AR-TCP: Actively Replicated TCP Connections for Cluster of Workstations

Zhiyuan Shao[1], Hai Jin[1], and Jie Wu[2]
[1]*Huazhong University of Science and Technology, Wuhan, 430074, China*
Email: {zyshao, hjin}@mail.hust.edu.cn
[2]*Florida Atlantic University, Boca Raton, FL 33431*
Email: jie@cse.fau.edu

Abstract

In this paper, we propose a novel scheme, called Actively Replicated TCP (AR-TCP), which transparently improves the service and data availability of cluster systems at TCP connection level. In this scheme, TCP connections are replicated and synchronized among all workstations of a cluster and can failover to healthy parts during failures. Moreover, request messages resulting in data exchang are delivered by the workstations atomically to guarantee data consistency. Read-only request messages are extracted and executed on only one of the replicas to improve the performance of simultaneous access. As an application, we build a fully replicated MySQL cluster based on our proposed scheme. The experimental results of the prototype implementation show that the cluster has small performance penalty on communication, high simultaneous read-only query performance and a small performance penalty on update operations.

1. Introduction

As a reliable point-to-point transport level protocol, TCP has been gaining more and more users on the Internet. Years of enhancement and fine-tuning have made it very efficient and robust. However, when one of the two peers crash, it is still not reliable enough to keep connections alive on-the-fly so as to mask server failure from the client.

Unfortunately, fault-tolerance of the TCP protocol is becoming increasingly important for many applications. For example, many organizations and enterprises (e.g., Google) enhance their throughput by clusters whose availability is usually guaranteed by using a front-end approach. This approach employs software packages (e.g., LVS [10]) or industry solutions (e.g., Cisco LocalDirector) as dispatchers to direct TCP connections towards the back end real servers, and guarantees service availability by avoiding new connections being directed to crashed nodes. However, the existing connections processed by the failed server will simply be lost, and thus expose clients to connection failures.

In order to solve this problem, many researches [2][9][11][16][18] have been conducted in the past few years. FT-TCP [2] uses a logger to record the on-going connections and reincarnates the connections of the crashed server by replaying the log on a new server. However, this solution introduces another single point of failure (i.e., the logger), and it is also time-costly during failover. To overcome the shortcomings of FT-TCP, ST-TCP [11], HARTS [9] and LW-HARTS [7] adopt the primary-backup approach, which replicates the TCP connections on-the-fly on multiple replicas. However, ST-TCP tolerates only single failure and requires identical processing speed in the replicas (which is unrealistic in the real world). HARTS and LW-HARTS, which are our previous research works, usually greatly sacrifice performance in communication when there are more than two replicas. Furthermore, almost all of these approaches are proposed to enhance service availability while the data availability, which is the prerequisite of the former, always remains unaddressed.

In this paper, we propose a novel scheme, named AR-TCP, which extends the research on fault-tolerant TCP to allow atomic multicasting at the transport layer of communication, and employs active replication techniques to realize data availability in clusters. By conducting experiments on a MySQL server cluster by using AR-TCP, we observe almost linear improvement in the performance of simultaneous read-only queries and a small penalty on the update operations.

The rest of this paper is organized as follows. Section 2 briefly surveys the related works. Section 3 introduces the system architecture of our research and states the problems studied in this paper. Section 4

Proceedings of the Japan-China Joint Workshop on
Frontier of Computer Science and Technology (FCST'06)
0-7695-2721-3/06 $20.00 © 2006 IEEE

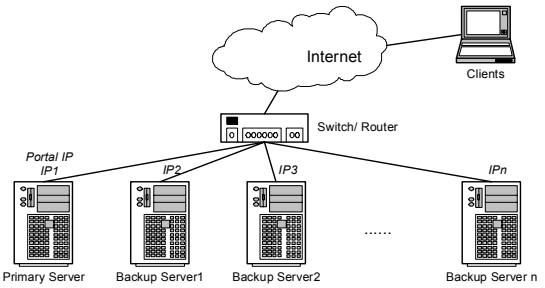

Figure 1. System architecture

explains our method. Section 5 addresses two unique problems in AR-TCP, i.e., the TCP sequence number translation and failover. Section 6 presents the experimental results of the prototype implementation. Section 7 concludes this paper.

2. Related Works

Atomic multicasting [1][3][8] and view synchronous communication [13][14] are two important communication abstractions that have been extensively studied in the context of fault-tolerant distributed systems. However, besides the disadvantages for practical applications such as heavy weight at the processors, prolonged delivery time, and complexity, both of these two abstractions take UDP as their basis. This inevitably jeopardizes the transparency if they are applied on legacy applications which employ TCP.

Active and semi-active replication techniques [17] provide strong data consistency among copies. The techniques from the former class require the replicas to deliver request messages atomically, and the responses of the replicas are sent directly towards clients. Techniques belonging to the latter class require the replicas to deliver request messages atomically, and use view synchronous communication to gather the responses so as to form a unique response back. Both classes of replication techniques require atomic multicasting as their prerequisite to distribute the request messages, thus suffer the compatibility problems mentioned above.

Many TCP fault-tolerance schemes [2][9][11][16][18] have been proposed in the past few years. Most of them are implemented by providing a primary server that actually handles the connections with one or several active and fully replicated backups. However, these schemes suffer some common drawbacks. For example, long failover time [2], unreasonable assumption on the processing speed of replicas [11], and much sacrifice on communication [9]. Moreover, the backup servers within these schemes are

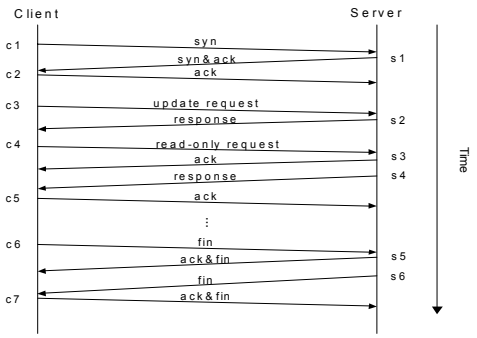

Figure 2. The message exchange flow of a typical TCP connection

simple followers and thus waste their potential processing capacities.

3. System Architecture

AR-TCP adopts the cluster architecture with share nothing semantics shown in Figure 1. Among the server nodes, there is a unique primary server and multiple backup servers. The primary server possesses the portal IP address of the cluster. All server nodes in the cluster have their own IP addresses (IP_1, IP_2,…, IP_n), which belong to the same subnet as the portal IP.

In this paper, we only consider the legacy client/server mode applications that adopt the event-driven model for serving connections, by which request messages are delivered in the order of receiving.

For convenience of discussion, we make the following assumptions. First, we assume the executions of an application on the server nodes are deterministic and all copies of this application respond identically to the same request. In AR-TCP, all the requests are delivered in order, and uncertainties due to concurrency are precluded to make this assumption reasonable. Second, we assume the application protocol is interactive. That is, the client must wait until obtaining the response from the preceding request from the server before sending a new one. Third, we assume that a request message received by a server node will be delivered if no crash failure happens, and the delivered request will be processed. Finally, as our scheme can adopt any independent failure detector, in order to simplify the discussion, we assume the failure detector used in our scheme is perfect [5].

In this paper, we consider only crash failures, and assume the network is always available and will not be partitioned. Messages sent from one server node to another will eventually arrive at its destination if a time-out based retransmission mechanism is adopted.

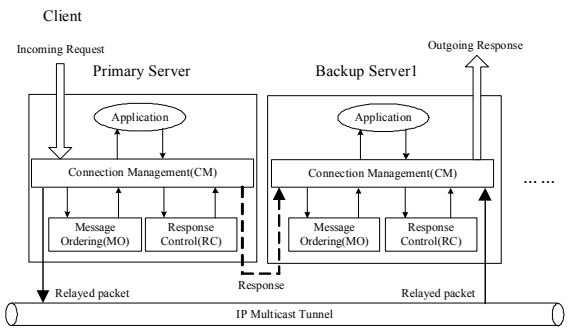

Figure 3. Communication Paradigm of AR-TCP

4. Proposed Methods

We start the discussion by classifying the incoming requests. A typical message exchange flow of a TCP connection used by legacy applications is illustrated in Fig. 2. We regard all the messages sent from the client to the server as requests, and catalog them into four classes: *connection related request*, *update request*, *read-only request* and *pure ACK*. Connection related requests are TCP control messages (e.g., SYN, FIN etc.). Packets *c1*, *c6*, *c7* in Fig. 2 belong to this class. Update requests are the messages sent by the clients in order to change the data or status of the server. In Fig. 2, packet *c3* is an update request. Read-only requests are the data access messages, which do not change the data of the server, e.g., packet *c4* in Fig. 2 is a read-only request. Pure ACKs are simple acknowledgements or keep-alive messages that have no payload. Packets *c2* and *c5* in Fig. 2 belong to this class. AR-TCP differentiates update requests from read-only requests by parsing the message content. In real applications, the request may be large and thus fragmented into pieces to be transmitted via the network. In this case, we call the request as a whole a request message, and the individual pieces request packets. For convenience, we call both connection related and update requests causal requests.

In order to replicate the connections, the server nodes should receive all of the incoming packets. Although this objective can be easily achieved by programming the switch [11], it is difficult to guarantee the atomicity of delivery. AR-TCP adopts a new communication paradigm for TCP connection shown in Fig. 3. In this paradigm, each server node consists of a *Connection Management* (CM) module, a *Message Ordering* (MO) module and a *Response Control* (RC) module.

The incoming request messages will arrive first at the primary server. The CM module of the primary server will conduct a legal check on the individual packets of these messages. A request packet will pass

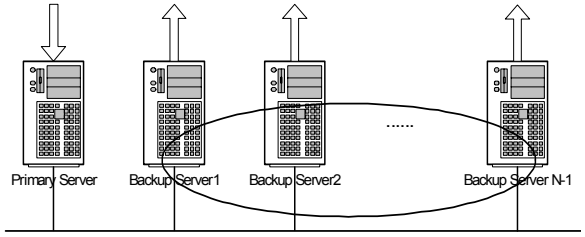

Figure 4. Round-Robin Response

the check provided it belongs to an established connection (or a *SYN* packet that initiates a connection), and its sequence and ACK numbers are correct. After that, the message will be parsed and handled according to their style. We will discuss the mechanisms used to process different kinds of requests in the following. Before discussion, we assume the *initial sequence number*s (ISNs) of the TCP connections are synchronized during establishment, which has been implemented in our previous research work [9].

4.1. The Causal Requests

Before relaying this kind of request messages, the MO module of the primary server will assign them an ordering number, which grows monotonically and re-folds at a bound. Packets of the same request message will be given the same number. IP multicast tunneling is used to propagate the request messages so as to improve efficiency. After having received these request messages, the CM module of the backup server will assert legality of these request packets, and the MO module will check the ordering number. Request messages will eventually be delivered to the upper layer application at each server node. In AR-TCP, we defined two strategies to control the delivery of request messages, namely *Best-effort* and *Safe*.

In the best-effort delivery strategy, all the server nodes of the group deliver the request message immediately after the received message has passed the legality and ordering check. The requests are delivered in the monotonically increasing order. Missing requests are discovered by detecting the gaps on ordering numbers, and NAK messages are used to ask the primary server for retransmission. As other sequencer-based protocols [3][8], this delivery strategy achieves short broadcast delivery time.

Safe delivery strategy requires all backup servers to send out positive ACKs containing the ordering number of every request message they have received. The reliability of these positive ACK messages is guaranteed by a time-out mechanism. Like the best-effort delivery strategy, the incoming request messages are delivered by strictly increasing order at all server

nodes. The difference is that any server node can only deliver a request message after it has gathered the positive ACKs from all backup servers in safe delivery strategy (i.e., it is *stable*).

With this strategy, if a request message is lost at some server nodes, the rest of the server nodes can only delivery this message after the message is eventually received by those who missed it. If one of the server nodes crashes, the failure detector will confirm the failure and exclude the server node from the cluster, then awake the remaining server nodes. Hence, the all-or-nothing property of communication is satisfied. It is easy to prove that safe strategy guarantees the atomicity of message delivery for the causal requests.

Responses to these causal request messages will first be intercepted by the CM modules and then given to the RC modules for processing at each server node. After accepted of the response packets at the RC module of the server nodes, their response numbers will be obtained. AR-TCP computes the response number by comparing the ACK number of the response packets with the sequence numbers of the causal request packets in history. In this way, the response numbers *s1*, *s2*, *s3*, *s4*, *s5* and *s6* in Fig. 2 corresponds to the ordering numbers of *c1*, *c3*, *c4*, *c4*, *c6* and *c6*, respectively. Pure ACK responses (e.g, *s3* in Fig. 2) are simply ignored by the server node to help the aggregator to identify the real responses. These response messages will converge (the dash line in Fig. 3) at a functionally chosen backup server (e.g., backup server1 in Fig. 3.), which is named as aggregator, in the form of a UDP packet. The latter will decide the final version after having gathered all response messages of the same response number, and then send it back to the client directly by socket rewriting [4].

Consider the *N-nodes* cluster configuration with *N*-1 backup servers. AR-TCP makes all the backup servers in the cluster work as the aggregator in a round-robin fashion shown in Fig. 4. Each server node has an individual rank number of *k*. Rank numbers of the backups can be sorted in an increasing order, which falls in the range of [1, *N*-1] and is named as response rank number. AR-TCP designates the server node with response rank number of (x%(N-1))+1 for the response messages, whose response numbers are x. As response numbers increase roughly continuously, the duty of aggregator will be distributed among the backup server nodes evenly.

4.2. The Read-Only Requests

AR-TCP processes read-only requests on different replicas in parallel to improve the performance. After receiving the read-only requests, the primary server will select one server node from the cluster in round-robin style to serve. The choosing of server nodes include the primary server (RR-P) or not (RR-NP). After choosing a server node, whose rank number is *k*, the primary server will append the read-only request message with a *scheduling header* including *k*. In order to avoid retrieving stale data, the most recent ordering number of the causal requests is also stored in the scheduling header.

Having received the relayed read-only requests from the tunnel, server nodes will look into the scheduling header to obtain the rank number, and compare that with theirs. If they match, the server node will continue to deliver the request after delivering the causal request with higher or equal ordering number than that in the scheduling header. We call this kind of read-only requests as *duty read-only request messages* for that server node. The response packets for such requests can be sent back directly to the client without aggregation. On the contrary, we call the read-only request messages that do not need to be processed *negligible read-only request messages*. If a server node receives such request packets, it will use the information (i.e., sequence and ACK number in TCP header) of the packets to keep track of the connection and then have them discarded.

4.3. Pure ACK Requests

In most cases, the pure ACK request packets are the last phase of TCP handshake (e.g., *c2* in Fig. 2), confirmation of acceptance (e.g., *c5* in Fig.2), or simple keep-alive messages of the connection. If they are not propagated to all the server nodes, the connections will malfunction.

Although pure ACK request packets should be received by all the server nodes (different from the read-only request), the reliability of the propagation is not mandatory (different from the causal requests), since the built-in retransmission nature of the original TCP is enough, even though some server nodes may receive the same ACK request packet several times or out of order. In AR-TCP, the primary server will give a special ordering number to the intercepted pure ACK request packets, and then have them relayed to the backups even though they are retransmitted from the view point of connection.

5. Unique Problems

With the introduction of ROWA strategy in AR-TCP, server nodes are required to serve the incoming requests selectively, and thus the TCP sequence-hole problem is an inevitable result. In the following

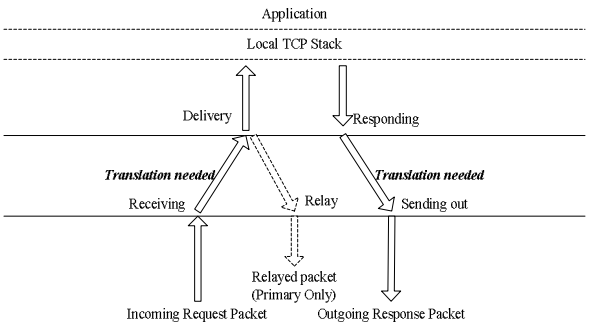

Figure 5. Typical Procedure of Packet Processing at the Server Node

section, we will discuss this problem and give a solution. The methods used to address the possible failures of server nodes are also discussed in this section.

5.1. TCP Sequence Number Translation

Before being delivered by the server nodes, a request packet should undergo proper translation to make it legal for the local TCP stack. A response packet should also be translated so as to make it acceptable by the clients. Fig. 5 shows the typical procedure of translation at one of the server nodes. We will discuss the translation mechanism according to these two naturally separated stages.

5.1.1. Stage one: from receiving to delivery. We denote the request packet as <*Req_CSN*, *Req_CAN*, *Req_LEN*>, where *Req_CSN* is the client sequence number, *Req_CAN* is the client ACK number and *Req_LEN* is the length of the request packet. We denote the packet after translation as <*Req_LSN*, *Req_LAN*, *Req_LEN*>, where *Req_LSN* is the local sequence number and *Req_LAN* is local ACK number.

Translating the TCP sequence number of the request packet is relatively easy. Let $\sum Req\_LEN\_NRO$ be the sum of the length of negligible read-only request packets received by the server node. Since negligible read-only request packets are extracted from the stream of the connection and processed by other server nodes of the cluster, their length should be subtracted, we have:

$$Req\_LSN = Req\_CSN - \sum Req\_LEN\_NRO \quad (1)$$

If the request packet is a new duty read-only request packet or a new causal request packet, and *CURR_SEQ* is the current sequence number of the node. Since the communication is interactive, if the server node receives a later request, the client must have received all the responses for the preceding requests, and *Req_LAN* should equal to *CURR_SEQ*.

However, if the server node receives a pure ACK, duplicated duty read-only or causal request packet, computing the local ACK number of the request packet is a little complicated. In this case, if the server node is processing a duty read-only or causal request packet when receiving the request packet, we call the packet under processing the base request packet, and denote it as <Req_CSN_{base}, Req_CAN_{base}, *Req_LEN*>. The local ACK number for delivering the base request packet is denoted as Req_LAN_{base}. The local ACK number of current request packet can be computed by the following equation:

$$Req\_LAN = Req\_LAN_{base} + (Req\_CAN - Req\_CAN_{base}) \quad (2)$$

Since having received such request packets does not necessarily mean that the client has received all the response packets for the base request packet before that, *Req_LAN* should be computed by adding Req_LAN_{base} to the increment of the client ACK number. However, if the pure ACK or duplicated duty read-only or causal request is received when processing a negligible read-only request message, the local ACK number still equals to *CURR_SEQ*.

5.1.2. Stage two: from responding to sending out. We represent the response packet of the server node as <*Rsp_LSN*, *Rsp_LAN*, *Rsp_LEN*>, where *Rsp_LSN* is the local sequence number, *Rsp_LAN* is the local ACK number and *Rsp_LEN* is the length. We denote the packet to be sent out as <*Rsp_CSN*, *Rsp_CAN*, *Rsp_LEN*>, where *Rsp_CSN* is the sequence number, and *Rsp_CAN* is the ACK number.

As the response of a server node will be sent back to the client when it is processing a duty read-only or causal request, i.e., base request, let $\sum Rsp\_LEN$ be the sum of the length of all response packets for the base request packet sent out before, *Rsp_CSN* and *Rsp_CAN* can be computed by the following equations:

$$Rsp\_CSN = Req\_CAN_{base} + \sum Rsp\_LEN \quad (3)$$

$$Rsp\_CAN = Req\_CSN_{base} + Req\_LEN \quad (4)$$

Since the server node has already delivered the request, it is reasonable to confirm with the client by using ACK number.

A TCP sequence number translation mechanism is designed to work in the CM module shown in Fig. 3. All request packets will be processed by the MO module after the translation, and all response packets are to be processed by the RC module without translation. By this way, standard TCP connections are imitated from the viewpoints of MO and RC. This implies the employed translation mechanism does not affect the message ordering system and satisfies all the three requirements.

5.2. Failover

We consider two typical types of failures in this paper: the failure of a backup server and that of the primary. Crash failure of one of the backups will make the cluster stop working temporarily, since the rest of the healthy server nodes cannot receive the positive ACKs or the response packets from the failed backup. The system continues to work until diagnosing the result of the failure detector wakes up the waiting nodes. The backup server immediately after the failed one in the round-robin ring (see Fig. 4) will be chosen to ensure roles for aggregating of the failed one. The retransmission mechanism of TCP assures that the response packets will eventually reach the new aggregator. The rank number of the failed node will be reclaimed and a new ring will be formed. Allocating the duties of aggregator for the communications after the failure should comply with the new ring.

Our scheme handles the failure of the primary by electing a new primary server among the healthy backups. The one with highest ordering number will be chosen as the new primary so as to keep the existing ordering number of causal request messages. If more than one backup satisfies this criterion, the one with the smallest rank number will be elected.

6. Performance Evaluation

To evaluate the scheme discussed in this paper, we implement a prototype system on a cluster of up to four server nodes, and conduct experiments on that. In the first subsection, we will present and analyze the experimental results to discuss the penalty on communication. In the second subsection and as an application, we discuss the performance of the MySQL server cluster built by using our scheme.

The server nodes of the cluster are PCs running on Redhat Linux kernel version 2.4.7-10 with hardware of Intel Pentium III 1GHz CPU, 512MB Memory and 100Mbps Intel EEPro NIC. The client machines are PCs running on Windows 2000 Professional (service pack 4) with hardware of Intel Celeron 1.7GHz CPU, 512MB Memory and RTL8139A NIC. We use 3COM 100Mbps switch to connect the clients and the server nodes. The PCs run our programs almost exclusively. We use MySQL server 3.23.41 in the experiments. MySQL server of higher versions (e.g., 4.0 or later) is not used, since they have huge query cache, which favors repetitive queries, and may affect the simulations made in our experiments.

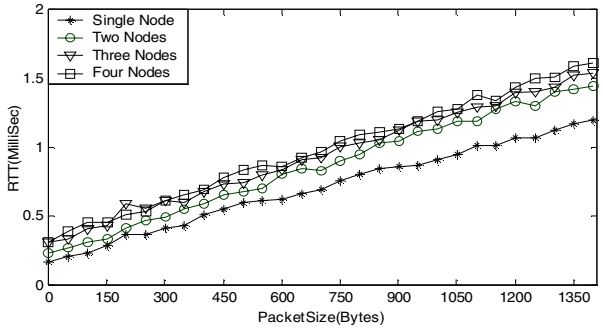

Figure 6. Penalty on Communication Performance

6.1. Communication Penalty

In Fig. 6, we compare the performance of TCP connections of AR-TCP with that of standard TCP. The performance is evaluated by Netpipe-2.4 [15] with different numbers of server nodes. The *round trip time* (RTT) between the client and the cluster is used to demonstrate the latency of communication. In the experiments, a best-effort delivery strategy is used.

From Fig. 6, we observe that the latency of a two-node cluster increases about 15% more than the standard TCP. This latency increase is due to the extra time used in ordering and relaying the incoming packets. However, the latency of communication does not increase significantly as the replica number increases from 2 to 4. Since IP multicast tunneling is used in AR-TCP, each incoming packet needs to be relayed once in most cases. Furthermore, round-robin response mechanism distributes the load for responding evenly among the backups.

6.2. Performance of MySQL Cluster

As an open source database management system, MySQL [12] has been gaining more and more users around the world, most of which are web sites. In common installations, it is used as a backend server, which provides data to the real server nodes of the cluster. However, when the visit rate of the web site continues to increase, the load of MySQL server soars. Single servers cannot always handle the load. Crash failure of the MySQL server will result in a disaster within these installations. Built-in fault tolerant schemes (e.g., cluster) of the MySQL cannot prevent connections from being lost.

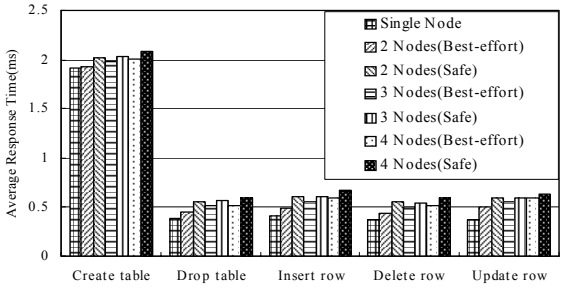

Figure 7. Performance of MySQL Cluster on Update Operations

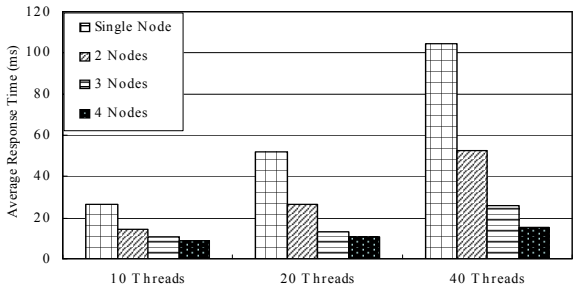

Figure 8. Performance of MySQL Cluster on Simultaneous Read-only Queries

We deploy AR-TCP in the OS kernel of the server nodes on which MySQL server is installed and the clients connect to the cluster with MySQL ODBC driver version 3.51.10. In this experiment, "SHOW", "SELECT", "EXPLAIN SELECT" queries are cataloged as read-only requests, while the other query requests are regarded as update operations.

6.2.1. Performance of update operations. First, we conduct experiments to study the impacts of our scheme on the performance of MySQL update operations. Fig. 7 shows the performance of such operations. Performance data of Insert, Delete and Update are obtained by conducting such operations on a test table, which contains ten integer fields.

Fig. 7 shows that the creating of a table is the most time consuming among the five operations. This is because MySQL server creates new files to hold newly created tables. The other update operations cost less time since the file is always open before operation. A smaller sacrifice on performance of update operations, which consume less time, than those consuming more time (i.e., create table) can be observed from Fig. 7, since the communication penalty can be better masked by the time consumed on the operations. This further means, to the complex operations (e.g., updates on multi-table), the sacrifice will be small.

Fig. 7 also shows that the update performance of the best-effort delivery strategy is always better than that of the safe strategy. The longer response time of the safe delivery strategy is caused by the prolonged message delivery time. However, the difference between these two strategies is not too much. Since our scheme is built in the OS kernel, positive ACKs can be sent out faster than by using application level programs. This results in smaller performance loss.

6.2.2. Performance of simultaneous read-only queries. We invoke multiple threads at the client side to evaluate the performance of the MySQL cluster when processing simultaneous read-only queries. Each thread loops for 1000 times, where each loop is a single "SELECT" query statement to retrieve 100 rows from a test table. We call such a thread a select thread for convenience. The performance of the MySQL cluster is evaluated with the workload of 10, 20 and 40 select threads. In the 10-thread case, all threads are invoked at a single client machine. In the 20-thread case, two client machines are used and each of them simultaneously invokes 10 threads. The 40-thread case is similar to the 20-thread case except that each client invokes 20 select threads simultaneously. Fig. 8 shows the performance of the cluster when processing simultaneous read-only queries. In these experiments, a safe delivery strategy is used for the causal request messages of the communication stream.

In Fig. 8 the average response time with a given number of server nodes increases approximately linearly as the number of threads increase. This simply fits the common sense that the server turns slower under more workload.

From Fig. 8 we also observe that the simultaneous read-only performance can be improved as the number of server nodes increases. Consider the 40-thread case, average response time decreases to 52.3ms when the cluster has two server nodes, and to 25.9ms when it has three nodes. The pattern of performance improvement is roughly linear. However, when there are four nodes in the cluster, the average response time is 15.7ms. This is because the contentions at clients block the linear increase on performance. The same phenomena can also be observed in 10-thread and 20-thread cases.

However, if there are more clients (i.e., practical scenarios), each invokes a few threads simultaneously accessing the cluster, contentions at the client machines will subside, and the performance acceleration on read-only queries will be higher than that observed from our experiments.

7. Conclusion

In this paper, we propose a scheme, named AR-TCP, which transparently improves the service and data availability of the legacy applications at TCP connection level. By conducting experiments on prototype implementation, we find that AR-TCP results in small penalty on communication. For the application on MySQL cluster, it achieves data consistency among the replicas with small sacrifice on performance of update operations, while performance of simultaneous read-only queries is greatly accelerated.

References

[1] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia, "The Totem multiple-ring ordering and topology maintenance protocol", *ACM Transactions on Computer Systems*, 1998, 16(2): 93-132

[2] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping Server-Side TCP to Mask Connection Failures", In *Proceedings of INFOCOM'01*, Anchorage, Alaska, USA, 2001, pp.329-337

[3] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast", *ACM Transactions on Computer Systems*, 1991, 9(3): 272–314

[4] A. Bestavros, M. Crovella, J. Liu, and D. Martin, "Distributed Packet Rewriting and its application to Scalable Server Architecture", In *Proceedings of the International Conference on Network Protocols (ICNP'98)*, Austin, Texas, USA, 1998, pp.290-297

[5] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", *Journal of the ACM*, 1996, 43(2): 225-267

[6] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study", *ACM Computing Surveys*, 2001, 33(4): 1-43

[7] H. Jin and Z. Shao, "Cluster Architecture with Lightweight Redundant TCP Stacks", In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03)*, Hong Kong, China, 2003, pp.464-467

[8] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal, "An efficient reliable broadcast protocol", *Operating Systems Review*, 1989, 23(4): 5-19

[9] Z. Shao, H. Jin and B. Chen, J. Xu, and J. Yue, "HARTS: High Availability Cluster Architecture with Redundant TCP Stacks", In *Proceedings of the International Performance Computing and Communication Conference (IPCCC'03)*, Phoenix, Arizona, USA, 2003, pp.255-262

[10] Linux Virtual Server, http://www.linuxvirtualserver.org

[11] M. Marwah, S. Mishra and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server", In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'03)*, San Francisco, CA, 2003, pp.373-382

[12] MySQL server, http://www.mysql.com

[13] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman, "A Dynamic View-Oriented Group Communication Service", In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'98)*, June 1998, pp.227-236

[14] A. Schiper and A. Sandoz, "Uniform reliable multicast in a virtually synchronous environment", In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'93)*, Pittsburgh, PA, USA, 1993, pp.561-568

[15] Q. O. Snell, A. Mikler, and J. L. Gustafson, "Netpipe: A Network Protocol Independent Performace Evaluator", In *Proceedings of IASTED International Conference on Intelligent Information Management and Systems*, Washington, DC, USA, 1996, pp.196-204

[16] F. Sultan, K. Srinivasan, D. Iyer and L. Iftode, "Migratory TCP: Connection migration for service continuity in the Internet", In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, 2002, pp.469-470

[17] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems", In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, Taipei, Taiwan, 2000, pp.264-274

[18] R. Zhang, T. F. Abdelzaher and J. A. Stankovic, "Efficient TCP connection failover in web server clusters", In *Proceedings of INFOCOM'04*, Hong Kong, China, 2004, pp.1220-1229

0-7695-2721-3/06 $20.00 © 2006 IEEE