes an unlimited number of mistakes.

III. Mencius revisited

Before we describe how Fast Mencius is derived from Mencius, we first briefly review Mencius.

As a multi-leader state machine replication protocol, Mencius [17] is derived directly from Paxos [12], [13]. To order the commands, Mencius runs an unbounded sequence of simple consensus instances. Let *no-op* be a command that leaves the replica state unchanged and generates no response. In each simple consensus instance, only one replica, which is called the coordinator, can propose any command, while the other replicas can only propose *no-op*. Simple consensus allows a coordinator to skip its instance with only one communication step: other replicas learn that *no-op* has been chosen in this instance once they receive a $Skip$ message, which is a $Propose$ message that proposes *no-op*, from the coordinator. The sequence of simple consensus instances is partitioned among all the replicas. Each replica is the coordinator (default leader) of an unbounded number of instances. The simplest way to assign instances to replicas is in a round-robin fashion: the i^{th} replica coordinates instance $cn + i$, where $c \in \{0, 1, 2, ...\}$ and n is the number of replicas.

In Mencius, simple consensus is implemented by Coordinated Paxos [17]. Coordinated Paxos differs from Paxos in that each Coordinated Paxos instance starts from the state in which the coordinator had run the Prepare Phase of Paxos for some initial round r. This is safe because in each instance all the replicas agree that the coordinator is the default leader.

Mencius is built upon the following rules.

Rule 1: Each replica p maintains I_p, the sequence number of the next available simple consensus instance it coordinates. I_p is also called the *index* of p. Upon receiving a command from a local client, p proposes the command in instance I_p with round r, and updates I_p to the next instance it coordinates. This rule ensures that a replica proposes a command immediately after receiving it from a client.

Rule 2: If replica p receives a $Propose$ message in instance i and $i > I_p$, before accepting the proposal, p updates I_p such that the new index $I'_p = min\{i' : p\, coordinates\, instance\, i' \wedge i' > i\}$. p also executes skip actions, by proposing *no-op*, for each of the instances in range $[I_p, I'_p)$ it coordinates. With this rule, the replicas proposing commands less frequently, e.g., fewer clients in their sites generate commands, skip their instances.

Rule 3: By accessing the failure detector $\diamondsuit\mathbb{P}$, if replica p suspects replica q has failed, and C_q is the smallest instance that is coordinated by q and not learned by p, then p revokes all the instances in range $[C_q, I_p]$ that q coordinates. Note that revocation is done by running both the Prepare Phase and Propose Phase of Paxos, as shown in Figure 1. With this rule, a crashed replica cannot prevent other replicas from committing their learned commands.

Rule 4: If replica p proposes a command $v \neq$ *no-op* in instance i, and p learns that *no-op* is eventually chosen in this instance, which means p has been falsely suspected and instance i has been revoked by another replica, then p proposes v again in a higher instance it coordinates. This rule allows a falsely suspected replica to propose a command multiple times, until false suspicions eventually cease.

These four rules guarantee that any client command sent to a correct replica is eventually committed, and it takes the minimal two communication steps ($Propose$ and $Accept$) for a proposing replica to learn the outcome of a consensus instance. However, the message complexity depends on the rates at which the replicas propose client commands: when the replicas propose commands at different rates, they need to execute many skip actions, by sending $Skip$ messages. Mencius solves this problem with two optimizations.

Optimization 1: If replica p receives a $Propose$ message from q in instance i and $i > I_p$, p uses the $Accept$ message that replies the $Propose$ to promise q that p will not propose

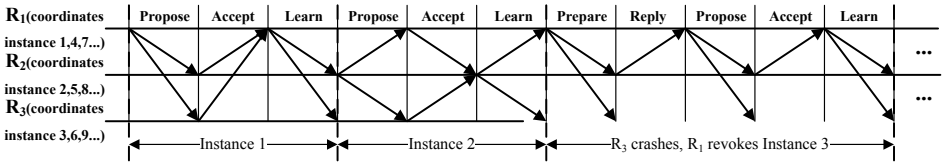

Fig. 1. Message pattern of Mencius

any client command in instances smaller than i in the future, i.e., *no-op* has been chosen in these instances. In this case, p does not need to send $Skip$ messages to q. p does not send $Skip$ messages to other replicas either. Instead, for each of these replicas, p waits for a future $Propose$ message sent to that replica, to promise not to propose any client command in smaller instances. This optimization is valid since Mencius assumes FIFO channel: before a replica receives an $Accept$ or $Propose$ message from p, it must have received all the previous $Propose$ messages from p, so it can safely infer the instances p skips.

Optimization 2: A replica p propagates $Skip$ messages to another replica q if the total number of delayed $Skip$ messages to q is larger than some constant α, or the messages have been deferred for more than some time t. This rule is used to limit the delay of the propagation of $Skip$ messages due to Optimization 1.

IV. Fast Mencius

In wide-area networks where the link delays are influenced by the network traffic and are highly unpredictable, it is with high probability that the connection speed of a replica drops below the other replicas, and becomes a slow one. Mencius has high performance in wide-area networks. However, its commit latency is limited by the slowest replica. We explain this through an example. Considering the scenario in Figure 2, the system consists of 3 replicas, among which A is a slow replica. Assume at time 0 each replica proposes its first command: A proposes in instance 1; B proposes in instance 2; C proposes in instance 3. With Mencius, at 50 ms B and C receive the $Propose$ message from each other, and reply with an $Accept$ message. At 100 ms B and C have collected $Accept$s from a majority of replicas (including the one from themselves), and learn that their proposed command has been chosen. However, at this time, B cannot commit its command chosen in instance 2, because it does not know the outcome of instance 1. Nor can C commit its command chosen in instance 3. At 500 ms, B and C receive the $Propose$ from A, and reply with $Accept$. At 1000 ms, A learns its command is chosen in instance 1, and broadcasts the $Learn$ message. A also commits this command. At 1500 ms, B and C receive the $Learn$ message from A and learn the outcome of instance 1. Only at this time can B and C commit their proposed command. As a result, although it only takes B and C 100 ms to learn their command has been chosen, their commit latency is 1500 ms, three one-way delays of the links incident upon A. Ironically, the slow replica A has a smaller commit latency of 1000 ms.

State machine replication requires that a replica commit a command only when it learns the command has been chosen

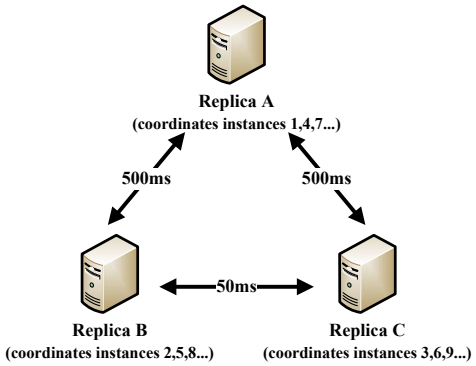

Fig. 2. A system with a slow replica

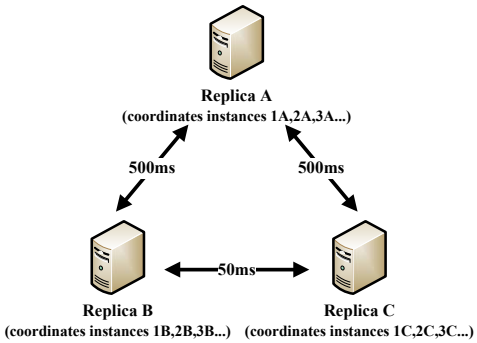

Fig. 3. Instance assignment scheme of Fast Mencius

in a consensus instance, and it has learned and committed the commands chosen in all the previous instances. With Mencius, to commit a command v chosen in instance i, a replica p has to wait for all the other replicas to propose commands, or to skip, by proposing *no-op*, in their allotted instances smaller than i. Otherwise, there will be gaps in the command sequence learned by p, and p cannot commit v.

Fast Mencius addresses this problem with two mechanisms, Active Revoke and Multi-instance Propose. With Active Revoke, fast replicas can proceed without being delayed by the slowest replica, while Multi-instance Propose enables slow replicas to have their proposed commands chosen by the replicated state machine, even in presence of Active Revoke and false failure suspicions.

A. Active Revoke

The intuition behind this mechanism is that, instead of requiring fast replicas wait for the messages, such as $Propose$, $Skip$, or $Learn$, from slow replicas to commit their commands, we allow them to actively revoke the instances coordinated by slow replicas, as long as their commit of client commands has been delayed for a sufficiently long time.

Different from Mencius, an instance number in Fast Men-

cius is of the form $counter \parallel id$, where id is the identifer of the coordinator of this instance. The instance numbers are ordered primarily by $counter$. For the instance numbers with the same $counter$, they are ordered lexicographically by id. For example, the system in Figure 3 has three replicas A, B, and C. Then the instance numbers will be ordered as $1A < 1B < 1C < 2A < 2B < 2C$.... This assignment scheme enables dynamically adding and removing replicas. Besides, given one instance number, all the replicas know unambiguously who is the coordinator of this instance.

Active Revoke is triggered if Condition 1 is met:

Condition 1 The commit of one of replica p's proposed commands has been deferred by an unlearned instance for more than some time τ, and the failure detector indicates the coordinator of the unlearned instance is alive.

If this condition is met, we say p is delayed by the coordinator of the unlearned instance. To expedite its advancement, p will revoke the instances coordinated by the slow replica that prevent it from committing its commands, by running both the Prepare Phase and Propose Phase of Paxos. Note that in Mencius, revocation is done only when a replica suspects another has failed. In Fast Mencius, revocation is also used to help the fast replicas speed up their commits. The problem arises from this design is that if multiple replicas concurrently revoke the instances coordinated by a slow replica, there will be competing $Prepare$ messages: a $Prepare$ with a higher round number will suppress a $Prepare$ with a lower round number. As a result, only the sender of the $Prepare$ with the highest round number can finish the revocation process, and the work done by other replicas is wasted. In fact, when multiple replicas are delayed by a slow replica, after one of them finishes revocation, the others can learn the outcome of the revocation with the broadcasted $Learn$ message. Therefore, the number of concurrently revoking replicas should be limited to save resources. This is achieved through the use of $Help$ messages in our design.

$Help$ messages are small status-checkers the replicas use to inquire other replicas about a particular piece of missing information, and determine the appropriateness of sending the following $Prepare$ message. After a replica p initiates Active Revoke, it first sends a $Help$ message to all the other replicas, which contains the instance numbers of the consensus instances it wants to revoke. Upon receiving the $Help$ message, another replica q replies with an Ack message, which includes the following information.

1. If q has learned the outcomes of some of the queried instances, then Ack includes the learned commands.

2. If q is revoking some of the queried instances, which means it has sent the $Prepare$ message in these instances, then Ack includes these instance numbers.

p collects the Acks from $\lfloor \frac{n-1}{2} \rfloor$ other replicas. Through these replies, p learns the commands chosen in some queried instances directly if they are in the Acks. p also knows which queried instances are currently being revoked by other replicas, and p will wait for these replicas to finish revocation and broadcast the results. Note that since p only waits for the Acks from a minority of replicas, it is rational for p to assume that the senders of these Acks are the relatively fast ones, so it can rely on them to do revocation. p revokes only the queried instances whose results are still unknown and no Ack indicates they are currently being revoked.

1) Protocol: The full Active Revoke mechanism is as follows.

Help. If Condition 1 is met, then p initiates Active Revoke. Assume the coordinator of the unlearned instance is q. Let C_q be the smallest instance coordinated by q and not learned by p, and let I_p be p's index, i.e., the next available instance coordinated by p. p sends $Help(C_q, C'_q)$ to the other replicas, where $C'_q = max\{i : q\ coordinates\ instance\ i \wedge i < I_p\}$.

Upon receiving the $Help$ message, another replica r composes a reply as $Ack(instance\ entries)$. The message may include multiple instance entries for the instances within the range $[C_q, C'_q]$ that q coordinates. If r has learned the results of some of these instances, then the corresponding entry is $\langle i, \text{`}l\text{'}, v\rangle$, where $i$ is the instance number, and $v$ is the learned command. Else if $r$ is revoking some of these instances, then the corresponding entry is $\langle i, \text{`}r\text{'}\rangle$. Otherwise, Ack is empty.

Revoke. p waits for the Acks from $\lfloor \frac{n-1}{2} \rfloor$ other replicas. Assuming set S consists of the sequence numbers of the queried instances whose results are still unknown and no Ack indicates they are currently being revoked, p starts revoking the instances in S by running both the Prepare Phase and Propose Phase of Paxos, as shown in Figure 4, with an optimization. The optimization is, if a replica already accepts a command that is not *no-op* in an instance, and it starts revoking this instance, then it broadcasts a special $Prepare$ message. Another replica's $Reply$ to this $Prepare$ does not convey the command it has accepted in this instance, if any. This optimization reduces the overhead of revocation in Fast Mencius. It is valid because by the definition of simple consensus, only the coordinator can propose any command in an instance, and the other replicas can only propose *no-op*. Therefore, if any replica accepts a command that is not *no-op* in an instance, it must have been proposed by the coordinator, and there is no need to let the $Reply$ contain the command if the revoking replica already knows about it.

During Active Revoke, if p learns the results of all the queried instances, either from q or from other replicas, then p stops doing Active Revoke immediately. Also, p performs Active Revoke for each unlearned instance coordinated by q only once.

2) Example: Consider the scenario in Figure 3. Assume at time 0 each replica proposes its first command: A proposes in instance *1A*; B proposes in instance *1B*; C proposes in instance *1C*. At 100 ms B and C learn that their proposed command has been chosen. However, they cannot commit, because they don't known the result of instance $1A$. Let the timeout threshold τ be 100 ms. Then at 200 ms both B and C initiate Active Revoke by broadcasting the $Help$ message. At 300 ms B and C receive the Ack from each other. Since neither of them knows the outcome of instance $1A$ or is currently revoking $1A$, both B and C decide to revoke $1A$

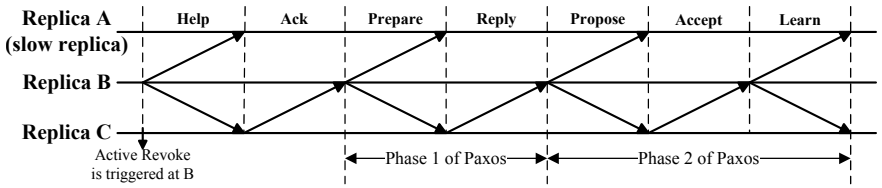

Fig. 4. Message flow of Active Revoke

and broadcast the $Prepare$ message. Note that with Paxos, the round numbers are partitioned among the replicas, e.g., in a round-robin fashion, so two replicas will never send $Prepare$ messages with the same round number. Assuming the $Prepare$ sent by C has a higher round number than the one sent by B, C's $Prepare$ will suppress B's $Prepare$, i.e., B will respond to C's $Prepare$, while C will not respond to B's $Prepare$. At 400 ms C receives the $Reply$ from B. Since neither B nor C has accepted a command in instance $1A$, C proposes *no-op* in instance $1A$. At 500 ms C receives the $Accept$ from B, and learns that *no-op* is chosen in instance $1A$. C sends the $Learn$ message to all the replicas and commits the command it proposes in instance $1C$. At 550 ms B receives the $Learn$ message from C, and learns the result of instance $1A$, so B can commit the command it proposes in instance $1B$. As a result, Active Revoke reduces the commit latency of B and C from 1500 ms to 550 ms and 500 ms, respectively. This improvement is achieved because Active Revoke enables the fast replicas to advance without learning from the slow replicas, and thus their commit latency only depends on the delay of the links connecting themselves, instead of the delay of the links incident upon the slowest replica.

B. Multi-instance Propose

Active Revoke speeds up the advancement of the fast replicas, by allowing the fast replicas to revoke the instances coordinated by the slow replicas. In the worst case, however, a slow replica may never be able to commit its proposed commands. We explain this using the scenario in Figure 3.

Assuming at time 0 each replica proposes its first command, replica A proposes a command v. With Active Revoke, replica B and C broadcast their $Prepare$ message at 300 ms. When A's $Propose$ message is delivered at B and C at 500 ms, they will not accept this proposal, because they have responded to a $Prepare$ with a higher round number. As a result, in instance *1A*, A cannot collect $Accept$s from a majority of replicas. At 1000 ms, A receives the $Learn$ message from C, which informs that *no-op* has been chosen in instance $1A$. Then A knows that instance $1A$ has been revoked by C. Following Rule 4 of Mencius, A proposes v again in the next available instance $2A$. However, assuming at this time B or C also proposes a new command, the commit of this command will be delayed by A, which triggers Active Revoke again and results in instance $2A$ being revoked. In the worst case, this situation may repeat an unbounded number of times, and A can never commit v.

Besides Active Revoke, the instances coordinated by slow replicas may also be revoked because of false failure suspicions. Since we assume the failure detector is unreliable, it may make mistakes. Compared with fast replicas, the slow replicas are more likely to be falsely suspected of having failed. When false suspicions happen, following Rule 3 of Mencius, the suspecting replicas will revoke the instances coordinated by the suspected slow replicas, which prevents the commands proposed by the slow replicas from being chosen and committed. Mencius's assumption on the failure detector is that false suspicions will eventually cease. However, this may take an unbounded length of time. Therefore, the slow replicas will suffer from large and unpredictable commit latency if they are falsely suspected.

To ensure the progress of Fast Mencius, i.e., any client command sent to a correct replica is eventually committed, and also to limit the commit latency of the slow replicas in presence of false suspicions, the slow replicas should be given opportunities to have their proposals chosen by the replicated state machine. This is achieved through our Multi-instance Propose mechanism. The intuition behind this mechanism is that instead of letting a slow replica, whose instances are being revoked by others, propose its command in one instance at a time, we let it propose its command in multiple instances simultaneously. In this way, even if other replicas revoke some of these instances, the slow replica can still commit its command if it is chosen in at least one of these instances.

1) Protocol: Multi-instance Propose is triggered if the following condition is met:

Condition 2 None of γ consecutive commands proposed by replica p is chosen by the replicated state machine.

A replica learns that its proposed command is not chosen, by receiving a $Learn$ message from another replica that informs *no-op* has been chosen in the instance in which it proposed the command. If replica p finds that none of its γ consecutively proposed commands is chosen, it can deduce that it is slow relative to others, and other replicas are revoking its instances. Under this circumstance, p initiates Multi-instance Propose to ensure that its commands can be successfully committed.

MultiPropose. With Multi-instance Propose, p proposes a command v in a block of consensus instances simultaneously, by broadcasting $MultiPropose(I_1,\ I_2,\ r,\ v)$. This message means that p proposes v in all the instances within the range $[I_1, I_2]$ that it coordinates. Since p is the default leader of these instances, the round number r is 0. To determine I_1 and I_2, let C_p be the smallest instance among the γ revoked instances coordinated by p, and I_p be p's index. Then $I_1 = I_p$, and the number of instances coordinated by p within the rage $[I_1, I_2]$ is equal to the number of instances coordinated by p within the range $[C_p, I_p)$. For example, if $C_p = 7p$, and

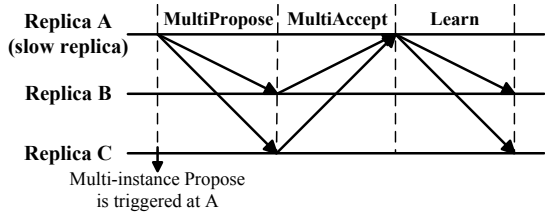

Fig. 5. Multi-instance Propose

$I_p = 10p$, then $I_1 = 10p$, and $I_2 = 12p$. After sending out the $MultiPropose$ message, p updates I_p to the next available instance it coordinates, which is $13p$ in this case. Following Optimization 1 of Mencius, p also uses this $MultiPropose$ message to promise not to propose any client command in instances smaller than I_1 in the future. That is, another replica learns that the outcomes of all the instances smaller than I_1 and coordinated by p, in which it has not received any proposal from p, are *no-op*, immediately after it receives the $MultiPropose$ message.

MultiAccept. After another replica q receives the $MultiPropose$ message from p, if its index I_q is smaller than I_1, q updates I_q such that its new index $I'_q = min\{i : q \text{ coordinates } i \wedge i > I_1\}$. This is the application of Rule 2 of Mencius in our Multi-instance context. q also checks in which instances within the range $[I_1, I_2]$ that p coordinates it can accept v, i.e., it has not responded to a $Prepare$ message with a higher round number in these instances. If q finds that it can accept v in at least one instance coordinated by p within the range $[I_1, I_2]$, then q composes a $MultiAccept$ message, which includes the sequence numbers of all the instances in which it can accept v, and sends the message back to p. Following Optimization 1 of Mencius, q also uses this $MultiAccept$ to promise p that it will not propose any client command in instances smaller than I_1 in the future.

Learn. p waits for the $MultiAccept$s from a majority of replicas, and broadcasts $Learn(i, v)$, where i is the smallest instance that appears in all the $MultiAccept$s from a majority of replicas. Note that there is at least one instance that appears in all the $MultiAccept$s received by p, which is instance I_2. Here is an example. Let the number of replicas be 5, with a quorum size of 3. Assume $I_1 = 10p$, $I_2 = 12p$, and the first 3 $MultiAccept$s received by p are $MultiAccept(10p, 11p, 12p)$, $MultiAccept(11p, 12p)$, $MultiAccept(12p)$. Then p broadcasts $Learn(12p, v)$. This $Learn$ message does not mean that v will be definitely committed at instance $12p$, because the results of instances $10p$ and $11p$ are still unknown. After sending out $Learn(12p, v)$, if p receives $MultiAccept(11p, 12p)$ from another replica, then it broadcasts $Learn(11p, v)$. Otherwise, if p receives two $MultiAccept(10p, 11p, 12p)$ messages, then it broadcasts $Learn(10p, v)$.

Commit. All the replicas will commit v at the smallest possible instance. This means that a replica can make the decision to commit v at instance i, only after it learns *no-op* has been chosen in all the instances smaller than i in the instance block p uses to propose v. In this case the replica will commit *no-op* at all the other instances in the instance block. Following the previous example, a replica decides to commit v at instance $12p$ only after it receives $Learn(10p, \textit{no-op})$, $Learn(11p, \textit{no-op})$, and $Learn(12p, v)$; it decides to commit v at instance $11p$ only after it receives $Learn(10p, \textit{no-op})$, and $Learn(11p, v)$; it decides to commit v at instance $10p$ immediately after it receives $Learn(10p, v)$. Note that the $Learn$ messages that choose *no-op* come from the replicas who are revoking instances coordinated by p. The message flow of Multi-instance Propose is shown in Figure 5.

It is possible that all the instances in the instance block p uses to propose v are revoked by other replicas. If p receives the $Learn$ messages from other replicas that indicate *no-op* has been chosen in all these instances, then p doubles the size of the instance block and proposes v again. By increasing the size of the instance block exponentially, p can quickly get an opportunity to have its proposal chosen.

A replica initiates Multi-instance Propose once for only one command, i.e., p proposes v using Multi-instance Propose, but it proposes the commands after v still following Mencius, as long as Multi-instance Propose is not triggered again. The reasoning is that after successfully proposing a command with Multi-instance Propose, a slow replica catches up with the advancement of the other replicas, and others' revocations will not prevent the commands proposed by the slow replica from being chosen, assuming the slow replica's connection speed is not dropping even more. Therefore, Multi-instance Propose is designed to provide a one-time boost to the slow replica, and it will go back to the normal mode of operation afterwards. When a replica is in the Multi-instance Propose state, it is not allowed to start Active Revoke. This is rational because being in the Multi-instance Propose state means this replica is relatively slow, and it should not revoke the instances coordinated by others. The replica is allowed to initiate Active Revoke only after it decides at which instance to commit the command proposed with its $MultiPropose$ message, i.e., when it quits the Multi-instance Propose state.

After a replica q receives the $MultiPropose$ message from p, and before it decides at which instance to commit v, it does not update its index following Rule 2 of Mencius when it receives subsequent $Propose$s from p. Optimization 1 of Mencius does not apply here either: q does not use the $Accept$ that replies a subsequent $Propose$ from p to promise not to propose any client command in a smaller instance. To let p know that this $Accept$ has a different semantics, q adds a flag in the message body. This design gives p priority to propose v: the other replicas need to revoke all the instances in the instance block step by step such as to invalidate the $MultiPropose$ message from p.

2) Example: Consider the system in Figure 3. Assuming each replica proposes a command every 10 ms, replicas A, B, and C propose their 1st command at time 0, the 2nd command at 10 ms, and so on.

At 100 ms B and C learn that their 1st command has been chosen in instance $1B$ and $1C$, respectively. However, they

cannot commit because the result of instance $1A$ is unknown. Assume the threshold τ used to trigger Active Revoke is 100 ms, and the threshold γ to trigger Multi-instance Propose is 10. Then at 200 ms B and C initiate Active Revoke by broadcasting $Help(1A, 22A)$. At 500 ms C finishes revocation and broadcasts the $Learn$ message for instances $1A$, ..., $22A$, informing that *no-op* has been chosen in these instances. The $Learn$ message is delivered at A at 1000 ms, which triggers Multi-instance Propose at A, since more than 10 consecutive commands proposed by A are not chosen. As the smallest revoked instance is $1A$, and A's current index is $102A$, A sets the size of the instance block to 101 and broadcasts $MultiPropose(102A, 202A, 0, v)$ to propose a command v at 1010 ms. The $MultiPropose$ message is delivered at B and C at 1510 ms. At this time B and C have revoked all the instances coordinated by A within the range $[1A, 132A]$, following Active Revoke. Therefore, B and C accept v in all the instances coordinated by A within the range $[133A, 202A]$, and their $MultiAccept$s are delivered at A at 2010 ms, when A learns that v has been chosen in these instances. A receives the $Learn$ message from C at 1880 ms that indicates *no-op* has been chosen in all the instances it coordinates within the range $[89A, 110A]$, and the $Learn$ message from C at 2100 ms that informs *no-op* has been chosen in all the instances it coordinates within the range $[111A, 132A]$. Then A can commit v at instance $133A$. All the other instances in the instance block A uses to propose v are considered *no-op*. As a result, the commit latency for v is 1090 ms, slightly larger than a round-trip delay between A and other replicas.

The commands A proposes after v will not be influenced by other replicas' revocations. Following the previous example, C learns it should commit v at instance $133A$ when it finishes revoking the instances coordinated by A within the range $[133A, 154A]$ at 1820 ms, and B learns the same result when it receives the $Learn$ message from C at 1870 ms. After that, when they receive a new $Propose$ message from A, they will follow Rule 2 of Mencius to make their indices match the instance number of the $Propose$, and their future revocations will not prevent the commands proposed by A from being chosen, as long as the network conditions stay the same. This is because when they send out $Prepare$ messages to revoke some of A's instances, they already received the proposals from A for these instances. As a result, they will propose A's proposals in these instances.

V. Discussion

A. Correctness of Fast Mencius

To revoke the instances of slow replicas, Active Revoke uses the Prepare and Propose phases of Paxos, whose correctness has already been proved [12]. The $Help$ messages are status-checkers that determine when revocation should be triggered, and they do not interfere with the revocation mechanism in each consensus instance. Besides, a replica is allowed to perform Active Revoke for each unlearned instance at most once. Thus, we avoid the liveness problem faced by Paxos when it has multiple revoking replicas, in which different replicas keep issuing revocation with increasing round numbers.

The correctness of Multi-instance Propose is guaranteed by ensuring that all the unfailed replicas will make the same decision at which instance to commit the command proposed by Multi-instance Propose. Fast Mencius runs an unbounded sequence of simple consensus instances. Each instance is implemented by Coordinated Paxos [17], which is the same with Paxos except for a different starting state, since in each simple consensus instance the coordinator is the default leader. Paxos ensures that in each consensus instance all the correct replicas learn the same result, so for each instance in the instance block used by a slow replica to propose a command v with Multi-instance Propose, all the correct replicas receive the same $Learn$ message. Also, a replica cannot commit a command unless it learns the commands chosen in all the previous instances. Therefore, all the unfailed replicas will make the same decision at which instance to commit v.

B. Setting parameters

Our Active Revoke and Multi-instance Propose mechanisms are triggered by two conditions using parameters τ and γ, respectively. Here we discuss how to set them.

τ determines when a replica delayed by a slow one should start Active Revoke. In Mencius, the maximum extra latency caused by delayed commit, which happens when there are concurrent $Propose$s, is a round trip delay [17]. Therefore, the minimal value of τ is a round trip delay between the non-slow replicas. A replica can estimate the round trip delay between itself and other non-slow replicas by measuring the time interval between when it sends out a $Propose$ and when it collects $Accepts$ from a majority of replicas. Assuming this delay is d, τ can be set to d. Using a larger τ reduces the number of times Active Revoke is triggered, while it increases the commit latency. In the evaluation we set τ to a round trip delay between the non-slow replicas.

γ determines when a slow replica, whose coordinated instances are revoked by others, should start Multi-instance Propose. Since being revoked in a sequence of consecutive proposed instances happens only when a replica is slow relative to others, γ can be set to the number of commands a replica proposes within a short time period. For example, in our evaluation we set γ to the number of commands a replica proposes within 100 ms.

VI. Evaluation

A. Experimental Setup

We implemented Fast Mencius using BFTSim [19], a simulation framework for state machine replication protocols. BFTSim couples a high-level protocol specification language and execution system based on P2 [16] with a network simulator built upon ns-2. BFTSim was originally proposed to implement and compare the Byzantine fault-tolerant (BFT) protocols, including PBFT [3], Q/U [1], and Zyzzyva [11], and it faithfully predicts the performance of the native protocol

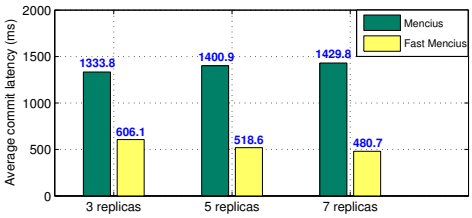

Fig. 6. Average commit latency of all the replicas

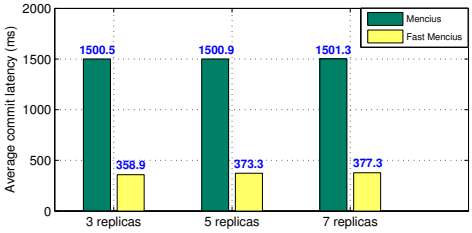

Fig. 7. Average commit latency of non-slow replicas

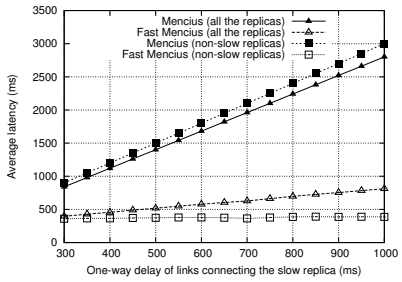

Fig. 8. Average commit latency

implementations [19]. Note that the functions needed to implement BFT protocols are a superset of the functions needed to implement the crash fault-tolerant consensus protocols, since the latter do not use the crypto operations required by the former. Therefore, BFTSim can also be used to implement and compare the crash fault-tolerant protocols.

For comparison, we also implemented Paxos and Mencius with BFTSim. To further validate the fidelity of BFTSim, we measured the throughput and latency of our Paxos and Mencius implementations under a range of network conditions, and reproduced the published results. For example, the evaluation in the original paper shows that with a three-replica clique topology, when the payload size of each command is 4000 bytes and the total bandwidth is 99 Mbps, the peak throughput of Mencius is 1550 operations per sec (ops), and the peak throughput of Paxos is 540 ops [17]. With our implementations and under the same network condition, the peak throughput of Mencius and Paxos is 1550 ops and 550 ops, respectively.

We simulated a star network topology, where the replicas are connected to each other via a hub node. Each link between a replica and the hub node is a duplex link, with a bandwidth of 20 Mbps. The one-way delay between a non-slow replica and the hub node is 25 ms. This gives an RTT of 100 ms between any pair of non-slow replicas. The link between the slow replica and the hub node has a much larger delay.

B. Experimental Results

Commit latency. As stated in the original paper: "Mencius's commit latency is limited by the slowest server." To get this result, we set the one-way delay between the slow replica and the hub node to 475 ms, which gives an RTT of 1000 ms between the slow replica and any non-slow replica. Each replica proposes commands at a rate of 100 ops. The size of payload in each command is 5 bytes. By varying the number of replicas, we got the average commit latency of Mencius. For comparison, we set $\tau = 100\ ms$, $\gamma = 10$, and got the average commit latency of Fast Mencius, as shown in Figure 6. The results illustrate that the commit latency of Fast Mencius is significantly smaller than that of Mencius. With the increase of the number of replicas, the average commit latency of Mencius increases, while the average commit latency of Fast Mencius decreases. The reason is shown in Figure 7. With Mencius, the commit latency of the non-slow replicas is about 3 times of the delay of the links connecting the slow replica, while the commit latency of the slow replica is 2 times of this delay. With Fast Mencius, the non-slow replicas have a much lower commit latency, and the commit latency of the slow replica is still about 2 times of the delay of the links connecting itself.

We got the commit latency of Mencius and Fast Mencius by varying the delay of the links connecting the slow replica, from 300 ms to 1000 ms with an interval of 50 ms. The system consists of 5 replicas. Figure 8 shows the average commit latency of all the replicas and that of the non-slow replicas. The results show that when we raise the delay of the links connecting the slow replica, the commit latency of Mencius increases considerably faster than Fast Mencius. Moreover, with Fast Mencius, the commit latency of the non-slow replicas is not influenced by the slow replica. We also measured the commit latency when a non-slow replica fails. The commit latency of the remaining replicas is not influenced. This is because both Active Revoke and Multi-instance Propose only require the participation of $f + 1$ replicas.

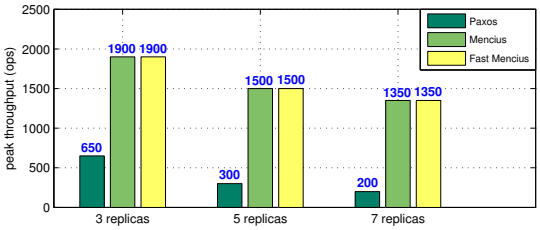

Fig. 9. Peak throughput without any slow replica

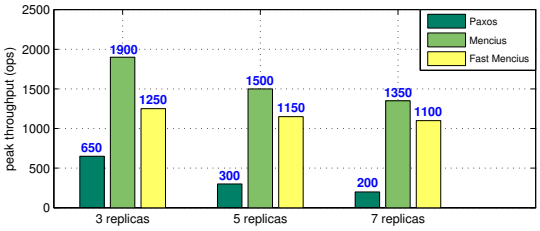

Fig. 10. Peak throughput with a slow replica

Throughput. When there is no slow replica in the system, Fast Mencius behaves the same as Mencius. Figure 9 compares the peak throughput of Paxos, which has high throughput due to its simplicity, with that of Fast Mencius, when the delay between every replica and the hub node is 25 ms, and

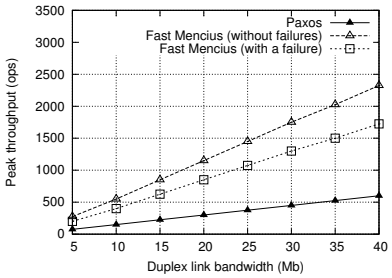

Fig. 11. Peak throughput with a slow replica

the bandwidth of each duplex link is 20 Mb. The size of payload in each command is 2000 bytes. The results show that, without any slow replica, the peak throughput of Fast Mencius is about n times of that of Paxos, where n is the number of replicas. The reason is that Fast Mencius, derived from Mencius, utilizes available bandwidth in a more balanced way with the rotating-leader design, while the single leader in Paxos limits the peak throughput it can achieve.

Figure 10 compares the peak throughput of Fast Mencius with that of Paxos and Mencius, when there is a slow replica. The delay between the slow replica and any non-slow replica is 500 ms. The results show that Fast Mencius still outperforms Paxos significantly. Also, the throughput difference between Fast Mencius and Mencius, which is the overhead incurred by our mechanisms, becomes smaller when the number of replicas increases. This is because the task of revoking the instances coordinated by the slow replica is done by more replicas.

Figure 11 compares the peak throughput of Fast Mencius with that of Paxos, when there is a slow replica, by varying the bandwidth of links from 5 Mbps to 40 Mbps with an interval of 5 Mbps. The delay between the slow replica and any non-slow replica is 500 ms. The system consists of 5 replicas. As expected, the peak throughput of both protocols scales with available bandwidth, while the peak throughput of Fast Mencius increases much faster than Paxos. Figure 11 also shows the peak throughput of Fast Mencius when a non-slow replica fails. Compared with the non-failure case, the throughput of Fast Mencius drops by around 25%. The reason is in a 5-replica system with a slow one, a failure decreases the available bandwidth of the non-slow replicas by 25%.

VII. Related Work

Fast Paxos [15] is an extension of Paxos that admits two execution modes. The fast mode allows clients to directly send their commands to the acceptors. Fast Paxos suffers from collisions, which happens when acceptors receive the commands in different orders. When the number of commands is large, the possibility of collisions is high, and Fast Paxos will have a higher average latency than Paxos. As for throughput, Fast Paxos cannot outperform Paxos, since the number of messages sent/received by each acceptor is the same with Paxos with no collision, and larger than Paxos when collisions happen.

Several consensus protocols deal with collisions by running Paxos and Fast Paxos concurrently. The scheme proposed by Charron-Bost and Schiper achieves the minimum latency between Paxos and Fast Paxos only in failure-free runs [5], while Hybrid Paxos [7] has a higher message complexity and thus lower throughput than Paxos. Besides, these protocols rely on a single leader. The unbalanced communication pattern makes them unable to fully utilize the available resources.

The authors of Mencius also applied the rotating-leader scheme to Byzantine fault tolerance, and proposed RAM [18], a low latency BFT protocol for wide-area networks. Similarly, EBAWA [20] is a BFT protocol that adopts the rotating-leader design. Different from RAM, EBAWA requires $2f+1$, instead of $3f+1$, replicas to tolerate f faulty replicas. It uses a trusted component on the servers to reduce the number of replicas and communication steps required for reaching agreements. Besides state machine replication, fault tolerance has also been widely considered in other areas [6], [21].

VIII. Conclusion

In this paper, we present Fast Mencius, a protocol for state machine replication that tolerates crash failures. Fast Mencius is derived from Mencius, and it enhances Mencius with Active Revoke and Multi-instance Propose. The evaluation shows that in presence of slow replicas, the commit latency of Fast Mencius is significantly smaller than that of Mencius.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.
[2] M. Burrows. The chubby lock service for loosely coupled distributed systems. In *USENIX OSDI*, pages 335–350, 2006.
[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *USENIX OSDI*, 1999.
[4] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
[5] B. Charron-Bost and A. Schiper. Improving Fast Paxos: being optimistic with no overhead. In *PRDC*, 2006.
[6] M. Ding and X. Cheng. Fault tolerant target tracking in sensor networks. In *MobiHoc*, 2009.
[7] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. HP: Hybrid Paxos for WANs. In *EDCC*, 2010.
[8] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
[9] S. Hemminger. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, pages 299–319, 1990.
[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.
[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.
[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
[13] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.
[14] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.
[15] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.
[16] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.
[17] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *USENIX OSDI*, 2008.
[18] Y. Mao, F. P. Junqueira, and K. Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *HotDep*, 2009.
[19] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *USENIX NSDI*, 2008.
[20] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *HASE*, 2010.
[21] W. Wei, F. Xu, C. C. Tan, and Q. Li. Sybildefender: Defend against sybil attacks in large social networks. In *IEEE INFOCOM*, 2012.