A Distributed Highly Available Calendar Tool for Workgroups

Huangxin Wang

1. Introduction

In this project, I implemented a highly available calendar tool. The project is implemented using java RMI. The distributed architecture is realized using Chord distributed hash table. Fault tolerance is guaranteed that even if some servers are down, the service is still available. To avoid data loss in the presence of servers shutdown, replications are implemented using primary backup scheme. Overall, the calendar tool I implemented have two main characteristics. On the one hand, the services is distributed which enable the scaling of the services. On the other hand, high availability is guaranteed that even some servers are down, users are still able to their latest version of data set.

2. Assumption

In the implementation, I make one assumption that at most one server can be down at a time. If there are two servers down simultaneously, then my implementation will not work. Before explain the reason, I will first introduce about the information keeps at each server node.

2.1 Data structure

Each node will maintain two kinds of information, one is location information of other node. The other information is the data of clients.
The location information include: finger table, successor, successor of successor(SOS), predecessor, predecessor of predecessor(POP).
The data information include: all the clients whose successor is the server node.

2.2 Assumption

In the fault tolerance part, we recover the failed successor or predecessor using SOS or POP. If two servers are down at the same time, assume successor and SOS are not available at the same time, them the fault will not be able to be recovered.
Another reason is there is only one backup for the data in my implementation, one is stored in the successor, the other is stored at the SOS. Therefore, if successor and SOS down at the same time, then it will not be possible to recover the data.

3.Chord Implementation

3.1 Node ID

Each server and each client will be have an id, they constitute the node. The server id is got by hashing its IP addresss. The client id is the hashed result of its name. We use Message Digest SHA-1 to covert the string into integers. The size of chord is tunable.

3.2 Chord scheme

The nodes are organized in a Ring. Each server node keeps a finger table, keep track of some nodes, and store some clients data. The clients are those who successor is this server node.

3.3 Node join

A node n can join chord by an arbitrary node n'. n' will be responsible for finding the predecessor and successor for n. If n' is null. Then n is the first node in the chrod. When a node join the main procedure it will go through including:

1) Node n gets an arbitrary node n' who is already in the chord
2) n ask n' to find the predecessor P, then n got its successor S, which is P is successors
3) n join between (P, S)
4) n initial its finger table by asking n' to find the successor of each entry of its finger table
5) n update other's finger table

When node leave or join, it may introduce some changes to other nodes. I implemented the update process according to the pseudocode in [1]. I made some minor changes to the update process.
1. in function update_finger_table(s,i)
Pseudocode: if s in [n,finger[i].node), then change finger[i].node to s
My implementation: I made the condition n exclusive to the condition, i.e. change the condition to {s in (n,finger[i].node)}
Reason: here is an example, if s==n, e.g. s=n=9, finger[i].node=23, finger[i].start=10, then it satisfies the condition in the paper, but if we set succ(10)=9, it is wrong.

2.fix_fingers
Maybe let other nodes help fix fingers will be more efficient. Because if a node is wrong, then let iteself update the finger table for itself will not push the finger table onto the right way.

3. Stable Process
The stablize thread will run periodically to check and update the finger table to make it consistent.

4. Primary Backup
Each node will keep a replication of its data in its successor. If the node who has primary copy is down, the replication will be changed to primary copy on its successor, and backup will be performed. If the node who has backup failed, then a new backup will be send to the new successor.

5. Failure Detector
Each server node will check whether its predecessor and successor is alive of not periodically.
In the case of predecessor is down, the node will take the following actions:
1) set predecessor as POP
2) set new POP = P of current predecessor
3) move data from backup to primary
4) backup data to successor
5) notify current predecessor to update successor, SOS, and backup data to it.

In the case of successor is down, the node will take the similar action.

6. Deadlock and Distributed commit protocols
6.1 Non-group events
We will first lock the calendar objects, schedule events, and then unlock the calendar objects. If it failed, then the entire calendar is restored from temporary backup that was created before the commit.

6.2 Groups events
To prevent deadlock, the calendar object related to the group events are accessed in increasing order by their id. Distributed commit protocol is implemented to deal with the case of partial failure.

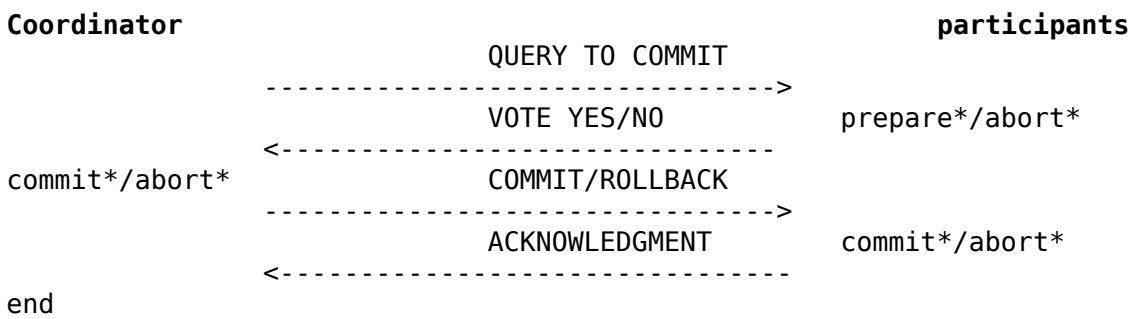
The two phase commit protocol are constituted by two phases, they are voting phase and commit phase.

Commit-request phase
1) The coordinator sends a query to commit message to all participants and waits until its has received a reply from all participants.
2) The participants replies YES/NO. If if reply YES, it will change the state to COMMIT, backup the data to the point before commit, lock the calendar object. If if reply NO, it will change the state to ABORT.

Commit phase
3)The coordinator collects the votes
If all the votes are YES, the coordinator decides to commit and sends doCommit request to each of the participants.
Otherwise, the coordinator decides to abort and sends doAbort requests to all participants that votes Yes.

4) Participants that vote Yes are waiting for the doCommit or doAbort request from the coordinator. When a participants receives one of these messages it acts accordingly.
In the case of doCommit, it will makes a haveCommited call as confirmation to the coordinator.

The message flow is:



7. Failure Handling
7.1 Failure Types Handled
Server crashes
When server crashes, the failure detector would adjust the finger table for the nodes affected. The client who has connected to the crashed server will be reconnected to the new server. The process of reconnection is transparent to the client.

7.2 Failure Types unHandled
1) Primary and back both down
In case of two nodes failed at the same time, my implementation may not work. Because if these two nodes happen to own primary and backup of some data, then these data will not be able to be recovered.
2) Successor and SOS both down

Also each nodes only keep track of its closest 4 nodes, this may not be sufficient. For example, if two noes failed at the same time, and these two failed nodes are the successor and SOS of current node, then current node may not be able to get its new successor in a short time. To avoid this long delay, we can keep more successors instead of keeping only two.

3) Untimely backup

If the server crashes before the backup is updated, then there will be no way to recover the lost update.

4) Coordinator down in two phase commit

For time limitation, I didn't handle the case when coordinator is down. To implement it, participants need to broadcast to other participants to get the decision.

Reference

[1]Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (SIGCOMM '01). ACM, New York, NY, USA, 149-160. DOI=10.1145/383059.383071 http://doi.acm.org/10.1145/383059.383071

[2] Two-phase commit protocol
http://en.wikipedia.org/wiki/Two-phase_commit_protocol