Comp 446 Project Presentation

By: Salvatore D’Agostino 9079661
Michael Di Loreto 9106979

Open **P**ersonal **I**nstant **M**essenger (OPIM)

INTRODUCTION

For our project we decided to focus on network communication over a private network and develop our own instant messaging protocol. Our goal was to develop this protocol using 2 parts; the server and the client. To achieve these results we knew that we would need to incorporate many components including SOCKETS for communication over networks, XML to facilitate proper data processing, a database to store all relevant information and finally a GUI to supply a front-end for users running the client. Our platform of choice to implement this protocol was LINUX.

When we started researching and conceptualizing our project we also began choosing specific applications to be used for each component of our design. Initially we would use the LINUX sockets system libraries for network communication, the libxml C++ api for XML parsing, the mysqlcppapi C++ api for MySQL and the Glade IDE for our GUI. After establishing this tool set, we began to conceptualize our custom protocol by attempting to write sample programs for each component.

The first component we attempted to implement was SOCKETS in the form of a simple echo server. After giving some thought to eventually expanding this program into our protocol, we noticed that we had to make our first obvious addition to our components tool set by adding in threads. This addition would allow us to achieve multiple connections between the server and each individual client. This task proved to be more ambitious than originally planned. The main reason was having threaded functions within an object. We tried many different approaches to solve this problem including templates until finally discovering the boost C++ library, which allowed us to run member functions of any object type inside a thread. We then wrote or own delegate thread class.

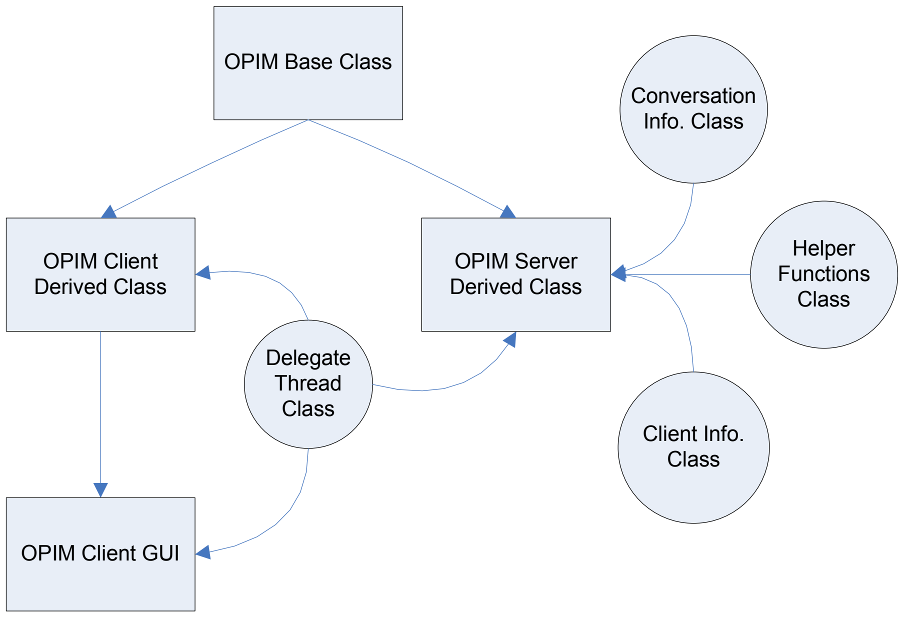
Secondly after further contemplation of the ease at which to implement our protocol we decided to use sqlite instead of MySQL. We found that sqlitewrapped was a suitable replacement to the mysqlcppapi C++ api.

After writing a successful XML test program, using libxml2 and beginning work on our GUI, we came to the realization that the Glade IDE was really not appealing to us. Instead we discovered the Qt GUI library for C++ and immediately decided to use that in its place.

Now each piece of the protocol had working sample program and we could finally begin connecting them together to from our final working OPIM chat software!

DESIGN

OPIM Base:
Our base design is a base class containing all the basic definitions of our protocol such as SOCKETS and protocol commands, which are inherited by the client and server (derived classes). Both the client and the server use an instance of the delegate thread class as a component to achieve threading. The GUI for the client operates on top of the client (derived class).

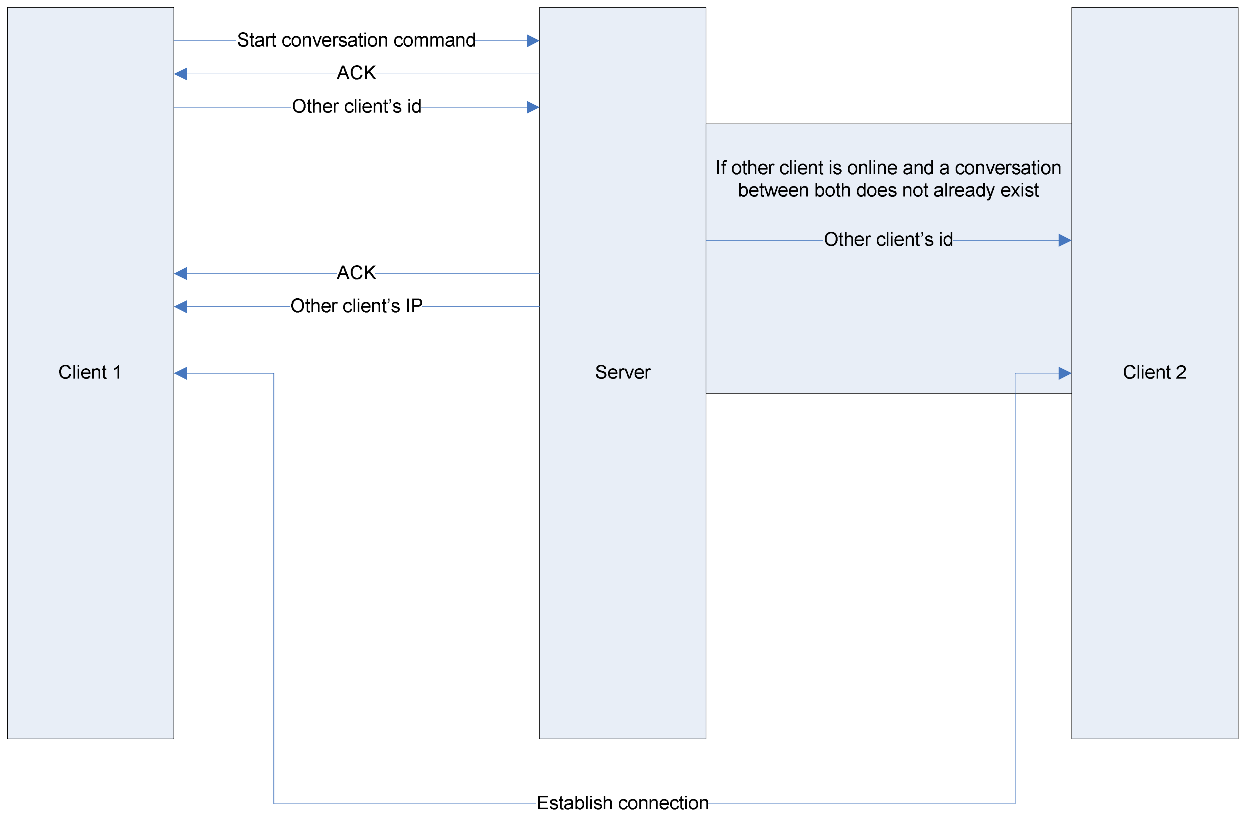


Communication between the server and the client is first created by the client making a connection to the server who is listening for incoming connections. Once this connection is established the server listens for an initial command from the client. If the command is recognized within the protocol the server sends an acknowledgment back to the client. At this point their communication can continue or they can disconnect depending on the command sent. Either way the server will again listen for another incoming connection from a new client. If an unrecognized command was sent the server closes the client’s connection and begins listening anew.

If the connection is to be kept between the client and server then the server sends a list of currently online users to the new client and adds the client to its list of online users. The server then waits for the next command from the client in an infinite loop until the client disconnects.

If the client wants to begin a chat with another online client they send the appropriate command to the server. The server then does the proper validation and if everything checks out, the server sends a start conversation command to the other. The passive client then waits for the active client to connect to them. The server then sends an acknowledgement back to the active client who then setups up a two-way connection with passive client thus ruling out the server from their conversation.

Conversation:


Start conversation command
ACK
Other client's id
If other client is online and a conversation between both does not already exist
Other client's id
ACK
Other client's IP
Client 1
Server
Client 2
Establish connection

Threads and Boost:
We designed our own thread class using Boost's function type from the Boost libraries. This enables us to achieve threading inside member functions of any object type thus allowing us to properly implement our protocol.

XML:
We use libxml2 to send large protocol messages from one end to another. Basically it provides a structured approached to sending large amounts of parse-able data in one single message.

Database:
We use sqlitewrapped only on the server side to store all relevant information. This information includes all users registered, all users currently online and a list of various user statuses (i.e. online, away, busy, etc).

Qt:
We use the Qt4 Designer to create our GUI for the client and then the Qt4 dev libraries to implement and link the GUI to our client backend.

IMPLEMENTATION

We implemented our base class by creating the “opim” class:

```
class opim
{
	public:
		enum Cmds { discon = 0, disconFlood = 8, nak, ack, reg, login, startConvo,
				endConvo, say, fullList, addClient, removeClient, editClient };
		enum status { online = 1, offline = 2, away = 3, busy = 4 };
		opim( void );
		virtual int setupSocket( sockaddr_in& );
		virtual int receiveCmd( int );
		virtual void sendCmd( int, int );
		virtual string receiveMessage( int );
		virtual void sendMessage( int, string );
		virtual int listenFor( int, sockaddr_in& );
		virtual int connectTo( const char*, sockaddr_in& );
};
```

We then used this base class to create our derived classes “opimServer” and “opimClientGUI”:

```
class opimServer: public opim
{
	public:
		opimServer( void );
		virtual ~opimServer( void );
		void bringOnline( void );
		void startServer( void );
		void* acceptClients( void* );
		void* handleClient( void* );
		void* serverCLI( void* );
		void updateClientsLists( int, Client* );
		void print();
	private:
		void setupDB();
		bool notInAConvo( Client*, Client* );
		Client* getClientById( int id );

		Database::Mutex _dbMutex;
		Database* _db;
		StderrLog _dbLog;
		vector<convo*> _convos;
		vector<Client*> _clients;
		int _socket;
		sockaddr_in _myAddr;
		bool _online;
		delegateThread* _dt;
};
```

```
class opimClientGUI : public QDialog, private Ui::Login, public opim
{
	Q_OBJECT

	public:
		opimClientGUI(QWidget *parent = 0);

	public slots:
		void getLogin();
		void sendRegister();

	private slots:
		void sendLogin(string&, string&, string&);

	private:
		QString user;
```

```
                QString password;
                opimUserList* chatWindow;
};
```

Here is how we implement the initializing of a conversation between two clients and the server, which shows our OPIM SOCKETS and sqlitewrapped functionality:

```
sendCmd( client->socket(), ack );

int clientId = atoi( receiveMessage( client->socket() ).c_str() );

cout << client->name() << " wants to talk to client " << clientId << endl << flush;

if ( clientId != client->id() ) //not starting a conversation with themselves?
{
        Query q( *_db );

        q.get_result( makeSql( "select 1 from connected_users where user_id = ?", longToString(
                      clientId ).c_str() ) );

        if ( q.num_rows() ) //other client online?
        {
                pthread_mutex_lock( &_clientsMutex );

                Client* otherClient = getClientById( clientId );

                if ( otherClient != 0 ) //client found in clients list?
                {
                        if ( notInAConvo( client, otherClient ) )
                        {
                                convo* c = new convo();

                                c->addClient( client );
                                c->addClient( otherClient );
                                _convos.push_back( c );
                                sendCmd( otherClient->sendSock(), startConvo );
                                sendCmd( client->socket(), ack );
                                sendMessage( client->socket(), otherClient->ip() );

                                cout << "Conversation started between " << client->name() << " and
                                        " << otherClient->name() << endl << flush;
                        }
                        else
                        {
                                sendCmd( client->socket(), nak );
                                sendMessage( client->socket(), "Already in a conversation" );

                                cout << client->name() << " already in conversation with " <<
                                        otherClient->name() << endl << flush;
                        }
                }
                else
                {
                        sendCmd( client->socket(), nak );
                        sendMessage( client->socket(), "Client is not online" );

                        cout << "Other client is not online\n" << flush;
                }

                pthread_mutex_unlock( &_clientsMutex );
        }
        else
        {
                sendCmd( client->socket(), nak );
                sendMessage( client->socket(), "Client is not online" );

                cout << client->name() << " other client is not online\n" << flush;
        }

        q.free_result();
}
```

```
else
{
       sendCmd( client->socket(), nak );
       sendMessage( client->socket(), "You cannot start a conversation with yourself" );

       cout << client->name() << " tries to start a conversation with themself\n" << flush;
}
```

Here’s an example of how we implement our delegateThread class to permit the server to begin listening for incoming client connections:

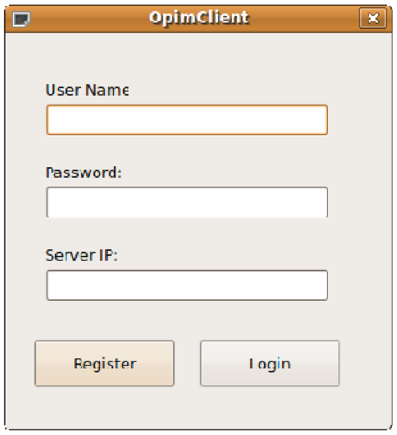
```
void opimServer::startServer( void )
{
       if ( _online ) //server online?
       {
              //starting the server threads.
              cout << "Starting acceptClients thread...\n" << flush;
              pthread_t thAcceptClients =
                     _dt->startThread( boost::bind( &opimServer::acceptClients, this, _1 ),
                                  (void*)0 );

              cout << "Server has started!\n" << flush;

              //waiting for all threads to die.
              pthread_join( thAcceptClients, NULL );
       }
       else
              cout << "Error: server has not been brought online." << flush;
}
```

Here’s an example of how we implement libxml2 for adding a new client to the users list via the parsing of an XML message:

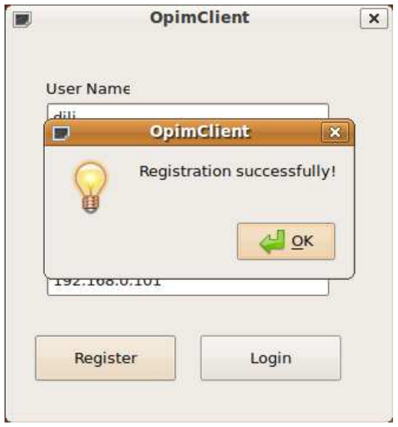
```
string xml = receiveMessage( _rSock );

cout << xml << endl << flush;

x = xmlReadMemory( xml.c_str(), xml.length(), "user", NULL, 0 );
root = xmlDocGetRootElement( x );

cout << "Adding to list: " << (const char*)root->children->children->content << ", " << (const
        char*)root->children->next->children->children->content << endl << flush;

listUsers->addItem(
       (QListWidgetItem*)(new clientItem( atoi( (const char*a)root->children->children->content
       ), (const char*)root->children->next->children->content ) )
);

xmlFreeDoc( x );       // free document
xmlCleanupParser();    // Free globals
```

APPLICATION

Here are some screenshots of our OPIM software. First, we have the Login/Register screen. This screen is the first screen that the client is greeted by when they run a copy of the client:

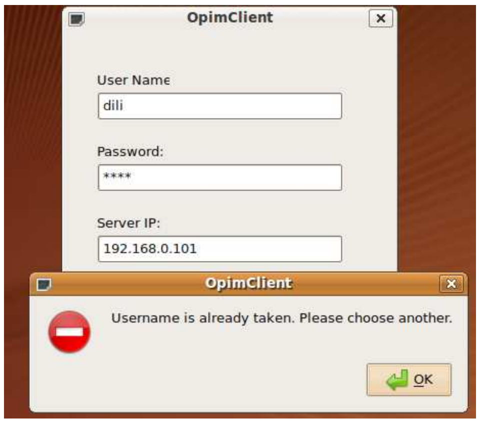


Once they Register a unique user name and password, they are greeted with this pop-up dialog indicating that their registration was successful (Note: They must have the IP of the server that they will be connecting to):



Registration successfully!

If they failed to register a unique username and/or password, they are greeted with this pop-up dialog telling them that the registration was un-successful:



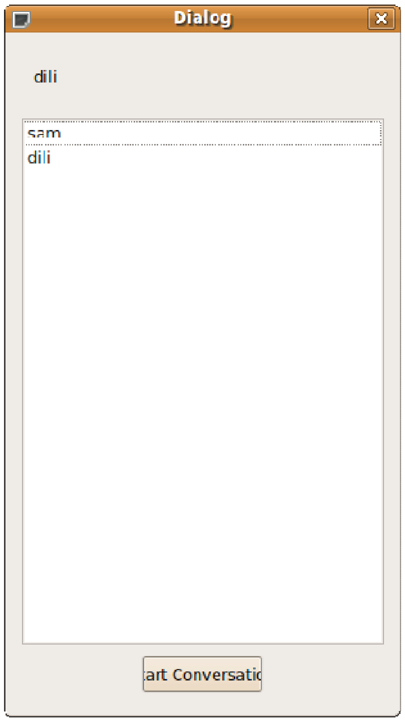
dili

****

192.168.0.101

Username is already taken. Please choose another.

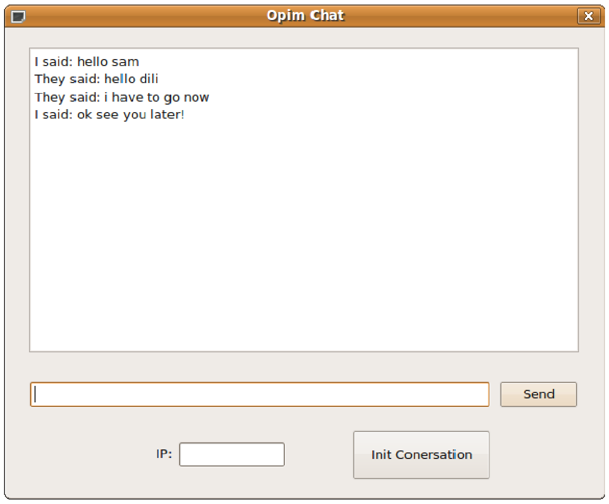
Once registered, the user can now login to the server. All they need to do is input their username, password and IP of the server they wish to connect to. Once logged in, the user will now receive a contact list window showing them who is currently online:



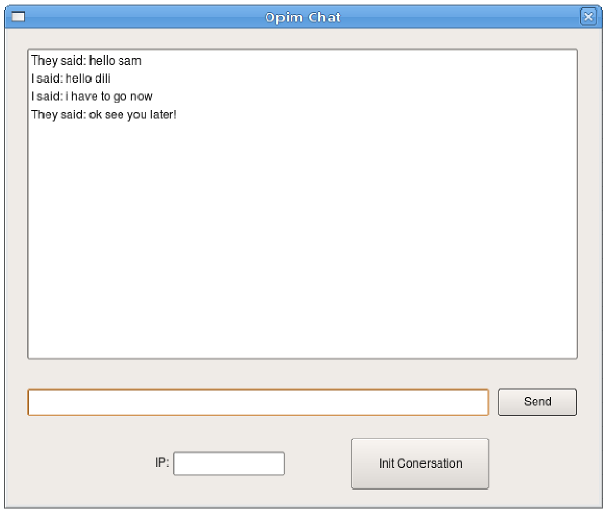
dili

sam
dili

If they wish to chat with a user who is online, they simply highlight that contacts name, and click "Start Conversation". If the connection to the other user is successful, each user will get a chat window where they can communicate with each other:



I said: hello sam
They said: hello dili
They said: i have to go now
I said: ok see you later!



They said: hello sam
I said: hello dili
I said: i have to go now
They said: ok see you later!

Conclusion & Future Work

This project has been a very insightful exercise for us. It gave us a realistic idea of how powerful and scalable a C++ program can be. We had worked with C++ in the past, but never used it at this level. It was our first time using tools like boost, incorporating things like XML and a database with C++ and implementing a Graphical User Interface to work with our code behind. That was probably the coolest part. We also learned a great deal more about threads and how to properly use them with synchronization (needed for things like the contact list). Also without inheritance, our project would have taken forever to create and would have probably had some issues with things like the GUI.

As for our project OPIM, we are hoping to continue development by adding many new features and fixing/tweaking current features. Some of the things we want to do are things like: file transfers between clients, a proper client list with customizable statuses, multi-user conversations, etc. Those are some among many other features that came to mind. At the same time, we would also like to improve on some security issues to help make the protocol fool-proof and hack-proof. This will probably be the most time consuming part, but nevertheless it is very important. We kinda skipped out it for now due to the lack of time, but we are well aware of the importance of security in a software application. Currently our XML implementation does not support exceptions, which is another key feature that we would like to address. Also having connection timeouts to unresponsive clients and/or servers would help to improve the functionally and stability of our protocol.

Overall we are quite happy with the results that we achieved. We admit that it would have been nicer to have some things work better than they do, but with the amount of problems that we ran into during development, being able to overcome these and still have a decently working application and protocol is still satisfying. One of the aspects about our client that we find to be pretty neat is that it is also cross-platform! We have tested running a client coded in VB.NET and have successfully connected to the server running on a Linux machine and communicated with a client also running on a Linux machine. This has given us some idea as to how we can implement our protocol to connect many clients on all kinds of platforms.

APPENDIX

Client.h:

```
class Client: public opim
{
	public:
		Client( Database*, int, sockaddr_in );
		virtual ~Client();
		bool registerMe();
		bool login();
		string name();
		int id();
		const char* ip();
		bool loggedIn();
		int status();
		int socket();
		void changeStatus( int );
		void sendListUpdate( int, Client* );
		void setSendSocket( int );
		int sendSock();
		string xml();

	private:
		sockaddr_in _myAddr;
		int _id;
		string _name;
		string _password;
		Database* _db;
		int _status;
		int _socket;
		int _sendSock;
		bool _loggedIn;
};
```

convo.h:

```
class convo
{
	public:
		convo(){ }
		bool inConvo( Client*, Client* );
		bool inConvo( Client* );
		void addClient( Client* );

	private:
		vector<Client*> _clients;
};
```

delegateThread.h:

```
#ifndef __DELEGATE_THREAD_H__
#define __DELEGATE_THREAD_H__

#include <boost/function.hpp>
#include <boost/bind.hpp>
#include <pthread.h>

using namespace std;

typedef boost::function<void* (void*)> func;

struct threadNfo
{
	func f;
	void* args;
};
```

```
class delegateThread
{
	public:
		virtual pthread_t startThread( func, void* );
		delegateThread( void* obj );

	private:
		static void* doStartThread( void* );
		void* _obj;
};

#endif
```

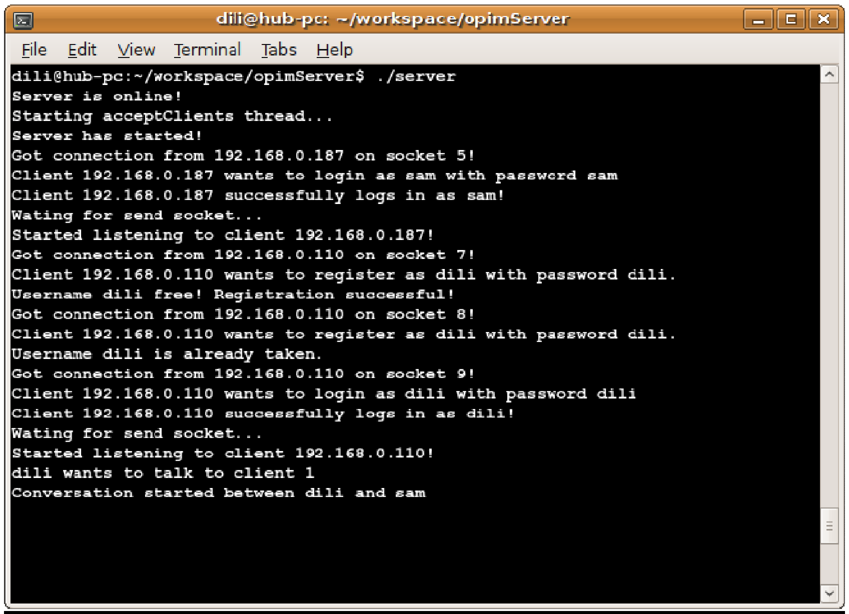
helpers.h:

```
#ifndef __HELPERS__
#define __HELPERS__

#include <string>
#include <sqlite3.h>
#include <libsqlitewrapped.h>
#include <sstream>

using namespace std;

namespace helpers
{
     template<typename T>
     size_t arraySize( T );
     string longToString( const long& );
     bool isNumeric( string );
     string safeQuote( string );
     string makeSql( string, ... );
}

#endif
```

opimClientGUI.cpp (constructor and register portion):

```
opimClientGUI::opimClientGUI(QWidget *parent) : QDialog(parent)
{
     setupUi(this);

     connect(btnRegister,SIGNAL(clicked()),this, SLOT(sendRegister()));
     connect(btnLogin,SIGNAL(clicked()),this, SLOT(getLogin()));
}

void opimClientGUI::sendRegister()
{
     string u = txtUserName->text().toStdString();
     string p = txtPassword->text().toStdString();
     string ip = txtIp->text().toStdString();

     /*Empty Feild*/
     if (u == "" || p == "" || ip == "")
     {
          QMessageBox::critical( this, "OpimClient", "Please enter a username, password and
                                 server IP." );
     }
     /*Send login info to Server!*/
     else
     {
          sockaddr_in theirAddr;
          int sSock = connectTo(ip.c_str(),theirAddr);
          int cmd;
          string message;

          sendCmd(sSock,reg);

          cmd = receiveCmd(sSock);

          if(cmd == ack)
          {
               stringstream ss;

               ss << "<register><username>" << u << "</username><password>" << p <<
                     "</password></register>";
               sendMessage(sSock,ss.str());
               cmd = receiveCmd( sSock );

               if(cmd == ack)
                    QMessageBox::information( this, "OpimClient", "Registration successfully!" );
               else
               {
                    message = receiveMessage( sSock );
                    QMessageBox::critical( this, "OpimClient", message.c_str() );
                    cout << message << endl << flush;
               }
          }
          else
               cout << "Bad startup command" << endl << flush;
     }
}
```

Server debug from example in "APPLICATION" section:


dili@hub-pc: ~/workspace/opimServer
File Edit View Terminal Tabs Help

```
dili@hub-pc:~/workspace/opimServer$ ./server
Server is online!
Starting acceptClients thread...
Server has started!
Got connection from 192.168.0.187 on socket 5!
Client 192.168.0.187 wants to login as sam with password sam
Client 192.168.0.187 successfully logs in as sam!
Wating for send socket...
Started listening to client 192.168.0.187!
Got connection from 192.168.0.110 on socket 7!
Client 192.168.0.110 wants to register as dili with password dili.
Username dili free! Registration successful!
Got connection from 192.168.0.110 on socket 8!
Client 192.168.0.110 wants to register as dili with password dili.
Username dili is already taken.
Got connection from 192.168.0.110 on socket 9!
Client 192.168.0.110 wants to login as dili with password dili
Client 192.168.0.110 successfully logs in as dili!
Wating for send socket...
Started listening to client 192.168.0.110!
dili wants to talk to client 1
Conversation started between dili and sam
```