1 Overview

This Calendar Implementation includes levels. Calendar Application components and Chord beneath them. *Calendar Manager*s reside in arbitary node and accept connection from *Client*s. *Calendar Manager*s communicate with each other through *Chord*, which can deal with routing or partial failure recovery. *Calendar Manager*s are responsible to distribute *Calendar*s in balance and help *Client*s find their own *Calendar*s.

2 Chord

Chord uses its *Finger Table* as partial route table and keeps its predecessor and successor of successor for correctness. So more than successive two node failures are out of responsibility in this implementation. For simplicity, one-to-one map scheme is applied on hashing medusa 19 nodes.

3 Replication

Replication happens at Application level, which is handled by *Calendar Manager*s. Each *Calendar* Object is hashed to 0-19 and distributed to corresponding node, which is *primary copy*. *secondary copy* is hold by the successor node. For correctness and availability, 1) A new node tries to pull *Calendar*s which should belong to it from its successor once it joins the *Chord*. And 2) Every Calendar Manager checks if its Calendars are *primary copy*, *secondary copy* or neither. For primary copy, it should update itself to its successor. For secondary copy, do nothing. Otherwise *Calendar Manager*s deletes this *Calendar* object.

4 Failure Detection

Client explicitly connects to one *Calendar Manager* via its address, then retrieves its *Calendar* stub. The *Client* detects if both *Calendar Manager* and *Calendar* are not alive periodically. Only *Calendar Manager* failure asks *Client* to do reconnection explicityly and new *Calendar* stub , which is originally secondary copy, would be automatically sent back if *Calendar Manager* is alive.

*Calendar Manager*s do not handle peers failures, which is corrected implicitly by *Chord*. Details are in section 2.

*Calendar Manager*s explicitly detect if Clients which have connected to it are alive. Then server decides if send notification of expiring Event to them.

5 Deadlock Avoidance

While do Group Event schedule/deletion/update, global sequence is under consideration for deadlock avoidance. Before these operations, Coordinator, the *Calendar* who start transaction, aquires locks in ascending order of participants' ID and releases locks in descending order.

6 Distributed Commit Protocol

Group Event schedule/deletion/update are also considered as automatic.

For schedule, after locking all relevant *Calendar*s, potential conflict between this group event and existing events is checked through all *Calendar*s. If everything is OK, the second phase, is then do schedule. Failure in the first phase affects nothing, but coordinator has to rollback all insertion done if failure happens in second phase.

Update is composed of two sub-transaction: deletion of old event and insertion of new event. Once deletion is done, re-insertion of old event is required if any failure happens.

This scheme can not handle the case of coordinator failure. Once the coordinator is down, whole system has to bear inconsistency and deadlock.