

**1. Objectives:**

- Learn to work with Nachos networking for communication among processes.
- Define and implement a “chat” protocol for interaction between a chat server and a chat client.

**2. Problem Statement:**

Chat rooms have become a popular way to support a forum for n-way conversation or discussion among a set of people with interest in a common topic. Chat applications range from simple, text-based ones to entire virtual worlds with exotic graphics. In this project you are required to implement a simple text-based chat client/server application.

**3. Problem Description:**

Email, newsgroup and messaging applications provide means for communication among people but these are one-way mechanisms and they do not provide an easy way to carry on a real-time conversation or discussion with people involved. Chat room extends the one-way messaging concept to accommodate multi-way communication among a set of people.

**4. Nachos networking infrastructure**

Nachos networking packages implements a very simple Unix domain (not internet domain), datagram socket. The files of importance to this project and their purpose are described below.

**threads directory:**

**lockcond.h, lockcond.cc:** for defining and implementing lock and condition synchronization primitives; needed for network applications to work.

**machine directory:**

**network.h, network.cc:** Data structures to emulate a physical network connection. The network provides the abstraction of ordered, unreliable, fixed-size packet delivery to other machines on the network. You may note that the interface to the network is similar to the console device -- both are full duplex channels.

**sysdep.h, sysdep.cc :** Interprocess communication operations, for simulating the network; Unix sockets creation, binding, closing etc. are called here to provide nachos socket functionality.

**network directory:**

**post.h, post.cc:** postoffice and mailbox, mail message definition and implementation.

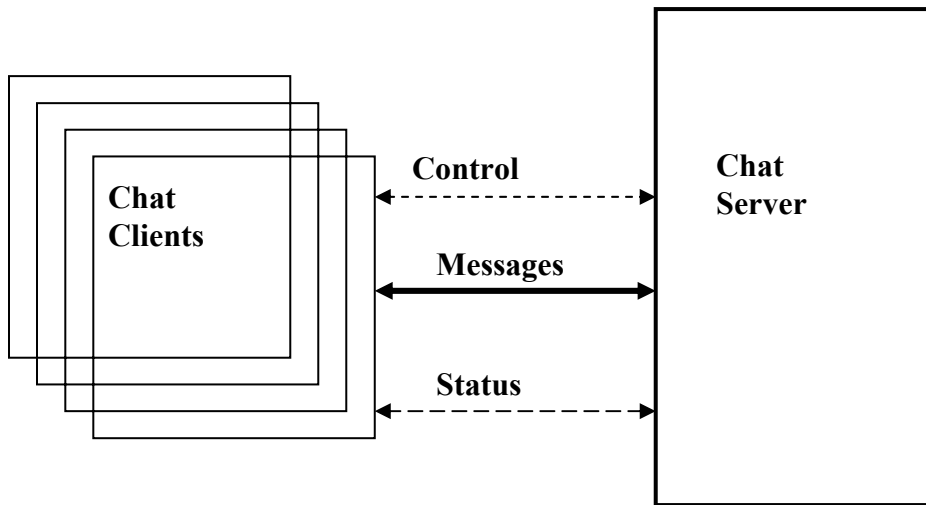
**nettest.cc:** application to test communication between host id 0 and host id 1 (these are hardcoded!)

---

<sup>1</sup> Created by B.Ramamurthy and C.Egert for Unix Socket programming. Adapted by B.Ramamurthy as Nachos project.

## 5. Chat Architecture:

A chat application consists of a Chat Client (CC) one per person, a Chat Server (CS) and a two-way communication pipeline between the client and the server to send and receive conversational, control and status messages.



**Figure 1: Chat Application Overview**

### 5.1 Chat Client:

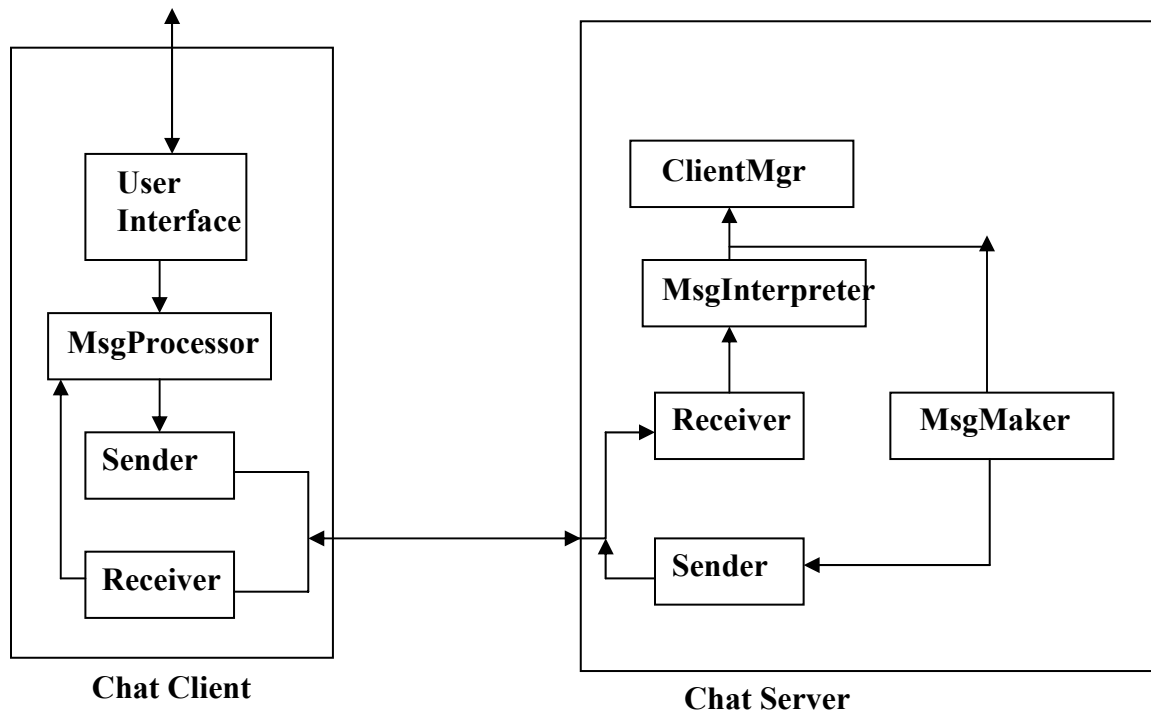
Typical features of a CC include: (i) select chat server (server id), (ii) select a nickname for interaction and (iii) ability to set and change user preferences such as number of messages displayed, change nickname, etc. You may design your own chat user interface.

#### Implementation notes:

- (i) The server id is specified at command line (`nachos -rs 1234 -m selfid -o serverid`) while executing the nachos process representing the client.
- (ii) Once the client process is running it could interactively ask the user for the nickname of the chat user and update this information locally and if needed by your design on the server.

### 5.2 Chat Server:

A chat server supports the set of clients for a room, by maintaining client handle (clientid in `nachos -rs -m clientId -o serverid`), and client name. Server also has a message interpreter that parses the message received from a client and delegates the command to the appropriate module. Sample architecture for the chat application is shown in Figure 2. You may change the architecture to suit your design.



**Figure 2: Chat Application Architecture**

The Chat client receives the user messages and user configuration set up commands and passes them to the server. Sometimes it is also possible to process some of the commands (Ex: Number of messages displayed) locally. MsgProcessor in Figure 2 Chat Client is responsible for interpreting messages from the user. Sender and receiver are for communicating with the server. Messages are constructed as described in the protocol below.

A chat server receives the commands and messages from the chat clients and processes them. MsgInterpreter is for unpacking, parsing and delegating the commands to the appropriate units on the server side. MsgMaker constructs the messages to be sent back according to the protocol described below.

We will use nachos sockets for communication between the server and the client.

### **Chat Protocol:**

Protocols such as TCP and HTTP provide rules for communication. They specify details of message formats; they describe how an application responds when a message arrives, and how to handle abnormal and error conditions. We describe the protocol we will use for the chat service in Section 8.

## **6. Implementation Details**

**Phase 1: Study all the code associated with the nachos networking.**

**Phase 2: Prerequisites: Lock and Condition: (15%)** Implement the lock and condition the skeleton for which are in threads directory. Test it. Run the nettest application in network directory by opening up two xterms/terminals and running two nachos processes (network id 0, 1) communicating with each other.

1. Implement lock and condition.
2. Go into network directory, gmake
3. Run nachos on two xterm/terminals using these commands:  
`nachos -rs 1234 -m 0 -o 1`  
`nachos -rs 1234 -m 1 -o 0`
4. You will observe the two processes sending messages and acknowledging.

**Phase 3: Ring Network: (5%)** Update the `nettest.cc` so that a set of nachos process with network ids (0, 1,2,3..) can communicate. To test this form a ring of at least three nachos processes representing three network nodes, node 0 sends message to node 1, node 1 receives and transmits the same message to node 2 and node 2 receives and transmits the message back to node 0 thus successfully completing a trip around the ring.

**Phase 4: (45%) Chat Server:** Implement the chat server that behaves according to the protocol described in Section 8. Test it with dummy data/hard coded data.

**Phase 5: (25%) Chat Client and Integration with Server:** Implement the chat client and a simple text interface and integrate it with the server.

**Phase 6: (10%) Documentation:**

This includes internal documentation (comments) and a **BRIEF, BUT COMPLETE** external document (read as: paper) describing what you did to the code and why you made your choices. **DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION.**

**7. Deliverables and grading**

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a `README.NOW` file and place it in the nachos *code* directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. I will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do anything to corrupt the system, and system calls should trap as many error conditions as possible.

**8. A Simple Chat Protocol**

**SERVERINFO**

```
cmd_serverinfo {
    byte  type  = 0;
    byte  cmd   = 0;
};

status_serverinfo {
    byte  type  = 1;
    byte  cmd   = 0;
    byte  status = N; // 0 = OK, 1 = FAIL
    byte  length = M; // Size of message
    byte[] message;
};
```

OUTPUT: [SERVER] – Copyright 2004 Your Name Here, CSE4/521  
OUTPUT: [SERVER] – ERROR : SERVERINFO Command Failed

### LOGIN <nickname>

```
cmd_login {
    byte  type  =    0;
    byte  cmd   =    1;
    byte  length =    N;
    byte[] nickname;
};

status_login {
    byte  type  =    1;
    byte  cmd   =    1;
    byte  status =    N;    // 0 = OK, 1 =FAIL
                          // 2 = Nickname in use
                          // 3 = Already logged in
};
```

OUTPUT: [SERVER] – User <nickname> logged in.  
OUTPUT: [SERVER] – ERROR : LOGIN error.  
OUTPUT: [SERVER] – ERROR : LOGIN nickname <nickname> in use.  
OUTPUT: [SERVER] – ERROR : LOGIN user already logged in.

### LOGOUT

```
cmd_logout {
    byte  type  =    0;
    byte  cmd   =    2;
};

status_logout {
    byte  type  =    1;
    byte  cmd   =    2;
    byte  status =    N;    // 0 = OK, 1 FAIL
                          // 2 = Not logged in
};
```

OUTPUT: [SERVER] – User <nickname> logged out.  
OUTPUT: [SERVER] – ERROR : LOGOUT error.  
OUTPUT: [SERVER] – ERROR : User not logged in.

### WHOSINROOM

```
cmd_whoroom {
    byte  type  =    0;
    byte  cmd   =    3;
};
```

```

status_whoroom {
    byte  type  =    1;
    byte  cmd   =    3;
    byte  status =    N;    // 0 = OK, 1 FAIL
};

```

```

data_whoroom {
    byte  type  =    2;
    byte  cmd   =    3;
    byte  entry =    count; // nickname count
    byte  pos   =    X;    // 0 = Middle, 1 = First, 2 = Last
    byte  length =    L; // message length
    byte[] msg;    // Single message Entry
};

```

OUTPUT: [SERVER] – ERROR : WHOSINROOM error.

OUTPUT: [SERVER] – User : <X> NickName: <nickname>

(As many data N data message representing N nicknames will be sent)

### **SEND <message>**

```

cmd_send {
    byte  type  =    0;
    byte  cmd   =    4;
    byte  length =    N;
    byte[] message;
};

```

```

status_send {
    byte  type  =    1;
    byte  cmd   =    4;
    byte  status =    N;    // 0 = OK, 1 FAIL
                                // 2 = Not in room
};

```

```

bcast_send {
    byte  type  =    3;
    byte  cmd   =    4;
    byte  length =    N;
    byte[] msg;    // Single message Entry
};

```

OUTPUT: [SERVER] – ERROR : SEND error.

OUTPUT: [SERVER] – ERROR : SEND not in room.

OUTPUT: [<nickname>] – <msg>

Send message is sent from a client to the server, which then broadcasts to all the clients.

### **WHISPER <nickname> <message>**

```

cmd_whisper {
    byte  type  =    0;
    byte  cmd   =    5;
};

```

```

    byte  nlength=    M;
    byte[] nickname;
    byte  length =    N;
    byte[] message;
};

status_whisper {
    byte  type  =    1;
    byte  cmd   =    5;
    byte  status =    N;    // 0 = OK, 1 FAIL
                            // 2 = Not in room
                            // 3 = nickname not found
};

bcast_whisper {
    byte  type  =    3;
    byte  cmd   =    5;
    byte  length =    N;
    byte[] msg;    // Single message Entry
};

```

OUTPUT: [SERVER] – ERROR : WHISPER error.

OUTPUT: [SERVER] – ERROR : WHISPER not in room.

OUTPUT: [\*<nickname>\*] – <msg>

## QUIT

This command on the server side shuts the server down and cleans up all the space. (On the client side Logout command itself will terminate the server program.)

**Important Note:** You are required to follow strictly the protocol given. This will allow us to test your sever with our client during the demo for the project. Please do not implement any more commands than specified above. You may not have time to do that. I would rather prefer you spend your time to implement the commands given in section 8.