Design Document :
Alerting system for evacuation in a Secure CITI

K Narayanan | Iteration : 1.0 November 21, 2006

Background:

The envisaged secure citi(S-CITI) users are all equipped with Zaurus SL 5500 pda's that are wireless enabled and run Linux 2.4.18. The objective of this project is to implement an alerting mechanism that runs on top of (α,t) protocol framework that supports different inter and intra cluster routing .[1] have developed a user space Distributed Dynamic Algorithm (DDCA) which sets up the routing framework i.e routing tables are updated dynamically by discovering and maintaining a neighbor list through periodic hello messages.

The alerting application is to use the abovementioned framework to communicate the evacuation plans to different users based on the number of users in a room(s) and the exits available to them. The location problem of the pda users in the building is solved by giving the users a list of rooms to pick from and an option to switch rooms as they move in the building. The server thus keeps track of the different users in the building and not only delivers individual evacuation plan, but also provides an additional chat feature ☺ for the users within a room.

System Requirement:

The system requirements are broken down to requirements at the client and the server.

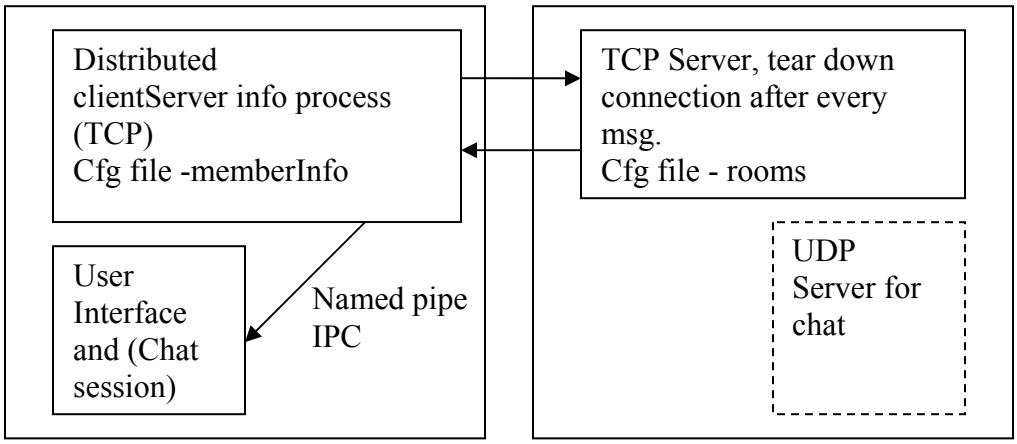
Client side requirement:

1. Clients must have a user interface to execute commands.
    a. to view available rooms
    b. to register to one of the rooms to receive alert messages (could also be chat messages from users in the room – future ?)
    c. to switch between rooms as they move in the building
    d. to quit the application when they exit the Secure CITI

Server side requirement:

1. Server must have a configuration file of the available rooms that a client can register to.
    a. Must support multiple clients simultaneously, and maintain location information of individual clients
    b. Facilitate register-response mechanism for the location registration process
    c. Must be amenable to quickly implement the chat feature for users within a room once the high priority alert mechanism is in place.

System Architecture:



Rationale behind the architecture:

The system requirement of the S-CITI system reflects a striking similarity to a widely prevalent application on the internet – the chat system. The architecture for such a system with suitable modifications would be an ideal design for our S-CITI alerting system [2].

The clients contact the server with their information <host><port> etc.(see section on tcp messages) and the room they would like to register, the server looks up the room info; if capacity is not reached it sends a response OK for the registration if not it rejects it. The response carries a randomly generated ID for each client that enhances security, which the client uses for the ensuing communication with the server.

The User Interface (UI) is a process forked by the client's Server info process (cSip), since the Linux pda's don't have xterm on them we cannot fork and exec xterm –e userIface. For now we have to run two processes at the client one for the - UI where the

alert and chat messages would be received. Additionally the UI is where the user issues commands to register/switch rooms. The second process in the client - cSip is the distributed server information store and TCP communication interface.

Decomposing the system functionality into UI and cSip ensures modularization and ease of code maintenance for future feature additions. The communication between cSip and the server uses TCP sockets, with connection establishment and tear down after the message is communicated (this is similar to http), this ensures that the connection states don't clog the server process.

cSip distributes the server role – plays an important part in the p2p chat system, it caches the peer clients <host> <port> information that is obtained from the server which it uses directly, instead of querying the server to send UDP chat messages.

Control messages between UI via cSip and server

The end of any message is indicated using the new line character ('\n'), following are some of the messages for the envisaged system.

- JOIN REQUEST <*membername*> <*hostname*> <*udpPort*> <*tcpPort*> \n
  <*membername*> is the nickname the member wishes to use.
  <*hostname*> is the name of the machine on which the client is running.
  <*udpPort*> is the UDP port of the client where chat messages may be sent.
  <*tcpPort*> is the TCP port of the information server where information requests must be sent.

The client registers to the server using this message and must send this message first, before sending any other messages.

- SWITCH ROOM REQUEST <*memberID*> <*roomName*> \n
  <*memberID*> is the ID assigned to the member. (Refer to JOIN RESPONSE below)
  <*roomName*> is the name of the chat room the member wishes to switch to.

- JOIN RESPONSE OK <*memberID*> \n (or)
- JOIN RESPONSE REFUSE \n
  <*memberID*> is a randomly assigned ID that is used to identify this client in subsequent messages.

The server can deny the client permission to enter the system/room either because the server limits the number of users in a room and prevents duplicate membernames within a room.

- SWITCH ROOM RESPONSE OK \n (or)
- SWITCH RESPONSE REFUSE \n

The server can deny the client permission to switch to a different room, for similar reasons given above

- MEMBER LIST REQUEST \n

This message asks for a list of all members logged on to the chat session.

- MEMBER LIST RESPONSE <*space delimited list of member nicknames*> \n

- ROOM LIST REQUEST \n

This messages asks for a list of names of all chat rooms.

- ROOM LIST RESPONSE <*space delimited list of rooms*> \n

- QUIT MSG <*memberID*> \n
  <*memberID*> is the ID assigned to the member.

This message is sent when the client terminates the chat session.

These messages are not exhaustive, and do not contain all the chat related messages some of which are listed below:

- MEMBER INFO REQUEST <*remoteMemberName*> \n
  <*remoteMemberName*> is the nickname of the member whose information the client wants.
- MEMBER INFO RESPONSE INFO FOLLOWS <*hostName*> <*TCPPort*> \n (or)
- MEMBER INFO RESPONSE UNKNOWN MEMBER \n
  <*hostName*> is the name of the machine on which the remote client is executing.
  <*TCPPort*> is the TCP port of the information server of the remote client.

If no member exists with the requested the server replies with a UNKNOWN MEMBER response. Otherwise, the server returns the <host> and <port> of the remote client.

Implementation Concerns:

1. Server multiplexes many clients using the FD_SET feature of select system call.
2. Named pipes or FIFO is used for communication between UI and cSip processes, have to look into the details of using mknod – block/char special files for implementing FIFO before proceeding.

References:

[1] On the implementation and performance of the (α,t) protocol on linux A Gopalan, S Dwivedi and T Znati
[2] Man pages and WWW for the innumerable examples on linux socket programming, e.g
http://beej.us/guide/bgnet/
http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html