Project report
ConfChat

Course: SERS (1st Semester - 2012)

Professor
Ricardo Lopes Pereira

Date of Submission: 14 December 2012

Group 5

Name	Student Number
Pushparaj Motamari	77160
Anh Phuong Tran	77182
Dipesh Dugar Mitthalal	77184

1. Problem statement

The ConfChat application allows registered users to chat to friends in one to one conversation or one to many (chat room) fashions. The application is built upon a P2P overlay network and can automatically function without any centralized server.

A monitoring function allows users get statistical data about the system such as number of nodes running, average message rate, etc.

2. System description

2.1 Functions

ConfChat provides the following functions:
#signup*username password full name*: register for new account
#login *username password*: log into the system
#search *keyword*: search users using word from their full name
#add *username*: add friends
#accept *username* /acceptall : accept friend request(s)
#call *username*: start 1-1 conversation with already accepted friends
//user will be automatically changed to offline mode if friend is not online
#list: list all available conversation
#off : see offlines messages
#request: see all pending requests
#friend: list all friend and their status (Online / Offline)
#create *roomname*: create a new room for group conversation (chat room)
#invite *owner roomname username*: invite *username* to *roomname* created by *owner*
#join *owner roomname*: join the invited room
#talk *username /owner roomname*: switch focused conversation
#bye : leave the current focusing conversation or chat room
#whoami: basic information
#monitor: monitoring function
#help: display help menu

2.2 Overlay network

In order to facilitate the above functions, we propose to use Chord as the Overlay Network. The chosen library is OpenChord[1]. The reason for choosing OpenChord is it is simple to use, as it doesn't require much of the configurations. OpenChord provides developers basic Chord API such as create (chord), join (chord), insert (key, data), retrieve (key,data), remove (key, data). With these primitive APIs, we are free to design our application with lots of flexibility. Besides, the data consistency characteristic is totally favorable in our application since we need to store data on the network.

OpenChord introduces a few minor drawbacks compared to other P2P Overlay Network implementations. Both FreePastry [2], an implementation of Pastry, or openKad [3], an implementation of Kademlia, provide developers a send (nodeID, message) method to send the message to a specific node in the network. Nodes can use this send() method to start

[1]http://sourceforge.net/projects/open-chord/
[2]http://www.freepastry.org
[3]http://code.google.com/p/openkad/

communicating with other nodes directly, but in openChord it is left for the upper application layer to handle.

2.3 Modules description

2.3.1 Register new users and log in

Any non-logged in user can register for a new account. The information will be stored under user's Password Key (only password) and Full Name Key. When a user logs in, confChat will retrieve his password from his Password Key. A check on the input value and retrieved value will either allow the users to log into the system or deny him.

2.3.2 Searching and friendship

When a new account is registered, confChat will store the username under the key deriving from hashing each word in the user's full name. A user can search for another user by retrieving all the objects stored under that key.

When a user wants to establish friendship with another user, he inserts his name in the friend's PendingFriendList key. Friend will get notified either by offline message, or by Notification (see 2.3.4). Once the friend accepts the request, he will move the other user's name from his own PendingFriendList to his FriendList (storing as object under the predefined key). He also adds himself into the friendship requester's FriendList. From this point, the two can start talking with each other.

2.3.3 One-One Chat

To falcilitate single conversation, we decide to use (TCP) Socket/Server Socket classes provided in java.net package. These classes allow us to open a TCP connection between two end-points to exchange chat messages. The choice of TCP connection is dominant compared to UDP, as our chat application requires fast, reliable, and orderly correct connection.

When a user logs in, he will store his IP address & listening port under his Address Key (a key specific to the user's IP address). Any users who want to contact this user retrieve the value from his Address Key and use those values to open TCP connection.

When the user logs out, he will remove the data from his Address Key. Any retrieve on his Address Key will return NULL, and therefore others will know that the user is not currently offline.

Should a node fails; he would not be able to remove his own IP address. If a user uses this value to contact him, he would not be able to open a TCP connection. Then that user will have to remove the out-dated value from the friend's Address Key.

Also when a user logs in, he will do the cleanup of old data before inserting new values.

When a user wants to talk to his friend who is currently offline, ConfChat will automatically change into Offline mode. All the following typed messages will be stored under the friend's Offline Key. This allows the friend to receive offline messages when he logs back in.

2.3.4 Sending Notification

During a confChat session, user might get some notifications from the system, such as a friend request coming, or some one invites him to join a Conference. This also can be realized by utilizing TCP connection as described from above. A set of 3 messages will be sent to users in the following order: friend name; notification message; bye message. (This, of course, can be reduced by concatenating messages. However we decided to do this for the sake of clearer code).

Also, this method is also used in checking whether the user is online or not. If a user tries to log in using the same account with an already-in user, he will fail, as confChat can "ping" the previous logged in one.

2.3.5 Conference

A user can create a new conference (chat room), as long as the room name is unique among hisown created rooms (to be clear, a combination of username & room name will be unique in the entire network)

Any currently participating user can invite other users to join by sending their friends a notification using pre-defined format. The invitation is kept in the user's local data, which will be lost if the user logs out or crashes. User will have to check whether he has an "invitation" for a room before being able to join it.

A list of all users currently participating in the Conference is kept under the key made from the combination of the owner and the room name. Any new comer, after receiving the invitation, can query data storing under this key. From that, he will obtain a list of all participating users. He will then insert his own address to the same key, and use the Message Routing Protocol described in 3.1 to announce his arrival to the current users. Each user will update his own's participant list when sees a newcomer announcement message. From that point, everyone can exchange message within the Conference.

When a user leaves, he will have to announce his leaving to everyone. Each remaining user updates their own participant list, while the leaving user removes his self from the room's key in Chord. Later coming user will not have this user's address, and therefore cannot send the message to this user.

Should a node fails, the Message Routing Method will do his job by removing the fail node's address, and announce the failure of node to all other participants.

2.3.6. Monitoring

By employing the Divide and Conquer method describing in 4.1, user can have a correct value of the system's statistical status in a good network condition. A good network condition is the condition of the network such that, for a given time, nodes have no broken links in their finger table.

The Divide and Conquer method will probe the network, forming a search tree that covers all nodes in the network. The number of reply messages will then equal to the number of nodes in the system.

Together with the value 1 for number of node, each node can answer the query with its number of storing entries of a specific type of object. That is to say, each node will announce the number of Registered User Objects and Offline Message Objects in the replies. The total number of counted objects equals the number of distinct objects times the configurational number of Replicas.

The same approach can also be applied in counting the number of running conference, if each Conference is declared under a Key. However, we can also count the number of running conference by calculate the sum of the reverse value of the number of each Conference's participants. For example, user A that joins 2 conferences at the same time, each have 3 and 5 participants, respectively, will announce his value as $1/3 + 1/5 = 8/15$. The total value of all nodes in the system will equal the number of running Conferences.

For the average message rate, we comprehend it as the sum of each user's average message rate (since they are calculated within the same period of time of 1s). Whenever a user logs in, he will have a local variable recording his log-in time. He also keeps track of all the messages he has sent. When asked, he will simply reply with the number of sent message over the elapsing time since log-in.

3. Message Routing Protocol

3.1 Motivation

A conference room should be able to support a large number of participants. We can realize this by utilizing the single conversation schema as stated above. There are 3approaches to realize this:

- Everyone opens a TCP connection to each of the participants in the room. This has the benefits of fast, reliable connections, but an active user would be quickly running out of resources (even though this is just a rare extreme case). The number of required connections for a *n* people chat room is $n(n-1)/2$

- Every one connects to 1 participant (can be room owner). This reduces the number of connections to *n* connections for an *n*-participants conference room. However, if the room owner crashes, the whole room will vanish and other participants will not be able to communicate with each other anymore.

- A mixture of the above 2 methods, when we have a small number of participants acting as "super users" and deliver the messages to other users connecting to them. Again, this stumples upon trouble when a super user fails, and the management of connection is difficult

3.2 Message Routing using openChord

To avoid all the problems mentioned above from using direct TCP connection in the application layer, we decide to make use of already established connections between nodes in Chord (openChord implementation).

OpenChord already provides us an Insert(key, object) method with which we can insert object into a node that is responsible for the key. This, together with roughly 10 more diffirent methods, is all handled by the RequestHandler class in the package de.uniba.wiai.lspi.chord.com.socket. We inserted into the source code a new case for our own sending message, upon receiving which a callback function will be triggered. By this, our application will know about incoming message as soon as it arrives. This newly created function is then used in falcilating the Conference as well as the Monitor.

Originally, whenever a request of inserting an entry to a key is made, openChord will look up for the node that is responsible to that key, and open a direct connection to the node to perform the inserting task. This is the same with the first approach, when everyone open connections to every one else in the Conference. We made a tweak in the code, limiting the look up part in the node's own finger table only. Sending message now will make use of already established connections between the node and his fellow in the finger table.

Another worth noticing point is that, openChord uses a blocking call for the Insert() method. The function will only return when it receives response from the other party. Since we are now sharing our communication with Chord-related task (stablization, finger table fixing...), we decide to make our function non-blocking (without waiting for the response). This makes sure that openChord will always have slot to recover itself from failures.

When a node fails, his ID will still be in other's finger tables. Some nodes might try to send messages to this ID, which is no longer reachable in the whole network. In the early stage of development, those messages will loop forever in the network, as no node has the match between localID and the message's destination. To solve this problem, we include a TTL value of 60 in each message. Once the TTL reaches 0, it will be discarded. We say that 60 is a big enough number, as it is the average hops number for a network consits of 2^{60} nodes, which is an unreal number.

This mechanism is also used in maintaining the Conference. When a node catches a conference message that running out of lifetime, he will remove the receiver's ID from the list in Chord (using information packed in the message). Then he will inform all the other participants about the leaving of the failed node. With this, the number of participants in a conference is kept up-to-date.

Currently, we always aim for the furthest node possible (the node that is closest to the desired destination in our own finger table). This choice makes the average number of hops remains at O(log n) level like the desired value in the Chord protocol. However, it has some drawbacks. We will discuss about this in the Critics section.

4. Monitoring Algorithm

We employ the Divide and Conquer method to probe for the network. The idea is to form a search tree, with lower level sub-trees covering smaller parts of the network without overlapping each other (Fig.1).

A node, upon calling the #monitor function; will start by sending himself a monitoring request message. The message clearly states the origin of request, the source of the message, a boundary over which all the messages following this message will not cross. For the first message, all of these values are the same and equal to the requesting node's own ID.

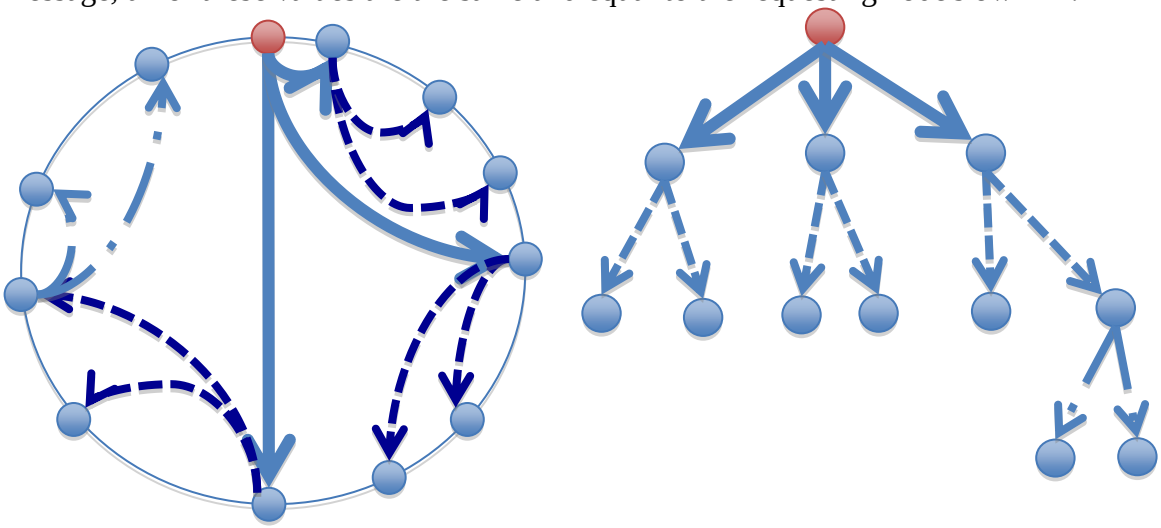


Figure 1: Divide and Conquer algorithm and the search tree illustration

Each node, upon receiving a request message, will look into his finger table and start sending the request away to all the nodes that lies between him and the boundary (so-called possible nodes). The order of message sending is reversed: node will send messages the last possible node first, than continue until it reaches the first node in the finger table, which is his own successor. The first sent out message has the boundary from the coming request, while the next request has its previous sent out request as boundary. By limiting the range of each request, requests will not overlap each other's, and therefore we will get the correct value when all reply messages return.

Also, to tackle the situation of node failure (no reply comes back), each node, after send out request, will start a timer. At the same time, node will have to notify the previous requester about his sending out request messages by an ACK message. Node, upon receiving an ACK message, will stop their timer. Else, node will keep counting down time, and when time runs out,

it replies with the data it has received so far. In our implementation, we set a time out value of 5 seconds (see Algorithm. 1)

5. Critics

Our 1-to-1 chat module is quite independent from the overlay network. That is to say, as long as the underneath overlay provides basic functions such as storing and retrieving object from the network, we can port our code to another overlay network without much effort.

The TCP connection performs very well in many testing scenarios. Even though we still have some minor bugs in displaying text message, our confChat's 1-to-1 module is stable and able to handle node failure pretty well.

We use a random function to open port to listening to incoming network. For now if the port is occupied, the application cannot start. In future work, we can keep on trying selecting port unitl we get a usable one.

The whole application is built upon Chord, therefore our application suffers from some of Chord's limitation. Should a series of consecutive ID nodes fail, user's data might be lost forever.

Our Conference and Monitor module heavily depend on Chord. Once the Chord fails to recover from peer churn, our application suffers. The reason is we use finger table to route the message. If a node fails, and the finger table is not yet fixed, we might not be able to route any more message.

As mentioned above, we use a simple greedy algorithm to select which node to forward the message. While it gives us a better performance in term of latency, a path through that node to a specific node is fixed, and therefore it's liable to failure. We have thought of 2 different approaches to overcome this drawback. The first one is, instead of choosing the furthest one, we randomly choose one from the possible nodes' list. This will reduce the chance of being affected from a node failure, but it increases the number of hops a message needs to travel. This translates in higher latency.

The second approach is to send more than 1 message to different nodes. This increases the amount of message over the network, but it also guarantees a higher successfull rate for each passing message.

One thing needed to mention about the use of Routing Message Method (RMM) is that, connection between 2 nodes is one-way. When a node receives a message from other nodes, it will route the message on a different way. A round of message (asks and reply) will basically travels around the Chord Ring before coming back to the first node. Also, RMM does not guarantee that the message is actually received by the desired node. Our future work will consider digging further into the openChord's source code, so that we can actually expose the underlying TCP connections to the application layer. This will make sure the desired node will receive a message, as well as reduce the message latency.

6. Evaluation

We have installed java in 96 nodes in planetlab using a bash script which ssh each node and SCP java RPM and installs. Then java executable is copied on all nodes and we tested on 6-10 nodes randomly. Sometimes nodes are very slow and the terminal just hangs. But we able to execute above test cases on nodes by copy pasting input in a one go on each node.

7. Conclusion

We have managed to implement a P2P chat application based on Chord using openChord implementation. The application is tested in Planet Lab. Even the number of testing node is small; our application can cope with a bigger number of nodes compared to the test.

Appendix 1. Test case and result from Planet Lab

S.NO	Test case	Result Expected	Actual Result	Remarks
1	Signup in one node (node1)and login in another node(node 2)	User must be allowed to login in node2	User logged on in node2	**PASS**

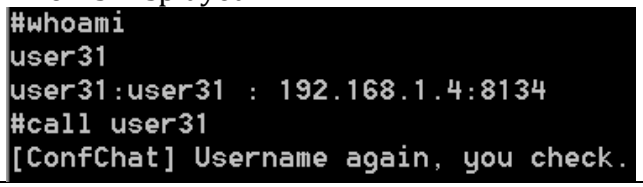
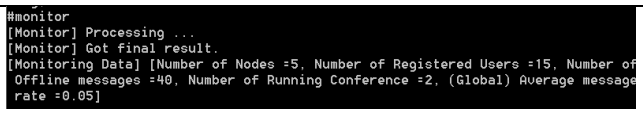
2	Login with incorrect userId or password	User should not be allowed to login	`#login user33 user3`
			`Wrong user name or password`
			User is Shown error message
				PASS

3	Searching for user whose full name is Dipesh Mitthalal	If user types"dip" system will not show any result. If user types Dipesh/dipesh/mitthalal/Mitthalal user id:Full name is displayed	`No match for dip`
			`#search mitthalal`
			`Search result for mitthalal`
			`1) dipesh:dipesh mitthalal`
			`#search dipesh`
			`Search result for dipesh`
			`1) dipesh:dipesh mitthalal`
			User is shown appropriate message
				PASS

4	Add friend with incorrect UserID	Error message to be displayed	`#add carlos`
			`[ConfChat] Is it you, or someone not even existed?`
			User is shown appropriate message
				PASS

5	Displaying Friend list	User should be displayed his list of friends with their status(ONLINE/OFFLINE)	`#friend`
			`[ConfChat] Friend list of user31`
			`1) user33 -- ONLINE`
			`2) user32 -- ONLINE`
			`3) user35 -- ONLINE`
			`4) user34 -- OFFLINE`
				PASS

6	(Following test case 5)Displaying Friend list when the node for which a friend is online(user 33) is crashed.(cr	Appropriate user's status should be updated to offline and updated list to be displayed	`#friend`
			`[ConfChat] Friend list of user31`
			`1) user33 -- OFFLINE`
			`2) user32 -- ONLINE`
			`3) user35 -- ONLINE`
			`4) user34 -- OFFLINE`
				PASS

	ashing by ctrl+c)			
7	Trying to chat with a user who is offline	When a friend is offline, you will be shown a message and then your messages will go as offline		PASS(end with #bye is must)
8	Trying to chat with a friend who just came online	User will be allowed to send online messages to his friend who just came online and messages are exchanged	User will be shown messages but a talk command is mandatory to have a realtime bidirectional communication. (manual entering of talk command mandatory from second time login on same node)	PASS(manual entering of talk command mandatory)
9	Trying to chat with your own self	Error to be displayed	Error is Displayed 	PASS
10	Node fail in 1-1 chat	Alert message is shown to another user	User is shown alert	PASS
11	Creating a conference room and inviting online users	Online users should be shown alert. Invitation cannot be sent to offline users	Online users are shown alert. Invitation cannot be sent to offline users and error is displayed	PASS
12	User participating in more than one conference	User is allowed to participate in more than one conference and he can switch using #talk command	By using #talk command and giving ownername and roomname one can switch between conferences as well as one to one chat	PASS
13	management	User is displayed the management statics on entering keyword:#monitor		PASS
14	One random user crashes while	Conference should be able to continue	At times it works when finger table in chord stablizes . some times it doesn't work and user is shown error of cannot send message	**FAIL**

	conference			

Table 1: Test case and result from Planet Lab

Appendix 2. Divide and Conquer algorithm in monitoring the Network

```
Upon receiving <OriginID, SourceID, BoundaryID, MsgTypeRequest>message:
        send_request = 0;
for ( to node in finger table between (localID, BoundaryID) ){
      limit = BoundaryID;
      send <OriginID, SourceID, limit, MsgTypeRequest>;
      limit = node.ID;
      send_request ++;
}
if (send_request = 0) {
      send to message.SourceID:<OriginID, localID, data, MsgTypeReply>
}
else {
      send to message.SourceID<OriginID,localID, MsgTypeAck>
}
start timer (5*1000) //wait for 5s
if (timer.end) {
      if (reply_count != send_request){
      error = true;
        send to message.SourceID<OriginID,localID,data, MsgTypeFault>
      }
      else {
      if (error = true) {
        send to message.SourceID<OriginID,localID,data, MsgTypeFault>
      }
      else {
        send to message.SourceID<OriginID,localID,localData, MsgTypeReply>
            }
      }
} else if (timer.reset) {//do nothing}
Upon receiving <OriginID, SourceID, MsgTypeAck>message:
if (localID != OriginID){
      timer.reset
}
Upon receiving <OriginID, SourceID, data, MsgTypeReply>message:
      localData = localData + data;
      reply_count ++ ;
Upon receiving <OriginID, SourceID, data, MsgTypeFault>message:
      localData = localData + data;
      reply_count++
      error=true;
```

Algorithm 1. Divide and Conquer method in monitoring network