Efficient Atomic Broadcast in Cloud Computing

Po-Jen Chuang and Wei-Ming Hsu

Department of Electrical Engineering
Tamkang University
Tamsui, New Taipei City, Taiwan 25137, R. O. C.
E-mail: pjchuang@ee.tku.edu.tw

Abstract. Atomic broadcast protocols, such as libPaxos [1], can help ensure transmission consistency in cloud computing services. To upgrade the performance of libPaxos, this paper presents a new protocol which divides the topology into areas to process messages in a distributed way and uses a rotating mechanism to effectively balance the server loads. Simulation results show that, when compared with existing protocols, the proposed protocol yields higher throughput and lower latency under different client loads.

Keywords: Cloud computing, atomic broadcast protocols, consistency, experimental performance evaluation.

1 Introduction

Today's internet services need to handle extensive traffic loads because users not only read the information but also become information contributors, producing massive writings. To cope with the situation, service providers usually involve multi-layered caching backup mechanisms to facilitate data access and upgrade system scalability. When facing large-scale, high-concurrency community network services, they tend to manage by *powerless* approaches which may require higher performance, storage, scalability or availability. For internet servers, how to handle the huge amounts of messages has become a major challenge.

A new type of database NoSQL (Not only SQL) [2], featuring high scalability and flexible storage formats, is designed to handle the colossal data so as to ensure service availability. It will ensure that each database maintains the latest information and messages do not conflict with one another. To maintain transmission consistency for cloud computing services, we can involve atomic broadcast protocols such as libPaxos, Mencius or RingPaxos. libPaxos [1], which implements the Paxos algorithm [3, 4], executes atomic broadcast in distributed applications. Consisting of clients, proposers, acceptors and learners, libPaxos can maintain desirable data consistency and performance in distributed architectures but needs to send frequent messages from proposers to acceptors and clients. When data build up, messages are likely to over-concentrate on a proposer, degrading the overall performance of the mechanism (because handling messages over-concentrated in one proposer takes excessive bandwidth and network loads). Mencius [5] adopts Paxos and multi-proposers [6] to

© SERSC 2013

distribute proposer loads. It lets each server act as a proposer to share the load. By reducing the excessive load on one proposer, it enhances the overall performance. The problem is, when both the message load and bandwidth are constant for proposers, performance improvement may cut back when proposer rotation is used to avoid excessive load concentration. RingPaxos [7] uses the ring topology to solve the proposer load concentration problem. It can distribute the proposer load and contain bandwidth consumption, but when an acceptor fails, it has to wait until the acceptor gets repaired in the round, decreasing the overall transmission efficiency.

To upgrade the performance of libPaxos, this paper introduces a new protocol which divides the topology into various areas to process messages in a distributed way and meanwhile uses an effective rotating mechanism to balance the server loads. Simulation results show that, when compared with other protocols, our new protocol generates constantly better throughput and less latency under different client loads.

2 The Proposed Protocol

Besides the multi-proposer concept, our new protocol brings in the concept of partitioning to handle each message and avoid extreme load concentration on a single server. The new protocol, the Area-based RingPaxos (ARP), partitions the ring topology into multiple areas. It uses *multi-proposers* to reduce the probability of message congestion and *consistent hashing* to implement the concept of partitioning. 3 proposers will perform the Paxos algorithm in 3 areas to collect message statistics, and each proposer will take turns acting as the leader proposer to balance load capacity. The basic principle of *consistent hashing* [8, 9] is that server nodes and keys (based on servers' IPs) are mapped to $0 \sim 2^{32}-1$ positions of the ring structure following the same hash algorithm. When a *write* request comes in, calculate the key IP's corresponding hash value: If corresponding exactly to a server's hash value, write directly to the server; if not, go clockwise to find the next server to write. Restart from 0 when finding no corresponding server after hash value $2^{32}-1$; continue clockwise to find the nearest server when a found server crashes. It can effectively reduce the server load or efficiently add/ remove any servers to balance the server load.

In our protocol, a server is placed under each area to act as an acceptor to transmit *prepare* and *deliver* messages (messages used to maintain data consistency), handle client requests and send them to the proposer in charge of the area. A proposer will collect client messages in its area and send them to the leader proposer (by sending *promise* and *result* messages to the acceptor or *discuss* messages to the leader proposer). The leader proposer then handles *discuss* messages received from other proposers and returns *agree* messages to ensure data consistency in different areas.

ARP divides *message processing* into 2 phases. In Phase 1a, when an acceptor receives a client request, it sends a *prepare* message to the proposer in charge to check the round number and ballot number, to make sure they are not repeated or overwritten (to avoid information inconsistency). After receiving the two numbers, the proposer will confirm if they are consistent with the current running round: If yes, return the latest *promise* message to the acceptor and end Phase 1; otherwise, ask the acceptor to resend the *prepare* message. In Phase 2a, the acceptor checks the *promise* message from the proposer and sends a *deliver* message (with the round and ballot numbers

promised by the proposer) along with the client request to the proposer. The proposer then sends a *discuss* message to the leader proposer which compares the message with messages from other proposers. When a message conflict happens, the leader proposer will check if the numbers are consistent with the latest values. If not, it will proceed to the next step. When the leader proposer sends the *agree* message to the other proposers in Phase 2b, data consistency in different areas is achieved. After receiving the *agree* message from the leader proposer, the proposer sends a *result* message to the acceptor which then sends the final resolution to the client.

In our protocol, when a proposer crashes, the acceptor will wait for the proposer's message to the end of the specified time limit, and then replace it by a neighboring acceptor. When a leader proposer crashes, the other proposers will wait until it recovers. An acceptor crash will not affect the operation of our protocol because we do not let a client send a request to a crashed acceptor; instead, we will lead the client request to another acceptor selected by *consistent hashing*.

Table 1. Simulation parameters

	Experiment Setting
Number of values	30(concurrently)
Value sizes	300、1000、2000、4000(bytes)
Number of nodes	4
Number of simulation	10
OS	Fedora 8

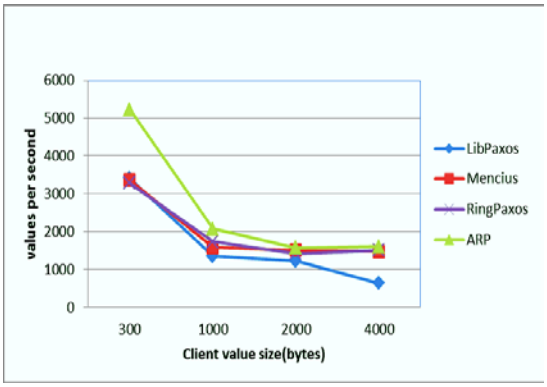

Fig. 1. Values per second vs. client value sizes.

3 Experimental Performance Evaluation

Experimental evaluation using the DETER Testbed [10] has been carried out to compare the performance of APR, LibPaxos, Mencius and RingPaxos in terms of

values per second (*vps*), *kilobytes per second* (*kbps*) and *latency*. Table 1 lists the simulation parameters. The results are obtained over a large number of values, specifically when clients send concurrently 30 values per 0.1 second. We measure the data size handled by each protocol from the client side (receiving all final resolutions), and use the maximal handled data size as the performance indicator.

Figure 1 depicts *values per second* (*vps*) for different client value sizes. Mencius and libPaxos give almost the same results at smaller value sizes when the required processing data and bandwidth usage are containable. At bigger value sizes, we detect smaller *vps*, i.e., degraded processing efficiency, because the server needs to do extra *save*. When the value sizes grow bigger, Mencius yields higher *vps* than libPaxos because its leader rotation mechanism avoids message concentration on a leader. The ring structure in RingPaxos reduces load buildup on a proposer and helps handle more client messages. At higher value sizes, our ARP produces close results with Mencius and RingPaxos, but at lower value sizes it produces the highest *vps* (by partitioning the ring into areas to reduce bandwidth requirement between proposers and clients or acceptors).

Figure 2 depicts *latency* (in milliseconds) vs. client value sizes. Latency is defined as the time duration when a client sends a request till it receives the final resolution. Each result is the average from 500 samples (client requests). The results show a reasonable performance trend: when the value sizes grow, latency increases for all protocols. libPaxos produces remarkably higher latency than others because it has only one proposer to deal with client requests. Mencius has much shorter latency than libPaxos because its multi proposers share the message load and speed up message processing. When client value sizes grow over 2000 bytes, RingPaxos has less latency than Mencius as it employs the ring structure to carry out message transmission. Our ARP, which uses acceptors to receive client messages in order to share/ease the burdens of proposers, yields the least latency (we use the various roles in the system to equally share the proposer loads and raise the processing speed). When client value sizes > 2000 bytes, latency of ARP comes very close to that of RingPaxos – this is because our topology for message transmission basically resembles the ring structure in RingPaxos. The overall advantage, nevertheless, goes to ARP thanks to its special topology partitioning mechanism.

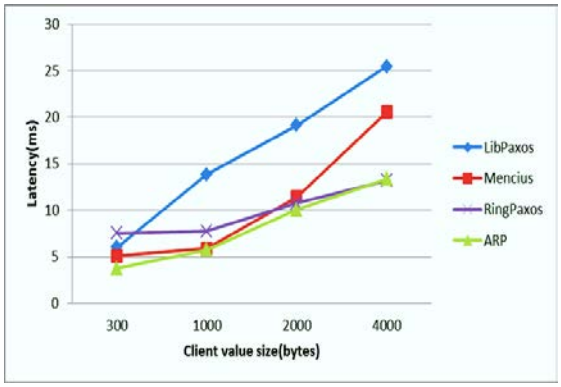

Fig. 2. Latency vs. client value sizes.

5 Conclusions

This paper presents an Area-based RingPaxos (ARP) atomic broadcast protocol to help ensure data consistency for cloud computing services. ARP is distinct in dividing the ring structure into various areas to process client messages in a distributed way and in using a rotating mechanism to effectively balance the server loads. Simulation results show that the two unique features enable the new protocol to outperform existing protocols in terms of throughput and latency, under different client loads.

References

1. Primi, M.: Paxos Made Code. Master thesis, Informatics of the University of Lugano (2009)
2. NoSQL, http://nosql-database.org/.
3. Lamport, L.: The Part-time Parliament. ACM Transactions on Computer Systems 16 (2), 133--169 (1998)
4. Lamport, L.: Paxos Made Simple. ACM SIGACT News 32 (4), 18--25 (2001)
5. Mao, Y., Junqueira, F., Marzullo, K.: Mencius: Building Efficient Replicated State Machines for WANs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 369--384 (2008)
6. Lamport, L., Hydrie, A., Achlioptas, D., Multi-leader Distributed System. U.S. patent 7 260 611 B2 (2007)
7. Marandi, P.J., Primi, M., Schiper, N., Pedone, F.: Ring Paxos: A High-throughput Atomic Broadcast Protocol. In: 2010 Dependable Systems and Networks, pp. 527--536 (2010)
8. Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y.: Web Caching with Consistent Hashing. In: 8th International Conference on World Wide Web, pp. 1203--1213 (1999)
9. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: 1997 ACM Symposium on Theory of Computing, pp. 654--663 (1997)
10. Benzel, T., Braden, R., Kim, D., Neuman, C., Joseph, A., Sklower, K., Ostrenga, R., Schwab, S.: Design, Deployment, and Use of the DETER Testbed. In: 2007 DETER Community Workshop on Cyber-Security and Test, pp. 1--1 (2007)