Software library for message-based reliable communication system

CMSC-621 Project Report

Mithun Sheshagiri
Jekkin Shah
Ihong Yeh
Gaurav Jolly

{mits1, jekkin1, iyeh1, jolly1}@cs.umbc.edu

Advisor: Dr. Kostas Kalpakis

Computer Science Department
University of Maryland Baltimore County

May 2002

Software library for message-based reliable communication system

Mithun Sheshagiri, Jekkin Shah, I Hong Yeh and Gaurav Jolly

Computer Science department
University of Maryland Baltimore County
Baltimore, MD 21227.
{mits1, jekkin1, iyeh1, jolly1}@csee.umbc.edu

Abstract:

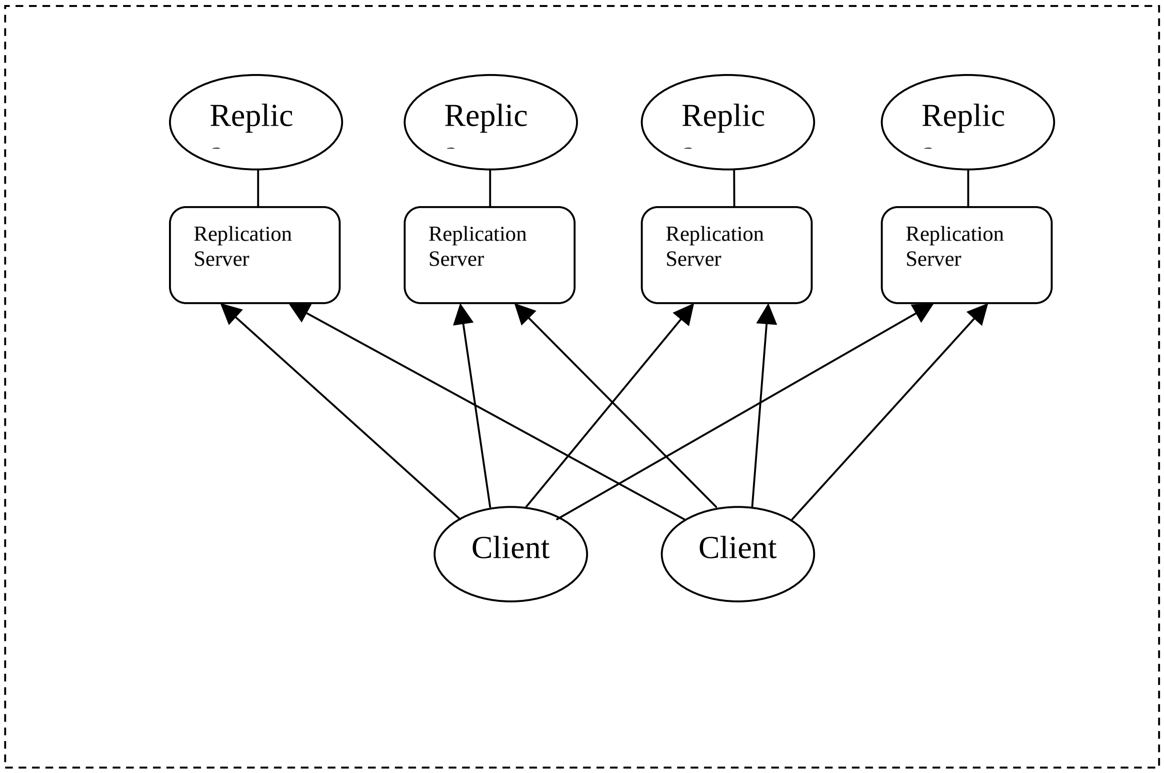
The advent of distributed systems has made it possible to share resources across networks. Thereupon a single resource can be accessed by multiple clients. However if this resource is stored at a single server it might become the source of system bottleneck and moreover the failure of this server will mean unavailability of the resource. Thus in order to increase availability and performance the resources are replicated across multiple servers. But inappropriate implementation of replication software can cause the reverse effect of deterioration of performance. For example discrete copies of the replicated resource might be updated in a diverse order by clients. Consequently each facsimile of the resource will advertise different data. This would be highly inadmissible. Therefore replicated servers should be able to sustain failures of its peers and the updates to shared objects should be consistent across the network. In this project we attempt to develop such an application, where data is replicated across multiple servers. Additionally our implementation (software library) maintains consistency of replicas by causally ordering the update messages for the replicated objects. Moreover the application continues to function even in presence of failures.

1. Introduction:

In a replicated server environment maintaining a unique copy might become a handicap of the network for several reasons. Firstly this introduces single point of failure; besides this to avail a particular service the client has to locate the server as it might not be in the client’s immediate vicinity. Secondly in these scenarios it is difficult to do efficient load balancing. Particular server may be bustling with activity as its resource is heavily utilized while a server for another resource might be idle. Thus in-distributed systems in order to increase the availability of resources, the resources are replicated across multiple servers.

However replication of resources introduces new challenges. For example if the users update these replicas in different order the duplicates of the resource will advertise conflicting information. Therefore all the updates to the replicas must be in a consistent order. Likewise failure of servers and later recovery should be transparent to the user.

In this project we implement a software library based on the atomic broadcast and causal broadcast protocols presented in [1] and [2]. We also apply the concept of Lamport's clock to establish ordering relationship between the messages. Besides this, each server maintains a separate copy (replica) of the resource rather than all clients accessing a unique copy.



Replicated Resource (replica)

The remainder of this report is structured as follows: Section 2 describes the presumed environment. In section 3 we state how we went about planning researching and implementation of our project. Section 4 contains the conclusion and the enhancements we wish to make in the future.

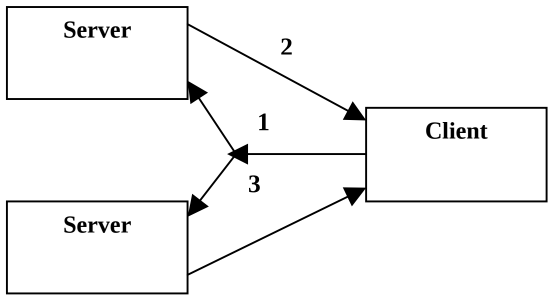
2. Assumptions:

We make following assumptions regarding the environment:

- All the servers must start before the first client can start.
- The communication channels are reliable.
- Clients do not communicate with each other.

- Once a server fails it will not come back unless the entire system is restarted.

3. Implementation of software library:

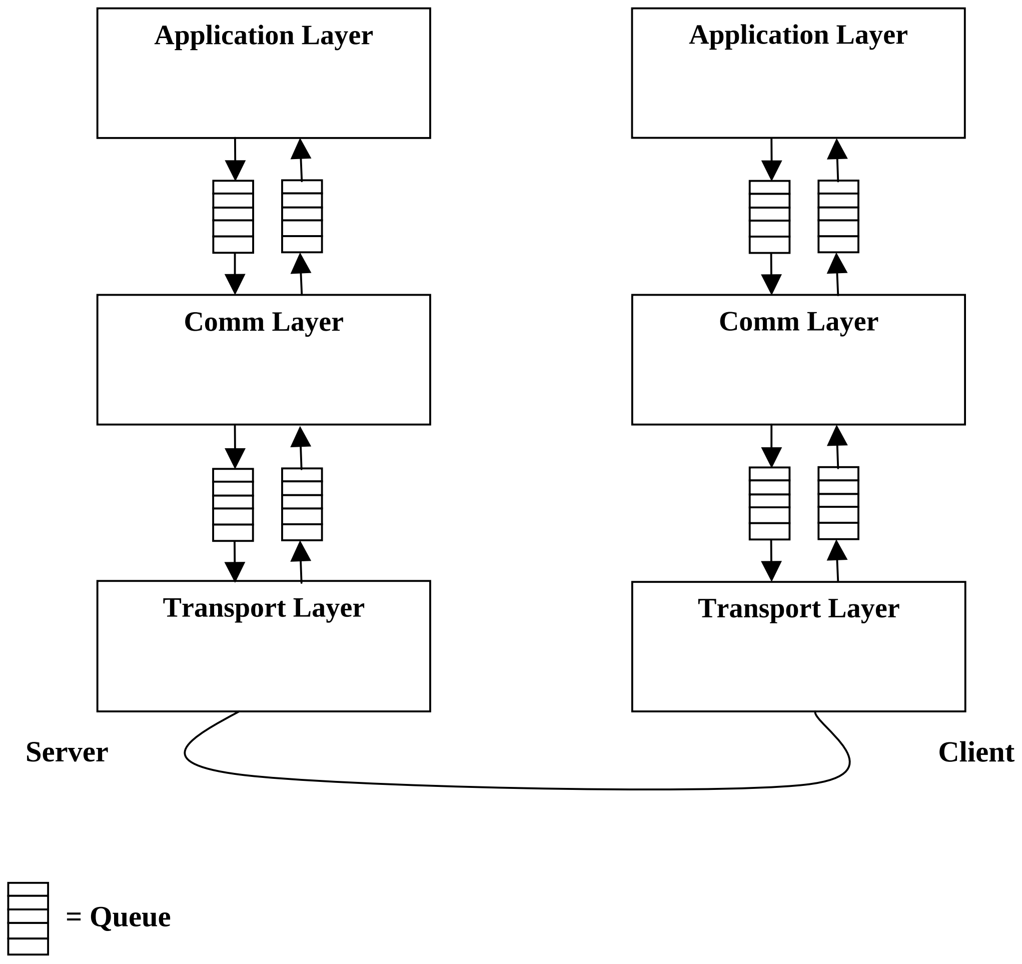


1 = ABCAST
2 = ABCAST ACK
3 = ABCAST2

Communication Primitives

Our software library is based on the ABCAST protocol [2] that ensures total ordering of the messages. In our implementation messages from the client are multicast to the server group using the ABCAST protocol. The servers in turn deliver the messages in an order consistent with the order in which the client sent them. The concept used to implement this ordering is similar to one proposed in the CBCAST protocol [1]**.** Moreover the ABCAST protocol ensures that order of delivery of messages at each server is congruous. The message exchanges in the ABCAST protocol are as follows: first the client multicasts an ABCAST message to the servers. The servers respond with an ABCAST-ACK that contains a timestamp that is one more than the highest timestamp of messages in servers queue. After receiving messages from all the servers in the group the client answers with an ABCAST2 message comprising of the largest of all the timestamps received from the servers. Therefore the message complexity of our protocol is (2+s) where s is the number of servers in the group.

The software library consists of three layers. Functionality of each layer is independent of the other. The topmost layer of the protocol stack is the *application layer.* The function of this layer is to forward the user requests to the underlying *Comm layer.* The Comm layer provides sagacity to our protocol stack. This layer is responsible for deciphering the message from its adjacent layers. Based on the message type it takes appropriate actions. The total ordering at the servers and causal order between the messages is done at this layer. It is at the Comm layer that on receiving a message from a client the server checks if it has received previous messages from the client. If it has received all the preceding messages it marks this message as deliverable and puts it in the queue. Likewise it also marks any subsequent messages already in the queue as deliverable. Conversely if there is any antecedent of this message that is not in the queue this message is marked as undeliverable and put in the queue.



The Protocol Stack

We have made the assumption that the clients do not communicate with each other hence messages of two distinct clients are not causally related. This implies that ordering among messages from different clients is not required. Thus we have used a modified version of the CBCAST protocol in [1] and have used Lamport's clocks instead of using vector timestamps. This reduces the message complexity as each node has to maintain just one counter instead of timestamp of all active nodes. The Comm layer is also responsible for handling server and client failures.

The server failures are detected and handled by the clients. After dispatching the ABCAST message the client waits for some predefined amount of time. If it doesn't receive a reply (ABCASTACK) from all the servers in its group view within this duration of time, it transmits a *probe message* to all the servers. After transmitting the probe message the client again waits for a predefined amount of time. The probe message includes the ID of server(s) whose ACKS have not reached the client. On receiving the probe message if the server is alive it will reply with a message comprising of its timestamp. Conversely if the server has failed it will not reply and the client will update its view by erasing the server ID from its view. This approach works since we are assuming that server and client failures are permanent in nature that is servers will not communicate with the group for the system lifetime once they fail.

Client failures are detected and handled by the servers. If the server has a message from a client at the head of its queue for a time duration more than certain predefined time. It will multicast a probe message to all the servers in its group asking them if they have received ABCAST2 from this client. If any of the servers has received the message from this client it will indicate so in its reply message. And this server on receiving the reply will also update its queue. Conversely if no server has received a message from this client and thus no one has the ABCAST2 then the probe initiating server will assume that the client has failed and will discard the message from its queue.

The lowermost layer is the *transport layer*. The function of this layer is to forward the messages to the destinations (inferred from the message from the Comm layer) in the network. This layer supports two types of communication primitives: multicast sockets and the datagram sockets. We have not used unicast messages when clients communicate with the server since unicast has an inherent disadvantage that usage of bandwidth multiplies with number of members in a group. Instead we have used multicast primitive in java to send messages to members of a process group. Another advantage of multicast protocol is that the user doesn't have to keep track of all the members of the group. The user sends one message addressed to the group and the underlying layer delivers message to all the members of the group.

Another assumption we make is that the transmission medium is loss less. Thus we have used datagram sockets for point-to-point communication. TCP is an expensive protocol as it has to maintain connections between two ends. Moreover it has an additional overhead that all packets are retransmitted in case one packet is lost. This introduces jitter as the packets will suffer delays.

The data replication application consists of five hash tables that can be updated by any of the clients. Our software library ensures that the updates to these hash tables are consistent across the group. Clients can both read and write to these tables. Moreover only one server can perform writes at a time and it has to obtain a write lock before performing a write operation. Hence for the writes to the replicas to be consistent across the group, the client locks all the servers before performing a write operation. Additionally the multicast address of the server group is hard coded. This address is constant for a multicast group. All the servers will receive ABCAST and other messages at this address. Besides this all the clients and servers are given ID's that are unique across the network. Moreover the ports of the clients are mapped from the IDs assigned to them. Hence the ports of each client are also unique.

This layered architecture ensures that changes can be made to each layer independent the other layers. Additionally the implementation of the layered architecture is identical for the clients and servers and hence is flexible. The communications between the adjacent layers is achieved using monitors (wait and notify primitives in java). The wait primitive blocks until notified by the adjoining layer regarding the availability of more message(s).

4. Conclusion and Future work:

In this project we have designed and implemented a data replication application. The data replication is supported by our software library that uses ABCAST protocol and a modified version of CBCAST protocol. Test runs of our implementation show that it ensures causal ordering between messages and it continues to function even in presence of failures of individual nodes. We have used multicast socket to reduce the number of messages across the network. Additionally we make use of Lamport's clocks instead of vector clocks used in CBCAST protocol, hence reducing the message complexity.

Our implementation has been designed for a single group system in which all the servers are the members of the group. In future we will like to extend our work to multiple groups where each server can be a member of one or more groups. In this we have assumed that once a server fails it will not join the group unless system is started from scratch; we will like to remove this restriction in our future implementations.

5. List of References:

1. Lightweight casual and atomic group broadcast. A. Schiper, K. Birman, P. Stephenson. ACM Transactions on Computer Systems, August 1991. Volume 9, Issue 3.
2. Reliable communication in the presence of failures. K. Birman, T. Joseph. ACM Transactions on Computer Systems, January 1987. Volume 5, Issue 1.
3. Horus: A Flexible Group Communications System. R. van Renesse, K. Birman, S. Maffeis. In Prof. of Symposium on Principles of Distributed Computing, pp. 80-89, 1995.
4. Bimodal Multicast. K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky. ACM Transactions on Computer Systems May 1999. Volume 17, Issue 2.
5. A review of experiences with reliable multicast. K. Birman. Software: Practice & Experience. Volume 29, Issue 9, pp. 741-774, 1999.
6. Advanced Concept in Operating Systems. Mukesh Singhal, Niranjan G. Shivaratri.