Rep4WS: A Paxos based Replication Framework for Building Consistent and Reliable Web Services

Xu Wang, Hailong Sun, Ting Deng, Jinpeng Huai
School of Computer Science and Engineering
Beihang University
Beijing,China
{wangxu,sunhl,dengting,huaijp}@act.buaa.edu.cn

***Abstract*—Web services are widely used to enable remote access to heterogeneous resources through standard interfaces and build complex applications by reusing existing component services. However, massive commodity computers, storage, network devices and complex management tasks running behind web services make them subject to outage and unable to provide continuously reliable services. To address this issue, we present a Paxos-based replication framework for building consistent and reliable web services. The framework mainly consists of a replication protocol and a set of failure tackling algorithms. First, in the replication protocol, besides keeping consistency of service replicas we introduce pipeline concurrency and RDG (Request Dependency Graph) to traditional Paxos so as to improve its performance. Second, we design failure recovery algorithms to recover the failed nodes, which guarantee that each web service has enough available replicas and thus can deliver expected reliability. Third, through an extensive set of experiments, we show that our method is effective in terms of keeping consistency and reliability of web services and it outperforms other replication methods.**

***Keywords*-Web service; consistency; reliability; replica; Paxos**

I. Introduction

In recent years, web services are widely used in Internet and cloud computing applications. Amazon[1], EBay[2] and many other Internet corporations leverage web services to enable developers to remotely access to their computing resources, storage capacities, and integrate their business web applications. With the growth of users and 7×24 continuous service requirement, people start to pay more attention to the reliability of web services.

With the rapid development of cloud computing, the data-centric web services become more popular in many big data clouds. Running behind these web services, those massive commodity computers, disks, network devices and complex management tasks may fail in any time. If failures are not tackled properly, the corresponding web services will become unavailable or even lose and corrupt user data [3]. Thus, it is of great importance to build reliable web services.

Replication is a well-known technique to improve system reliability. With the replication technique, if one replica fails, other replicas will continue to provide services. However, one challenging issue with replication is how to maintain the consistency among multiple replicas. A replicated web service is consistent if the following two conditions hold: (1)The web service is deterministic, which means that it delivers the same output with the same input; (2)All replicas execute the incoming user requests in the same order. Another issue is how to recover the failed replicas so as to maintain enough available replicas.

Several works have been done for replication strategies, such as primary backup [4], two-phase commit (2PC) [5] and group communication (GC)[6]. Primary backup systems often adopt periodical synchronization between the primary node and backup one. If the primary node crashes, the backup will be inconsistent with the primary one. In addition, it is more likely for two nodes to crash simultaneously in large-scale distributed systems, which can cause catastrophic consequences [7]. Therefore, three or more replicas are needed in replication systems. In the 2PC protocol, due to the existence of the coordinator, single node failure is inevitable. And its performance is poor since the coordinator is blocked before all replicas are ready. Finally, group communication performs poor with temporarily unavailable replicas [8]. Furthermore, few of Web service replication frameworks using group communication consider recovering failed nodes.

Paxos [9,10], as an asynchronous consensus algorithm in distributed systems, is essentially optimal [11] and has been proved its safety and liveness [12]. It can tolerate up to f non-Byzantine failures when there are $2f + 1$ replicas. Moreover, Multi-Paxos [10] that runs multiple Paxos instances can be used to guarantee that all replicas execute requests in the same sequence. The Paxos-based replication framework can avoid the drawbacks of primary backup, 2PC and GC. First, it can easily support three or more replicas which can tolerate simultaneous crash of multiple nodes. Second, Paxos can maintain consistency by accessing to only a majority of replicas instead of all. Thus it can achieve better performance than 2PC. Third, compared with group communication, Paxos incurs less overhead for its natural tolerance of temporary unavailability.

In this work, we present Rep4WS, a Paxos-based replication framework for building consistent and reliable web services. To the best of our knowledge, we are the first to introduce Paxos to address deterministic Web service replication issues. Rep4WS mainly consists of a replication protocol and algorithms for tackling failures. In the replication protocol, a distinguished replica is elected as the *leader* to propose requests. And the others act as *followers*. By following the

replication protocol, all replicas of a web service are able to execute requests in the same sequence and thus achieve consistency. The replication protocol involves two phases: the agreement phase and execution phase. In the agreement phase, we use pipeline to improve its concurrency rate, which is called *pipeline concurrency*; in the execution phase, we use *requests dependence graphs* (RDGs) to improve its performance. By building RDGs to characterize the dependence relations of requests, many requests are able to execute in advance by some replicas if all requests on which they actually depend are executed. For example, given a request sequence is $(read(A), read(B), read(C))$ in a centric data web service, where "$read$" is an operation of the web service. Though the request $read(C)$ is after $read(A)$ in the sequence, it may arrive earlier in some replicas because of network latency. Normally, $read(C)$ cannot be executed until $read(A)$ and $read(B)$ arrive. In fact, it can be executed in advance and does not break consistency because it does not depend on $read(A)$ and $read(B)$ at all. The RDGs can clearly describe the dependence relations of requests and make some requests execute in advance, so they can be used to optimize the performance of the execution phase. In terms of failure tackling, we design two algorithms for recovery. If the failure node is the leader, a leader election algorithm is presented to decide which replica is the new leader; if one follower fails, we also propose a recovery algorithm to recover the follower to the leader's state and then rejoin it to the available replica group.

The main contributions of the paper include: (1)We present a Paxos-based Web service replication framework, which can build consistent and reliable web services; (2)We describe how to use pipeline concurrency and RDGs to improve the framework's performance; (3)We present two algorithms of leader election and follower recovery to deal with failures; and (4)We have implemented the framework and experimental results show its effectiveness and performance advantages compared with other replication methods.

The remainder of this paper is structured as follows. Section II introduces related work. Section III describes overview of the replication framework. The replication protocol is presented in Section IV. Section V describes how to tackle failures. Section VI shows our experimental results. Finally, Section VII concludes.

II. Related work

A. Web Service Replication Frameworks

Primary backup replication has been used to achieve fault tolerant web services [4]. It enables the primary web service to periodically checkpoint data to its backup. But if the primary fails, the state of the next checkpoint will be lost, which results in inconsistency.

There are several group communication toolkits, such as Horus [13] and JGroups [14], and many Web service replication frameworks based on them. Birman [15] proposed an extended architecture of adding reliability to web services; Ye [16] introduced a middleware to support reliable web services; WS-replication [17] was an infrastructure providing high available web services. Compared with Rep4WS, these frameworks based on group communication perform poorer in the performance. The reason is that, when any member crashes, it must incur comparable overhead only when the leader of a Paxos-based framework crashes. While the followers become unavailable, the Paxos-based framework will not bring any extra overhead [18].

B. Other Replication Systems

Besides Web service replication frameworks above, there are many other replication systems. A data platform [5] used 2PC to maintain the consistency of data replicas. But the single node failure and poor performance is inevitable because of the existence of the coordinator and strong synchronization in 2PC. Furthermore, the failure of any node will lead to abort in 2PC; DeDiSys [19] was a replication architecture for the platform-independent middleware; Postgres-R [20] was a data replication system using GC to order and replicate transactions. Since DeDiSys and Postgres-R were based on GC, they had similar disadvantages described above.

C. Paxos

Paxos [9,10] is an asynchronous consensus algorithm in distributed systems which is used to agree on a given value for a group of nodes. It can reach a consensus while a quorum of nodes are alive. As described in [10], Paxos proceeds as follows in a group of nodes:

1(a): A node which wants to propose a value selects a proposal number n and sends a $prepare$ request with number n to all nodes.

1(b): A node receives a $prepare$ request and gets the number n. Let n' be the number of the highest-numbered request which the node has already responded. If $n > n'$, it responds to the request with a promise not to accept any more proposals numbered less than n, and with the highest-numbered proposal that it has accepted if the proposal exists. Otherwise, the node ignores the request.

2(a): If the proposer receives $response$ messages from a majority of nodes in the group, then it sends an $accept$ request to each of those nodes for a proposal numbered n with a value v. If the highest-numbered proposal among the responses exists, v is its value. Otherwise v can be any value.

2(b): A node receiving an $accept$ request numbered n with a value v accepts it by sending an $accepted$ message if it does not violate all promises made by the node.

The SMART [18] elaborated how to migrate Paxos-based replicated stateful services. However, SMART did not discuss recovery from temporary unavailability for replicas that may be more lightweight than migration. Zhang [21] focuses on replicating nondeterministic grid services based on Paxos. Rao [22] used Paxos to build a scalable, consistent and highly available datastore Spinnaker. Though writes in Spinnaker were proposed in parallel, it was not clear whether the pipeline concurrency was used. In addition, in our work, we construct dynamic RDGs to further improve the performance of Rep4WS on the basis of the specific feature of web services.

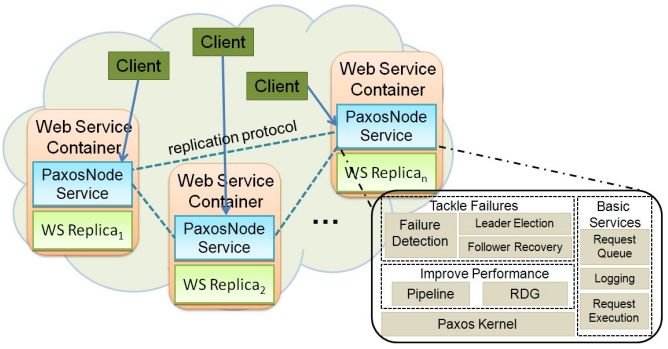

Fig. 1. Rep4WS-A Replication Framework Based on Paxos

III. Overview of The Framework

Web services are deployed over Internet, which rely on SOAP engines to transport messages over SOAP. As shown in Figure 1, if one web service has n replicas in Rep4WS, every web service container of a replica will deploy a PaxosNode service. These PaxosNode services constitute a configuration, including their addresses and the leader information. Each PaxosNode service contains several components, which enable Rep4WS to run a replication protocol and failure tackling algorithms.

First, all replicas run a replication protocol to execute requests in the same sequence and achieve consistency, which will be described in Section IV. Request queue,logging and request execution are basic services, and Paxos kernel runs basic Paxos instances. For improving performance, Pipeline component can run α Paxos instances from request n to $n+\alpha$ in parallel, where α is a tunable window size parameter. Moreover, RDG component is used to construct RDGs to execute some requests in advance for improving concurrency.

Second, once failures occur, Rep4WS will tackle them by running corresponding algorithms which will be presented in Section V. When one follower fails, the leader will run the recovery algorithm. However, the possibility that the leader may fail is terrible. The leader must send periodical HEARTBEAT messages to followers. Once a follower does not receive a HEARTBEAT message for a while, the follower will start the algorithm of leader election to elect a new leader.

Note that, although multiple proposers are allowable in the standard Paxos, which may improve system throughput, it will produce livelock when different proposers decide the orders of the requests which are ready to be submitted [10]. So we select a distinguished PaxosNode in the current configuration as the leader for proposing requests, while the others act as followers. If requests from clients are sent to followers, they need be redirected to the leader. In practice, web service providers usually offer developers with client packages, such as eBay SDKs [23]. We can cache leader information in such client packages to reduce redirection times.

IV. The Replication Protocol

This section describes our replication protocol, which is based on Paxos but differs from a standard one. Figure 2 indicates the whole process of the replication protocol, containing the agreement phase and execution phase. The agreement phase is made to agree on which request is the ith request in the sequence, and the execution phase decides when and how to execute requests.

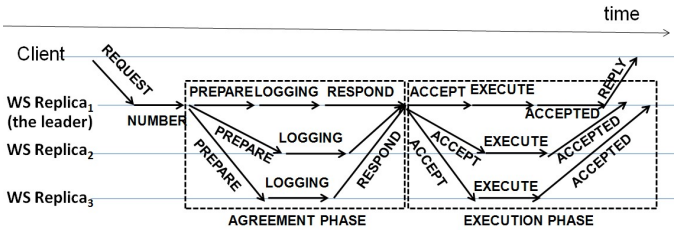

Fig. 2. Replication Protocol in Rep4WS

As shown in Figure 2, when a request arrives at the leader, the leader firstly assigns the request a positive integer as its unique identifier representing its order of arrival, such as 001. Then the leader propagates the request to all replicas via PREPARE messages, including itself. Once a replica receives the PREPARE message, it will log the message into local storage, and then return a RESPOND message to the leader. After the leader receiving RESPOND messages from a majority of replicas, it sends ACCEPT messages to all replicas. If a replica receives the ACCEPT message of one request, the request will enter execution phase. After execution, the result will be sent to the leader by the ACCEPTED message. Once the first result of one request arrives at the leader, the leader will reply to the client.

With the replication protocol, all replicas run in compliance with a FSM shown in Figure 3. Take request $n+i(0 \leq i < \alpha)$ as an example. Normally, a replica receives the PREPARE message, logs the request and receives the ACCEPT message, then the state of the request changes to "$Accept$". But the replica can receive the ACCEPT message before the PREPARE message. This is highly possible due to network latency. After logging, the request also goes to state "$Accept$". The replica will wait until the RDG component tells it the request $n+i$ is executable. Then it will execute the request by local web service invocation and obtain the result. When the replica has sent the result to the leader by a ACCEPTED message, the state of request $n+i$ is set to be "$Finished$". The RESPOND message is ignored because it has no influence on the states of requests. Note that, FSMs in memory may be lost in failures. Only three states "$Logged$", "$Accept$" and "$Executed$" are persisted in the stable storage.

A. Pipeline Concurrency

The agreement phase can perform in parallel, namely pipeline concurrency. The concurrency rate is decided by the window size α. It means that requests $n, n+1, ..., n+\alpha-1$ can be concurrently proposed to the agreement phase if all requests whose identifiers are less than n have been executed. The requests being processed constitute the current pipeline. Pipeline concurrency is able to improve performance especially when massive requests are proposed. Note that the window

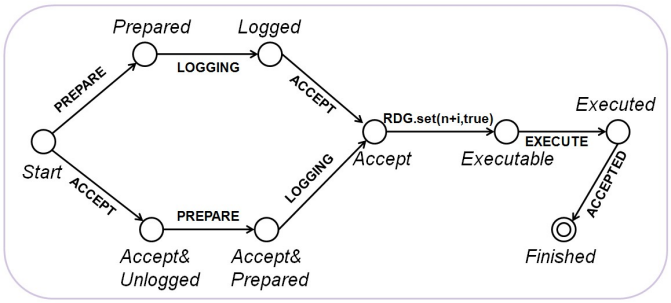

Fig. 3. The FSM of One Replica

size α should not be set too large because one request must wait until its all dependence requests are executed before its ultimate execution in the execution phase. Although a request $n+\beta(\beta < \alpha)$ enters its execution phase much earlier, if β is larger, its dependence requests have to wait for longer time to be executed. That is, its response time perceived by the client does not become shorter. In practice, when α is large enough, the average response time will almost have no change. Furthermore, when α is too large, each replica must consume many resources like memory and threads to deal with the waiting requests in the execution phase. Then it will influence the performance of executing requests.

B. RDG Optimization

Generally, to maintain the consistency of all replicas, the request $n+i$ is executable only when all requests before $n+i$ have been executed. However, in the web service scenario, some requests do not have to satisfy this constraint. For instance, eBay web services include many searching and information retrieval operations. Requests to these operations can exchange their execution orders and do not break consistency.

At first, we model a web service as a state machine which takes a request in a state and produces an output and a new state. Specifically, $WS(s_n, r_n) = (s_{n+1}, o_n)$ means that after executing a request r_n in the state s_n, the web service outputs o_n and transfers to the state s_{n+1}. We use an operation exchangeable table (OET) for a web service to describe the exchangeability between any two operations, and request dependence graphs (RDGs) for all requests to indicate the dependence relations between any two requests.

Definition 1.[Operation Exchangeable Table] $T = \{(x, y)|x \in P, y \in P,$ and x and y are exchangeable$\}$, where P is the collection of operations in a web service. For any state s and two requests r_x and r_y which invoke operation x and y respectively, if $WS(s, r_x) = (s_x, o_x)$ and $WS(s_x, r_y) = (s_y, o_y)$, and $WS(s, r_y) = (s'_y, o'_y)$ and $WS(s'_y, r_x) = (s'_x, o'_x)$, operation x and y are exchangeable if and only if $s'_x = s_y, o'_x = o_x$ and $o'_y = o_y$.

Definition 2.[Request Dependence Graph] $G = (V, E)$, $V = \{r|r$ is a logged request$\}$, $E = \{(x, y)|x \in V, y \in V,$ and $(op(x), op(y)) \notin OET\}$, where requests must be logged to insure that their contents have been acquired, *op* maps requests to their operations, and request x depends on y if and only if there is a path from x to y in G.

In the replica FSM, once a request's state transfers to "$Accept$", it will ask the RDG component whether the request is executable. The RDG component uses Algorithm 1 to decide it. As shown, RDG is dynamically constructed for request $n+i$ (lines 3-10). If there is an edge (x, y) and another path from request x to request y in RDG, they both means that request x depends on y. So we can remove such edge to simplify the RDG (lines 11-15). Finally, only if all requests y where there is an edge $(n+i,\ y)$ in the RDG are executed, the request $n+i$ is executable.

Algorithm 1 RequestExecutableDesicison

```
1: let T be the OET;
2: let V = {requests n, n+1, ..., n+i}, E = ∅;
3: for each request y from request n to n+i do
4:    for each request x from request y+1 to n+i do
5:       if ((op(x), op(y)) ∉ T) then
6:          E = E ⋃ (x, y);
7:       end if
8:    end for
9: end for
10: let RDG = (V, E);
11: for each e = (x, y) of E do
12:    if there is anothor path from x to y excepting e then
13:       E = E − e;
14:    end if
15: end for
16: for each e = (x, y) of E do
17:    if x =request n+i then
18:       if y.state = Executed|y.state = Finished then
19:          continue;
20:       else
21:          return false;
22:       end if
23:    end if
24: end for
25: return true;
```

In practice, we can use an incremental algorithm instead of building RDGs from the beginning of the current pipeline for each request, which can be easily implemented.

V. Tackling Failures

Web services can keep their consistency of multiple replicas and reliability with the replication protocol if all replicas operate normally. Unfortunately, they usually encounter various failures. We assume that failures are non-Byzantine and simultaneous failed nodes are less than half of all nodes, which are also the assumptions of standard Paxos. This section describes how to deal with failures of the leader and followers.

A. Leader Failure

If the leader fails, followers will detect it and then begin the leader election. Algorithm 2 describes the details of leader election. Let $f.LR$ denote the collection of logged requests

for replica f, and v denote the variable whose value will be made consensus in one Paxos instance. Once a follower f discovers the failure of the leader, it starts a pre-election Paxos instance with $v = f.LR$ (line 1). When one follower m has learned LR from a majority of followers, it will compute the maximal-numbered request $MLR.mlr$ of MLR (line 5) , the union set of LRs. Then m selects one follower containing $MLR.mlr$ as a new leader candidate and try to propose it using a Paxos instance to all live replicas. Although more than one nodes may have learned LR from a majority of nodes and select different nodes as new leader candidates, all live replicas will make a consensus on an unique node as the new leader due to the safety of Paxos(line 7). Finally, all nodes learn the unqiue new leader, MLR and then increase their own current configuration ID by 1.

Algorithm 2 LeaderElection

1: this follower f runs a Paxos instance with $v = f.LR$;
2: wait until one follower m has learned LR from a majority of nodes;
3: **if** $f = m$ **then**
4: $MLR = \bigcup_{majority} LR$;
5: $MLR.mlr$=the maximal-numbered request of MLR;
6: the new leader candidate lc is one node containing $MLR.mlr$;
7: m runs a Paxos instance with $leader = lc$;
8: **end if**
9: f learns the new leader and corresponding MLR;
10: f increases its current configuration ID by 1;

After leader election, the new leader will transfer all nodes into a consistent state and take the responsibility of a leader. As a leader, it catches up to the consistent state by executing requests which have not been executed in MLR and by filling in *noop* request if one request neither in log nor in MLR, where a *noop* request is simply an extra request whose execution does nothing; For any follower, it removes requests from log whose identifers are greater than $MLR.mlr$ or mismatching with MLR, catches up to the consistent state by executing requests which have not been executed in MLR and by filling in *noop* request if the request is not in MLR. Generally, the new leader and all followers reach consistency by taking three ways: removal, catching up and filling up. Then the new leader opens for new requests from clients.

Note that Algorithm 2 guarantees that any executed requests in all replicas are in MLR and their identifiers are less than $MLR.mlr$. This property of Algorithm 2 allows us not have to roll back any replica when reaching the new consistency. Since the execution of any request must happen after a majority of replicas accept it in the replication protocol, the request is in a majority of LRs. And MLR is a union set of one majority of LRs. So the request also appears in all $MLRs$. According to Algorithm 2, the new leader has the maximal-numbered request of one MLR. That is, the identifer of any executed request is no more than $MLR.mlr$.

B. Follower Failure

If some followers fail, the replication protocol will consider that they are running very slowly or network latencies are very long. In the agreement phase, since the number of failure nodes is less than half the number of all replicas, the leader will eventually reach an agreement with the live replicas. In this way, the replication protocol naturally tolerates the failures of followers. However, the failed follower becomes unavailable and the reliability of the corresponding web service decreases. So recovery is necessary. In practice, failed followers are recovered in parallel while the other nodes run normally.

When a leader has detected one failed follower, it will create a thread to recover the follower. The recovery includes three phases: local recovery, removal, and catching up, which is depicted in Algorithm 3. Let $f.LR$ denote the collection of logged requests, $f.ER$ denote the collection of requests which have been executed and $f.frcp$ denote the first request of the current pipeline for the replica f. Similarly, $leader.LR$,$leader.ER$ and $leader.frcp$ can be defined. $number(r)$ gets the identifier of request r. According to Algorithm 2, the leader has all preceded $MLRs$ and configuration IDs. At first it sends MLR whose configuration ID is the same with that of the follower (line 1). In local recovery phase, the follower re-executes the executed records, including the ER of the current pipeline (lines 2-7), if it has lost them from the checkpoint. If the leader is re-elected during the failure of the follower, the follower must remove some inconsistent requests with MLR from its log (lines 8-16). These requests consist of those whose identifiers are greater than $MLR.mlr$ and which mismatch with MLR. During catching up phase, the follower catches up with $MLR.mlr$ and then the leader's ER(lines 17-36). Finally, the follower replaces its current configuration ID with the one of the leader.

Note that if the leader does not change during one follower's failure, the removal phase and catching up with $MLR.mlr$ will not work. When the leader changes, e.g., the recovery of an old leader, the failed node must do something to achieve the consistent state as if it did not fail in the configuration when it failed. In addition, if the leader changes more than one time, since each leader election assures the consistency of all live replicas, the recovered follower will be consistent with all replicas if it has the same state of the current leader.

VI. Experimental Results

This section shows the experimental results of Rep4WS. Some experiments are performed to compare with conventional replication approaches. Then we conduct several experiments to understand the overheads of Rep4WS.

Our experimental test bed is built on seven commodity computers. They are all DELL OPTIPLEX 755 computers, each with a Intel Core 2 3.0GHz processor, 4GB of memory and 500GB 7200 RPM SATA hard drive. Each runs Windows XP Professional with Service Pack 3 and is connected with a 100Mb/s LAN subnet. Apache Axis2 SOAP engines[24] as web service containers are installed on all nodes.

Algorithm 3 Recovery

receive MLR from the leader;
for each record r from the recent checkpoint to $f.frcp$ **do**
  re-execute record r;
end for
for each request r in $f.ER$ **do**
  re-execute request r;
end for
for each request r in $f.LR$ **do**
  if $number(r) > number(MLR.mlr)$ **then**
    remove r from log;
  end if
  if $r' \in MLR \wedge number(r') = number(r) \wedge r' \neq r$ **then**
    remove r from log;
  end if
end for
update $f.LR$;
for number $i \in [number(f.frcp), number(MLR.mlr)]$ **do**
  if there is a request r numbered i in $MLR - f.ER$ **then**
    re-execute request r;
    if $r \notin f.LR$ **then**
      logging request r;
    end if
  end if
  if there is no request numbered i in MLR **then**
    logging request $noop$;
  end if
end for
update $f.ER$;
for each request r from $f.frcp$ to $leader.frcp$ **do**
  if $r \notin f.ER$ **then**
    re-execute request r;
  end if
end for
for each request r in $leader.ER$ **do**
  re-execute request r;
end for
f sets its current configuration ID with that of the leader;

A. Performance Comparison

We design an experiment to compare the performance of Rep4WS with those based on 2PC and GC, as well as the basic Paxos that does not use pipeline concurrency and RDG optimization. We implement the standard 2PC protocol to guarantee the consistency of all replicas. A typical GC system is JGroups[14] which is an open-source distributed framework and almost supports all features of GC. In our experiment, we use JGroups 3.0 and the protocol stack includes TCP and SEQUENCER for reliable multicast and total order requests.

The web service we select is used to store keys and their values, namely KVStore. Its basic operations are as follows: $read(key)$, read the value of the key from the hard disk; $write(key, value)$, update the value of the key. If the key does not exist, insert the key and its $value$ as a new row; $delete(key)$, delete the key and its value. In this experiment, all requests we evaluate are the ones to invoke the operation $read(key)$. Since these requests are executed very quickly, it highlights the factors we concern, such as network latency and replication protocols.

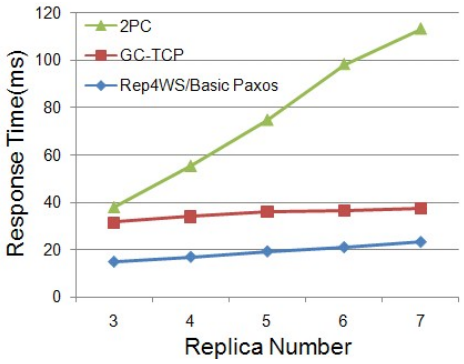

Fig. 4. Consecutive Requests

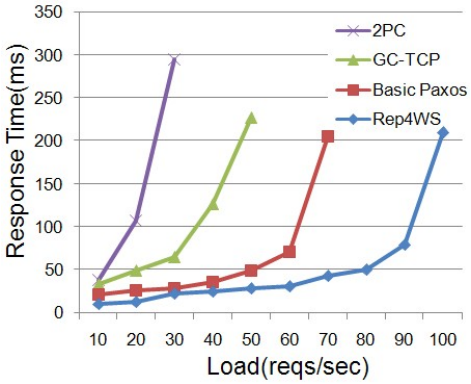

Fig. 5. Concurrent Requests

Figure 4 shows the average response time based on 2PC, JGroups(GC-TCP), Basic Paxos and Rep4WS respectively as the replica number is increased. For each response time, a client sends 12,000 consecutive requests to invoke the same $read(key)$ and calculates their average response time. As shown, generally the average response time of Rep4WS is 54.1% and 27.7% of that of JGroups(GC-TCP) and 2PC respectively. Since only one request is executed in one time, Rep4WS and Basic Paxos perform the same. In addition, the response time rises as the replica number is increased. This is because new replicas will incur extra overheads in message transport, thread synchronization and so on. Figure 5 depicts the concurrent performance among Rep4WS, Basic Paxos, JGroups(GC-TCP) and 2PC as the load is increased when the replica number is default 3. We can see that the average response time of 2PC, GC-TCP and Basic Paxos curves much sooner than that of Rep4WS.

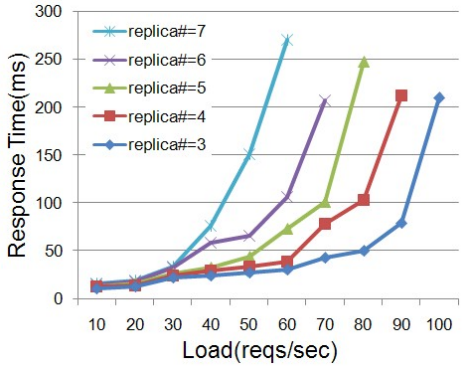

Fig. 6. Load Impact

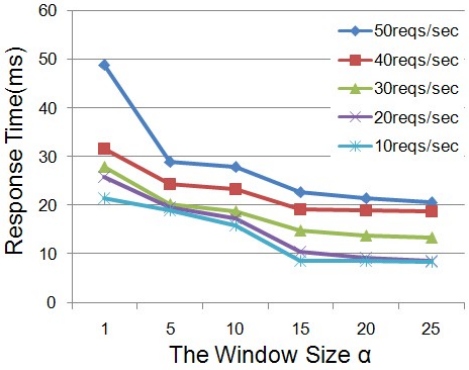

Fig. 7. The Window Size

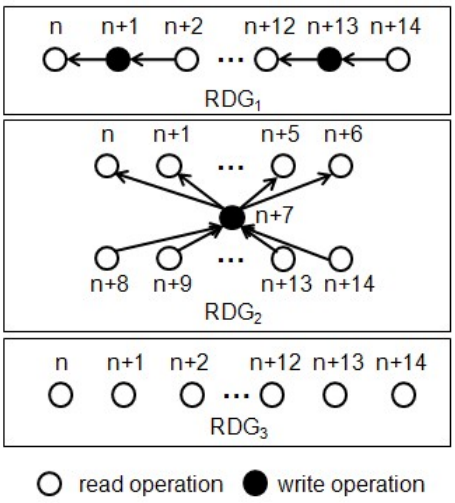

Fig. 8. Three Typical RDGs of KVStore

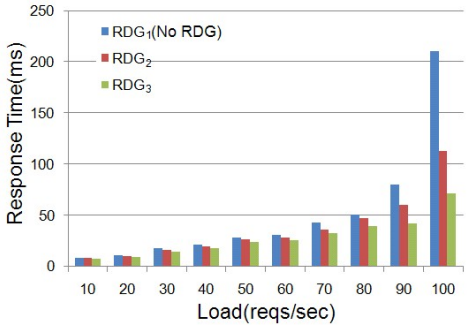

Fig. 9. RDG Optimization

Rep4WS performs better than those based on JGroups(GC-TCP) and 2PC in both cases and Basic Paxos in concurrency case. The main reasons are that Rep4WS only needs accessing a majority of replicas and is improved its concurrency by pipeline concurrency and RDG optimization.

B. Load Impact on Performance

This experiment is to evaluate the average response time of Rep4WS as the load is increased when the replica number changes. For each response time, clients send requests in parallel to invoke the same $read(key)$ and calculate their average response time. As seen in Figure 6, the average response time curves sooner with the replica number increasing because of extra overheads incurred by new replicas.

C. Window Size α in Pipeline Concurrency

In Rep4WS, the pipeline concurrency is used to improve its parallel performance. This experiment is designed to measure its impact on performance where RDG optimization is disabled. Figure 7 shows the response time as the window size α is increased. As shown, the average response time can decrease 40%-66% when the window size α increases form 1 to 25. However, when α=15,20,25, the average response times drop down very tinily whatever the load is. The reason has been described in Section IV. So the best value of α is the one from which the average response time begins to converge.

D. RDG Optimization

During the execution phase of the replication protocol, RDG enables some requests to execute in advance for inducing their waiting time. In this experiment, the window size is set its default value of 15. Three typical RDGs of web service KVStore are shown in Figure 8. In RDG_1, the $read$ and $write$ operations of KVStore are requested alternatively; only one $write$ request appears in the middle of RDG_2; and all requests are $read$ operations in RDG_3. In the execution phase, only the requests of $read$ operations can exchange their execution order. Note that RDG_1 is equivalent to no usage of RDG. Figure 9 shows the average response time of these three RDGs as the load is increased. As shown, the response times from high to low are RDG_1,RDG_2,RDG_3 in the same load. And the response time of RDG_3 is smaller 15%-66% than RDG_1. In generally, the performance of RDG_3 is averagely better 26% than RDG_1. The reason is that the average waiting times of these three RDGs during the execution phase of the replication protocol are RDG_1,RDG_2,RDG_3 in descending order.

E. Leader Election

If the leader encounters failure, the web service will enter a small unavailability window. This experiment is performed to

measure the time of leader election, which is from the failure of the leader detected by replicas to the new leader learned by all replicas. According to Algorithm 2, the time of leader election may depend on the replica number and the window size α. But the experiment shows that the leader election time almost equals in the same replica number regardless of the α value. Its reason is that the calculation of $MLR.mlr$ and the selection of a new leader candidate cost less than $1ms$ whatever the value of α is. As shown in Table I, the time of leader election is proportional to the replica number because every replica almost brings about the same network latency during leader election.

Although the leader failure makes the web service unavailable, the probability is very low and the unavailable time is very short. In addition, clients can use some brief methods to deal with this issue.

TABLE I
LEADER ELECTION TIME

Replica Number	3	4	5	6	7
Leader Election(ms)	85.2	110.14	126.27	185.89	205.74

F. Recovery

Once one follower fails, the leader will detect its failure after 2 seconds which is the default HEARTBEAT interval and then start to recover the follower. At the same time, the leader also receives new requests. This experiment is designed to measure the recovery time with different loads. As shown in Table II, the recovery time becomes longer as the load is increased. This is because the number of requests which need be catched up rises when the load increasing. Note that the recovery performs in parallel and almost has no impact on the normal operations.

TABLE II
RECOVERY TIME

Load(reqs/sec)	10	20	30	40	50
Recovery Time(s)	0.19	0.46	0.78	1.03	1.3
Load(reqs/sec)	60	70	80	90	100
Recovery Time(s)	1.82	2.21	3.02	3.52	4.57

VII. CONCLUSION

In this paper, we present a Paxos-based replication framework Rep4WS for building consistent and reliable web services, mainly including a replication protocol and algorithms for tackling failures. In the replication protocol, pipeline concurrency and request dependence graphs are used to improve its performance. We also discussed how to tackle failures for the leader and followers when less than half of replicas encounter non-Byzantine failures. Our experiments show that Rep4WS is effective and outperforms conventional ones.

In future, we will enhance Rep4WS by supporting nondeterministic web services and some limited Byzantine failures. In addition, Rep4WS can be extended for more widely web-based services, such as RESTful services, data services and so on.

ACKNOWLEDGMENT

This work was supported partly by National Natural Science Foundation of China (No. 61103031), partly by the State Key Lab for Software Development Environment (No. SKLSDE-2010-ZX-03), partly by the Fundamental Research Funds for the Central Universities (No. YWF-11-03-Q-040, YWF-11-05-013 and YWF-11-02-206), and partly by China 863 program (No. 2012AA011203) and by CPSF(No. 2011M500218).

REFERENCES

[1] Amazon Web services. http://aws.amazon.com/
[2] eBay Developer Network. https://www.x.com/developers/ebay
[3] Amazon.com. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. April 2011
[4] LIANG D, FANG C L, CHEN C et.al. Fault tolerant Web service. Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC 2003)
[5] F. Yang, J. Shanmugasundaram, and R. Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In CIDR, 2009
[6] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. ACM Computing Surveys, 33(4):427-469, Dec. 2001.
[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder,A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In FAST, pages 8:1-8:28, 2008.
[8] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS-26),pages 168-177, Sendai, Japan, June 1996.
[9] L. Lamport. The Part-Time Parliament. In ACM Trans. On Computer Systems, pages 133-169, 16(2), 1998.
[10] L. Lamport. Paxos Made Simple. ACM SIGACT News, 32(4):18-25, December 2001.
[11] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults-a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).
[12] B. Lampson. The ABCD's of Paxos. In Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC '01), page 13. ACM Press, 2001.
[13] R. Van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. Communications of the ACM, 39(4):76.83, April 1996.
[14] JGroups. http://www.jgroups.org.
[15] K. Birman, R. van Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In Proc. of Int. Conf. on Software Engineering (ICSE), 2004
[16] YE X, SHEN Y. A Middleware for Replicated Web Services. In ICWS 2005
[17] SALAS J, PEREZ-SORROSAL F,et.al. WS-replication: a framework for highly available web services. International World Wide Web Conference(WWW) 2006
[18] Jacob R. Lorch,etc. The SMART Way to Migrate Replicated Stateful Services. ACM SIGOPS 2006.
[19] J. Osrael, L. Froihofer, K. M. Goeschka, S. Beyer, P. Galdamez, and F. Munoz. A system architecture for enhanced availability of tightly coupled distributed systems. ARES'06, IEEE Computer Society,2006
[20] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In VLDB 2000
[21] X. Zhang, F. Junqueira, etc. Replicating Nondeterministic Services on Grid Environments. In HPDC 2006
[22] Jun Rao, Eugene Shekita, Sandeep Tata. Using Paxos to Build a Scalable, Consistent,and Highly Available Datastore. In VLDB 2011
[23] eBay SDKs for APIs. https://www.x.com/developers/ebay/documentation-tools/sdks
[24] Apache Axis. http://axis.apache.org/