Select Example: Simple Echo Protocol

Note that you DO NOT need to greet clients in your lab 4 implementation.  This is only here for example purposes!

Echo Protocol:

1) Client connects.

2) Server sends a greeting. (Not recommended / necessary for your lab!)

3) Client sends a string.

4) Server echoes string back.



Figure describing an FSM (Finite State Machine) of the ***Server States per client***.

The server has four states: **Wait, Echo, Greet and Disconnected.** The server spends most of its time in the **Wait State** when

the client sends no data.

1. **Transition: Wait - Echo – Wait**

When the server is in the waiting state, it can only transition to Echo or Disconnected. When the client sends a string,

the server transitions to the **Echo State.** The server then echos the string and transitions back to the **Wait State.**

1. **Transition: Wait/Greet/Echo - Disconnected.**

When the client issues a disconnect message, the server transitions to **Disc. State** for this client from any of the states it

currently is in. The client can disconnect at any time, and therefore can transition to the disconnected state from any state.

1. **Transition: Disconnected – Greet**

The server can only transition from the Disconnected to Greet State. When a client connects for the first time,

the Server transitions from disconnected to connected, and sends a greeting to the Client.

1. **Transition: Greet-Wait**

Finally, from the Greet state the server can transition to waiting or disconnecting. Once the server sends a greeting to

the client, the server transitions into wait state.

**Pseudo-code and text example on how to use select to implement our Echo Protocol:**

typedef struct {

  int connected; // 0 - Not connected. 1 - Connected.

  int greeted; // 0 - Not greeted. 1 - Greeted.

  int needs\_echo; // 0 - No. 1 - Yes.

  char data[1024];

  int offset;

} client;

**Main:**

client clients[MAX\_CLIENTS];

memset(clients, 0, MAX\_CLIENTS \* sizeof(client));

server\_sock = bind(), listen();

**Select loop - Phase 1**: populate sets with sockets we want to check. Always add server socket.

fd\_set rfds, wfds; //declare a set of read and write file descriptors

int i;

int maxfd = server\_sock;

FD\_SET(server\_sock, &rfds); // add your server socket to the list of read file descriptors

for (i = 0; i < MAX\_CLIENTS; i++) { //loop through all the clients and add them to the appropriate set.

  if (clients[i].connected && clients[i].greeted) {

    FD\_SET(i, &rfds);

    if (i > maxfd) maxfd = i;

  }

  if (clients[i].connected && (!clients[i].greeted || clients[i].needs\_echo)) {

    FD\_SET(i, &wfds);

    if (i > maxfd) maxfd = i;

  }

}

maxfd += 1;

**Select loop - Phase 2: Call select()**

int result = select(maxfd, &rfds, &wfds, NULL, NULL); // call select.

If none of the requested operations are ready, select will block. When select returns, it will modify these sets.

**Select loop - Phase 3**:

Check the sets. For each socket, if it's still in the set after select returns, it's safe to read/write ONE TIME.

for (i = 0; i < maxfd; i++) {

  if (FD\_ISSET(i, &rfds)) {

    if (i == server\_socket)

      accept\_new\_client(i);

    else

      read\_client(i);

  }

  if (FD\_ISSET(i, &wfds)) {

    if (!clients[i].greeted)

      send\_greeting(i);

    else

      send\_data(i);

  }

}