# Design

## System Description

### Core Vision

Our goal is to build an instant messgaging system in which multiple clients can securely communicate as individuals as well as in a group chat. Additional functionality may be added as described below.

### Key Features

- The ability for individuals to message privately or in a group
- Chat generates session keys for each communication session between clients
- Messages can be sent and received in multiple sessions using the terminal
- Time permitting: Notifications letting users know when their messages have been delivered
- Time permitting: Read receipts which can be turned on/off per a user's preference

### Description

To implement our instant messaging system, we're going to use a client-server architecture in which the clients (users) send messages to other users by using a chat server to generate session keys that allow clients to communicate directly. Additionally, in the event that a client is not online, the chat server will store all messages intended for that client until the client reconnects to the chat server, at which point it will receive its messages. All sessions will be secured by using the Needham-Schroeder Protocol for generating symmetric keys.

In accordance with the Needham-Schroeder Protocol, clients will connect to the chat server with a pseudorandomly generated key, and they will be able to inform the chat server when they want to connect to another client in order to communicate. Upon receiving this request, the chat server will generate a session key for the two clients to communicate, and it will present both of them with the key. In the event that a client wishes to communicate with an offline client, the chat server will store any messages the first client (the sender) wishes to send in a B+ Tree, and then it will forward them on to the intended recipient when they log online.

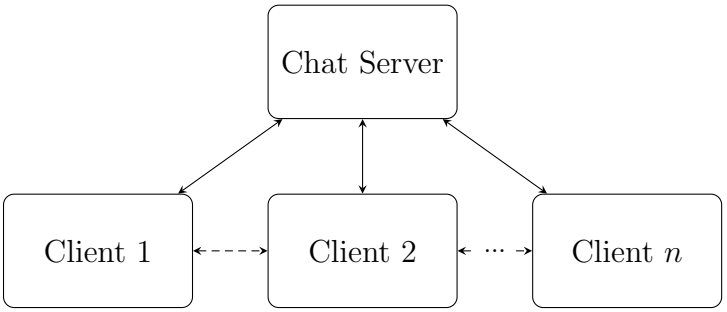
Communication will be handled using the TCP protocol as well as OCaml's implementation of the C-Socket API (the tools for doing this are found in the TCP and Unix modules). Additionally, we will use the Random module to generate pseudorandom numbers, and we will potentially be using the Sys module to enable reading in messages from the command line (the alternative is that we only have to use ANSITerminal). Additionally, we have intentionally omitted description of our group messaging implementation from this section, because Professor Clarkson has made it clear to us that we should focus on implementing simple two-way connections before embarking on an attempt to also implement group messaging.

As a final note, if it has not already been made clear, we expect that our final implementation of the chat server will differ in some way from this charter/design document. However, we believe that this document accurately outlines our core vision, the features we hope to implement, and the general ideas and design issues we will be considering when implementing our instant messaging system.

# Architecture

Our components will include a central Chat server which each Client must connect to in order to establish a unique server key to be able to send messages to other clients. Additionally, clients will be connected to each other as they obtain session keys to enable direct communication with another specific client or group of clients.

## Architecture Diagram:

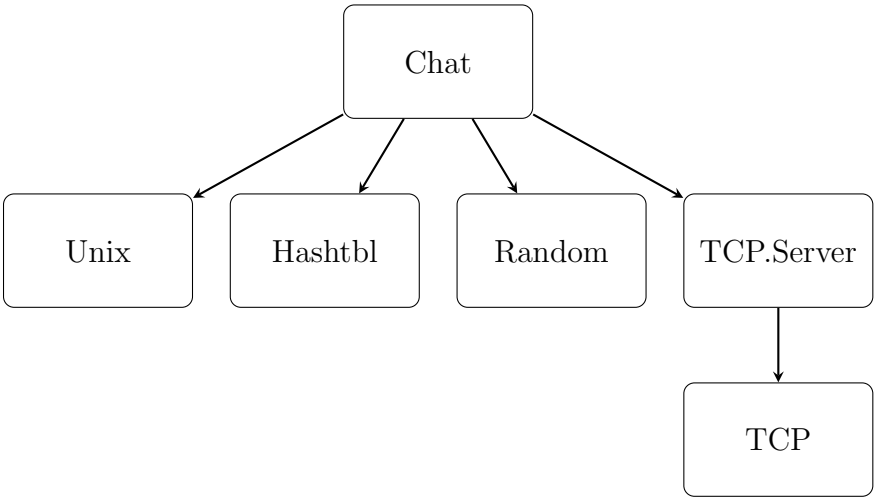


# System Design

## System Design MDDs:

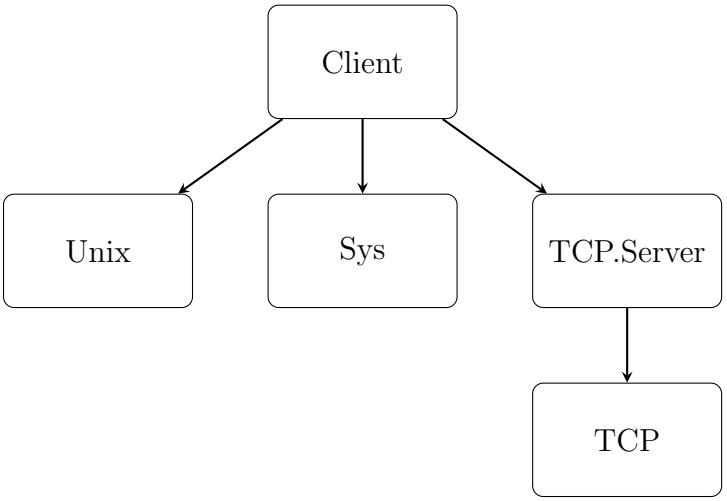
### Chat:

We need to implement a Chat server to connect clients to each other based on the client keys. This Chat server will have to be able to store messages in the event that the client intended to receive a message is not online at the time the sender sent the message. These must be able to be retrieved when the receiver is online again.




### Client:

The Client module will be able to take in input from the command line that are plain text messages to be sent to another client. Each Client has a Client key established with respect to the Chat server, and will establish session keys when communicating with other client(s).




# Module Design

# Data

### Chat:

- Hashtable of (Client, Client key) pairs
- B+ tree of (Session key, Message) pairs

### Client:

- Hashtable of (Client, Session key) pairs
- Client server key

# External Dependencies

- Unix
- TCP
- TCP.Server
- Sys
- Random

# Testing Plan

## 1)

We will verify that users (clients) have a functioning REPL for users to enter messages to peers.

## 2)

Our testing plans associated with connection and distribution will consist of four major portions:

### 2.1)

Testing of initial client-server binding using a pseudorandom key generator. This will be accomplished by verifying that the server has received the correct socket address from client 1 when a connection has been initiated between two peers utilizing a server connection. We will then verify that the clients have received a valid asymmetric key that will connect them to the server. We will also verify that the individual keys have been created using a valid hash key function.

### 2.2)

Testing of the formulation of the shared client session key. The exact calculation of the session key will be tested by proving that our calculation is communicative and reproducible among peers. Also as described in 'System Description', the session key will function as a symmetric key that will be shared by both clients. Thus, testing will verify that both clients contain an identical static session key throughout connection.

### 2.3)

Testing of the socket connection between the two clients. Since the server will be relaying client 2's socket address to client 1, after key construction, we will make sure that the server relay is valid and client 1 receives and binds to the correct socket distributed by client 2. The (modified) connect function will also be tested by verifying that it receives the valid session key shared by the clients.

**2.4)**

We will test the actual message distribution between clients. This will simply be tested by verify that the clients receive the other client's message correctly and fully.

## 3)

Verify that the buffer of the server maintains outgoing messages that will be later received by a client not currently online. We will also validate that the server correctly distributes the messages once possible (receiving client connects to server).

## 4)

Test that the termination of messaging connection is completed gracefully once a client(s) leaves.

## 5)

Finally, we will be testing possible error cases such as unexpected dropping of clients from the server or peers during communication, server dropping, peer communication when receiver is not online, message buffer overflow, and invalid key generation.

*Note:* If implemented, we will also be handling the testing of group communication. Group communication testing will be very similar to the single peer to peer communication, however we will validate that the server is correctly handling multiple peer connections and session key distribution.