Distributed Generic Name Service

Zhaohua Li Zhiqiang Ma Lin Gu
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Email: cs_lzxab@stu.ust.hk, zma@cse.ust.hk, lingu@cse.ust.hk

***Abstract*—On many distributed platforms, consensus among multiple servers should be obtained in order to maintain consistent system state. Paxos has been selected as the core of many different distributed services, such as lock services and storage services. This report presents an empirical study on a name service based on Paxos in a virtualized environment. Details of the design as well as matters concerned during implementation are discussed. We also propose an API for general name services.**

I. Introduction

Name service is a core function in cloud-based systems. It supports storing attribute and value pairs as well as retrieving information when it is given the name of an entry. Therefore, clients can write data into the database of the name service server and read what they have written later, or they can communicate with other clients by writing and reading a specific entry with a name they have agreed on. Remote procedure calls have embeded this service to store the function name and the computer destination in order to match users requests to the right machine [2]. DNS also serves as a world-wide venue to resolve the IP address for a domain name. Similar use cases can be seen in a large number of applications and systems.

In prior work, we have implemented a name service server served by a single process. This server listens to clients and handles their requests 24 by 7 if it runs steadily and encounters no failure. However, the server can never be perfect and we should expect failures of the server. Hence, multiple servers should collaborate to avoid the suspension of the service.

To provide a reliable and fault-tolerant service, we need multiple machines (or virtual machines) to act as a server cell. To guarantee users can obtain the same response from different servers when the database is steady, the database of different servers should be consistent regarding the state of entries in the database in a particular time when all pending updates have been fulfilled. To obtain the consistency, we run a consensus algorithm, Paxos, among those servers [6][5].

In this report, we discuss the design of the distributed name service and the details of the implementation. We show the evaluation results and discuss possible improvements as future work.

II. Related Work

One noticeable example of Paxos-base system software is the Chubby lock service for loosely-coupled distributed system which intends to provide datacenter-wide locking as well as reliable storage [3]. While Chubby combines the locks and reliable storage in one service, Boxwood[7] is another lock service with a different design: it explicitly contains three major parts, a lock service, a Paxos service and a failure detection service. The Paxos algorithm in Boxwood is implemented as a library, which is used to stored client states, and might be able to be reused in other projects.

ZooKeeper is also an important service with the Paxos core for maintaining configuration information, naming, providing distributed synchronization, and providing group services [4]. The broadcast protocol in ZooKeeper, Zab, employs the Paxos algorithm to make the service tolerant to failures and loss of messages [9].

Google's Megastore is another application of the Paxos algorithm for data storage and retrieval among its geographically distributed datacenters [1]. Megastore uses Paxos to replicate user data and provide responsive services among different datacenters. Megastore uses a non-master based Paxos implementation.

III. Design

The name service is a database with names and attributes which supports data insertion and retrieval across a network. "Attributes" are the values determined by "names". The collection of names and their associated attributes constitute a database, in which a name is assumed to be unique. With a specified name and different commands, the content of the name can be created, updated and deleted. Although these three actions are different in context, they are all handled by the Paxos protocol.

A. Requirements

To achieve consensus among multiple nodes with lossy communication channels in the Paxos protocol, a series of actions, which is called a synod, is conducted before an entry can be inserted or updated in the database. A vote is a node's decision upon a synod. A majority of the nodes need to confirm their availability for the votes and vote for the related update later. Successful votes are then broadcast to the nodes for later updates. Through these communications among nodes, only one value is allowed for a synod with a particular instance number. This algorithm provides the consistency among servers by requiring that the selected value must be accepted by a majority of servers. A technical challenge is that, to become an applicable service in datacenters, it must

be fast in response and stable.

B. Definitions

In order to provide a fault-tolerant distributed name service, multiple processes on multiple nodes need to maintain their own databases to prevent server failures as well as to increase the throughput of the service. To guarantee the consistency of the database of different processes, we run the Paxos algorithm among the processes. The following sections present the basic design of the service. Some terminologies are defined below.

Replica
: A process duplicated to store data with redundancy as well as to improve the throughput of the service.

Synod
: A series of actions conducted to achieve consensual knowledge of a particular update among multiple nodes.

Vote
: A value proposed to meet agreement among multiple processes during a synod.

President
: A process which is in charge of proposing a vote among the replicas

C. Communication between replicas

A process should have a copy of other replicas' IP addresses and port numbers in order to communicate with them. Since different replicas have different sockets to listen to clients, it is sufficient to use the IP address and port number to distinguish the replicas.

Every process has a process table and it needs to maintain it continuously. When a process needs to send messages to other processes, it needs to retrieve IP addresses and port numbers of the receiving processes. The IP address and port number of the president should be marked in the process table.

In addition, the table should also record time stamp for the addresses. Addresses that failed to renew their live status should be neglected.

D. Replica Roles

The president is in charge of the process of voting of every individual synod. It records the total instance number, LATEST. When it wants to purpose a new vote, LATEST will be the instance ID.

A thread will be created to be the president of a single synod [6] (see Fig. 5 for the state diagram of a single synod and Table I for the detailed definition of terms). It sends prepare requests with proposal ID SYNOD_N to a selected majority of processes in its ID table. It waits for the "promises" until a predefined time interval elapses, say WAIT_PROMISE_TIME. If it cannot receive replies from a majority of processes within the time interval, it resends the "prepare" request with a different majority and repeats the previous process. If it receives the promises from a majority of processes, it determines the voted value according to the replies and continues to the "accept" step by sending to the majority of processes "accept messages".

To learn about the instance value of a specific single synod, we let the president wait for the replies of "accept messages", namely "accepted messages", from replicas. If it can collect a majority of replies, it means that the value is successfully determined, i.e. a majority of replicas know about the vote value. Then it sends "confirm messages" to other replicas to broadcast the vote value, then the synod ends successfully. If it cannot collect a majority of "accepted messages", it means that some of the replicas previously promise their acceptance fail to accept the vote due to another promise to a higher vote number or failure and the synod ends with failure.

TABLE II
CONFIGURATION OF CONSTANTS

SERVER PORTBASE	1700
RENEW PORTBASE	1800
RENEW TIME (sec)	30
RENEW TIMEOUT (sec)	60
RENEW SLEEP (sec)	10
WAIT PROMISE (sec)	1
WAIT PROMISE TIME (μs)	55000
HASHSIZE	10000
DGNS1 IP	10.0.1.101
DGNS2 IP	10.0.1.102
DGNS3 IP	10.0.1.117
DGNS4 IP	10.0.1.118
DGNS5 IP	10.0.1.119

E. Replicas

Apart from requests from the network, replicas receive voting messages from president. It assigns a task to a thread with the messages. The voting tasks look up the instance log and react according to the Paxos protocol.

A replica updates its own database sequentially according to its log until it hits an empty entry for an synod. An empty synod occurs when a replica has a failure earlier and misses the vote or when the entry is indeed empty for all replicas (a vote to be determined). When it encounters an empty entry, it asks the president for the synod information and updates it accordingly. All updates are done by a separate thread.

F. Failures and membership management

When a failure occurs, some servers might stop listening to the president and clients. New servers will be started to guarantee a large enough number of replicas know about the information and duplicate the database. However, new servers should not contribute to the votes immediately after the start up since the information in their database is not complete.

When a new node starts, it sends renew messages to other replicas and announce its existence. Other replicas will notice that it is new and avoid sending it vote messages such as "prepare" and "accept". The new member will start copying older vote decisions from other replicas. Only when its vote information is nearly up-to-date is it eligible to participate in

TABLE I
SYNOD STATE DESCRIPTION

State Action	What it actually does
P_INIT	initialize the synod with client request information
P_REQUEST_PREPARE	select a majority of replicas and send prepare messages to them
P_WAIT_PROMISE	wait for promise from the selected majority and count the number of promises
P_REQUEST_ACCEPT	send accept messages to the selected majority
P_WAIT_ACCEPT_RESPONSE	wait for accepted messages from the selected majority the count the number of responses
P_REQUEST_CONFIRM	send determined vote value to all replicas to announce the vote

votes.
Also, packets can be lost due to congestion in the network, and replicas might fail to receive messages from others. In this case, the president might not receive enough responses from other replicas, and replicas might miss the confirm messages, which leads to empty vote entry. Whenever a replica hits an empty entry in the vote databases and it cannot update its database, it sends a WANT request to the president to acquire the missing data. When a president receives a WANT request, it replies according to its database state or start the synod again if its database does not contain that particular entry of vote.

G. Running

Each running process maintains an instance database in memory. The synod information is stored in a hash table, which we will illustrate later.

1) Set: Whenever a replica receives a "set_value" request, it needs to notify other replicas of the update. If it is not the president, it directs the request to the president and let the president begin a vote and determine the instance number of the "set_value" request. If it is the president, it proposes a vote immediately. For example, if DGNS 2 receives a "set_value" request, it forwards it to the president to start a new synod (see Fig. 2).

2) Read Data: fast read and slow read: Packets are directed to replicas to improve the processing throughput. Since each replica has its own database, it can process "get_value" by reading locally. However, this does not guarantee the result is the newest because it is possible that a replica does not know the latest update. We may call this kind of read "fast read". Fig.1 shows that all replicas, including the president, receive requests from clients.
If a user requests for the newest value, we may process a "slow read" for him. This requires that all "set_value" requests before the slow read should be processed. This can be done by using a set command for a get instance.

H. View of service from clients

What clients can see is only the name service. They send requests and get responses. They do not know whether the service is distributed or not, so we should be able to direct requests to replicas with less waiting requests.

IV. IMPLEMENTATION

This part presents some implementation detail of the service. Configuration of the system values are presented in

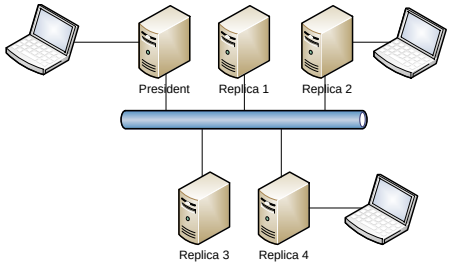

Fig. 1. All servers listen to the clients and replies to the client directly for "get" requests.

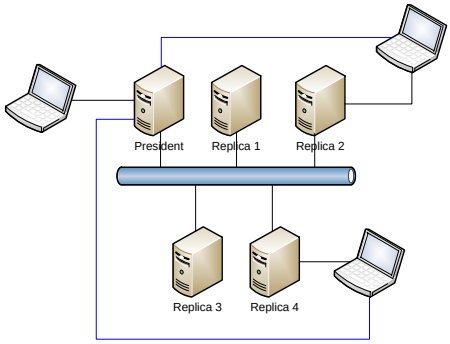

Fig. 2. All servers listen to the clients and will forward the message to the president when the requests are "set". After the president decided the synod value, it replies to the client directly.

Table II. The main state diagram for the president and the replicas are shown in Fig. 3 and Fig. 4, and Table III shows the action of participating threads.

A. Maintenance of process address table and status renewal of processes

When the service is started, processes load the configuration file with the IP address and port number into the IDtable. Processes need to renew their status continually. A replica sends heartbeat messages to others in every interval specified as RENEW_TIME.
To avoid "heartbeat messages" messing up with client requests, a thread is created just for renewing status and processing "heartbeat messages". A separated socket is opened for this

TABLE III
ACTIONS OF THREADS

Thread Name	Thread Action
dataquery	Fast read
pSynod(president only)	create a synod prepare messages to them
prepare	Read previous vote information and replies to the president
accept	Update the vote information and replies to the president
confirm	Update the vote log
Forward_msg(replicas only)	Forward the ”set” from clients to the president

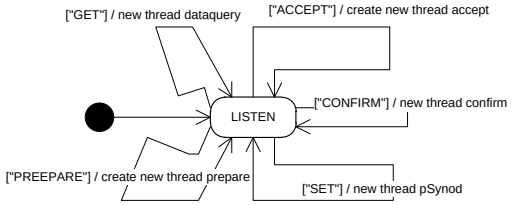

Fig. 3. President Main State Diagram

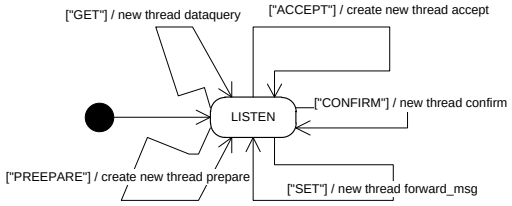

Fig. 4. Replica Main State Diagram

thread.

B. Initialization

When the service starts, all replicas are created with their unique identification numbers (replica IDs) set by the administrator. The replica with ID 1 (DGNS 1) is regarded as the president in the current implementation.
Different replicas have different ports to listen to clients. The base of the SERVER_PORT is 1700, and the port for a specific replica with identification number ID is SERVER_PORT+ID. Thus, distinct port for replicas with the same IP is guaranteed.
Inside the IDtable maintained by each nodes, every entry is an IDnode for a particular process with the IP address, port number and timestamp in precision to a microsecond. Assuming the clock error among processes to be around 0.6 second per week [8], we do not implement the synchronization of clocks of different processes. This might affect the performance of the IDtable when some processes fail to renew their status in time according to the time of another replica. This time error will not introduce error into the service because the outcome will only be false failure of peers.

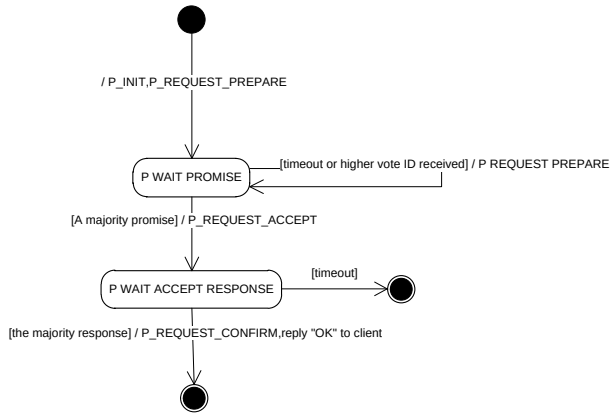

Fig. 5. Synod State Diagram

C. In memory hash table

To accelerate the speed for read and write, an in-memory hash table is implemented (modified from an on-line source). The tuple to be stored is in the form (name, attribute), name acting as a unique key and hashed to an ID which will be used to store and relocate the value.
The hash function is defined as follow, which is the original design of the on-line source. For a string S with n characters

$$s_1, s_2, s_3, s_4, ..., s_n$$

the hash ID is calculated as

$$HI = (s_1 \times 31^{n-1} + s_2 \times 31^{n-2} + ... + s_n) \bmod HASHSIZE$$

where HASHSIZE is a predefined value.
Moreover, the database of synods and confirmed vote information is stored in the memory as well. This avoids reading from and writing to hard disks frequently. The entries are not randomly hashed by a hash function; instead, they are stored in a table of linked list consecutively. For example, with size of the table to being 100, vote #1 will be stored in the first linked list, vote #2 in the second linked list, vote #101 in the first list, etc.

D. Programming Interface

DGNS presents its programming interface as a library.

1) Library at Client Side: The library is implemented in the file "service.c". Clients only need the lib file and "service.h" to use the service. Two functions are provided for the clients to contact the Generic Name Service servers, with self-explanatory names "set_value()" and "get_value()".

In the current version of the library, to communicate with a replica, the client needs to know the replica's IP address and port number. An example of IP addresses and port number of the replicas are shown in the Table II .

The API for the library are

```
int set_value(char* name, char* value, char* addr, int port);
int get_value(char* name, char* addr, int port, char res[]);
```

For example, we want to insert the pair ("almond", "tree") into the database, where "almond" is the name and "tree" is the value corresponding to "almond" to DGNS1. Then we use the function as

set_value("almond", "tree", "10.0.1.101",1701);

Later, we want to get the valueassociated with "almond". We use

get_value("almond", "10.0.1.101",1701);

Both functions return 1 if successful (the server replies with confirm information within a time frame), and 0 if not (no reply from the server, or no entry retrieved for get_value).

V. Evaluation

To provide a fast and reliable name service, we want to measure the performance of DGNS in terms of throughput on average for small number as well as large number of users. Two kinds of experiments are conducted; one is to measure the response time of the service with different size of database, and the other is to measure the response time for different size of using clients. The first experiment, namely sequential experiment, is to evaluate the throughput of the service, that is, the average time needed to satisfy a request when only one client is using the service. This experiment has one independent parameter, size of the database. We want to measure the influence on the service speed by enlarging database in the replicas. The second experiment, namely the concurrent experiment, is to measure the influence on the speed of response when a large number of clients are using the service. The success rate and response time for different number of clients are traced.

The evaluation is based on a predefined database with pairs of names and attributes. Every name or attribute is a string of characters less than 10 bytes, and the pairs are stored in a file from which the name and attribute are loaded during evaluation. When a client needs to send a request, it selects a pair of words and send the pair to the server to "set" the attribute associated with the name, or send the first element of a pair to "get" the attribute. The database contains different pairs of words depending on the configuration. The server consists of 5 replicas on 5 different virtual machines DGNS1 to DGNS5 (all on the physical host 10.0.1.24, with IPs 10.0.1.101, 10.0.1.102, 10.0.1.117, 10.0.1.118 and 10.0.1.119, respectively).

In the sequential request evaluation, multiple batches of 1k, 10k and 100k pairs of names and attributes are loaded into the database first and then selected to write into the server to produce different memory footprint. Fig 6 shows the percentage of requests satisfied within a time range. The majority are completed in less than 2ms, while some requests can take longer than 10000ms.

We also simulate different number of concurrent clients. The clients are on the same VM 10.0.1.102. In total, 5 VMs on the same rack are used to support the service. The clients send requests to one of the 5 VMs randomly each time. During an experiment, a user sends one "set" request. The experiments were conducted multiple times with 10 users and 100 users. Fig. 7 shows the results of the evaluations. From the client side, not all requests are fulfilled (see Table **??**), while from the server's view, they have processed all requests for the 1^{st} and 2^{nd} experiment. When the user size increases to be 1000, the president died twice.

The time spent on the server is also measured in details. In memory logging is applied to store the time and loaded after every synod. Since the requests are numbered after readClientRequest(), the timer starts when a synod recognizes the synod ID for the current vote, which is marked at time 0. Afterwards, the timer records the time spent since time 0. Fig. 8 shows the average execution time for different batches of sequential requests. The overall throughput of the service is shown in Fig. 9, which presents that the throughput is around 700 per second. Even with increasing number of processed requests, the throughput is quite stable.

Performance of fast read requests are shown in Fig 10. From the figure, we can see that the maximum time consumed for a non-null reply (the tuple with the key requested exists in the database) has a log-relationship with the total number of requests, while the average response time for both null and non-null replies stay around 500 microseconds. For the null replies, however, the maximum time increases to more than 3ms likely due to the limited current hash size.

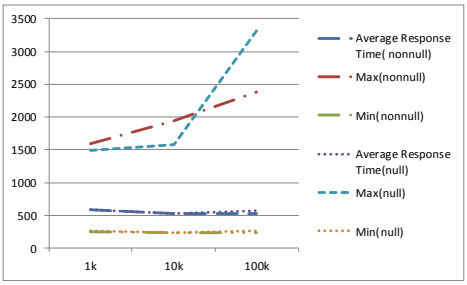

Fig. 10. Time cost of sequential "get" requests

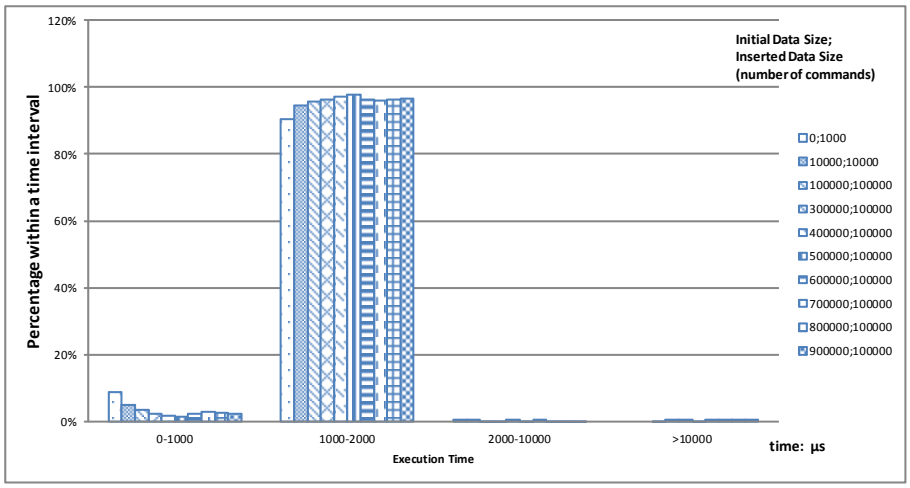

Fig. 6. Time cost of different size of sequential "set" requests. Different batches have different initial data sizes in memory as well as different data sizes to be inserted.

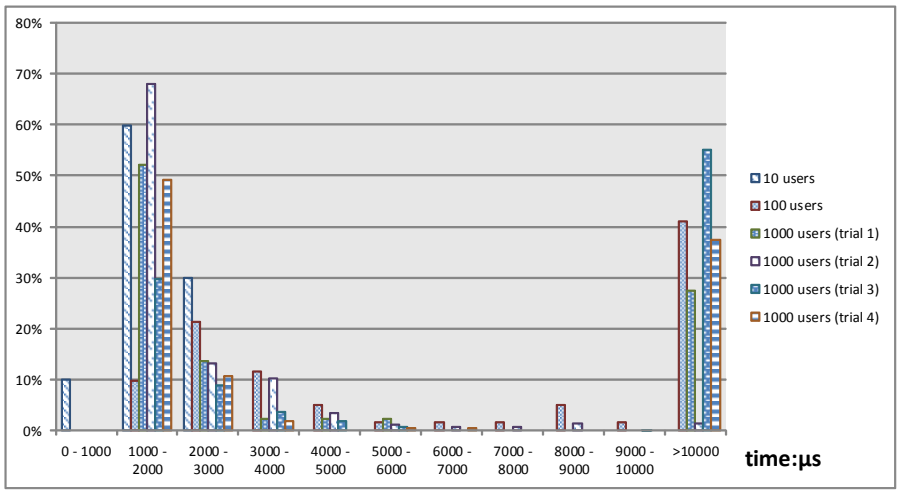

Fig. 7. Time cost of different number of users sending "set" requests.

VI. Conclusion

This report presents the design and implementation details of the Distributed Generic Name Service. Two major experiments on virtual machines have been conducted to evaluation the performance of the service. Further experiments on physical machines will be conducted.

References

[1] Ammar Benabdelkader, Hamideh Afsarmanesh, and Louis O. Hertzberger. Megastore: Advanced internet-based e-commerce service for music industry. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, DEXA '00, pages 869–878, London, UK, UK, 2000. Springer-Verlag.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[3] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[4] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[5] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[6] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[7] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 105–120. USENIX, December 2004.

[8] Hicham Marouani and Michel R. Dagenais. Internal clock drift estimation in computer clusters. *J. Comp. Sys., Netw., and Comm.*, 2008:9:1–9:7, January 2008.

[9] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered

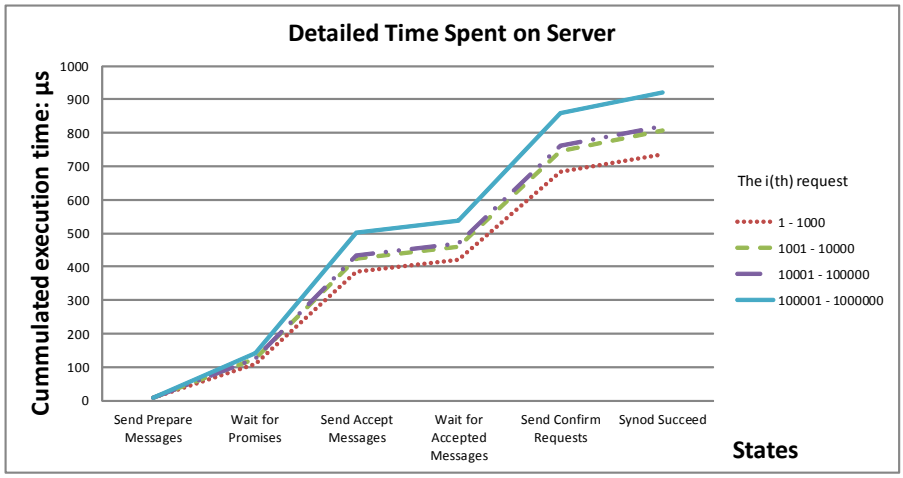

Fig. 8. In memory time measured for the president

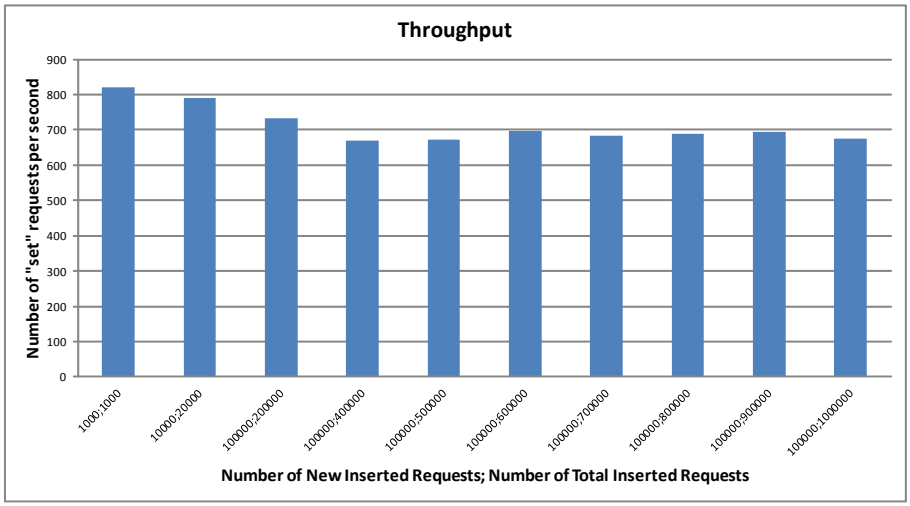

Fig. 9. Throughput of the service when the various inserted number of requests

broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, New York, NY, USA, 2008. ACM.