John Quinn
Robert Wirthman
CSE 30332
Final Project Report

Abstract:

This report details the process of transforming NDIM (Notre Dame Instant Messenger) from an idea to a fully functional application. First, the need for NDIM is explained in "The Problem." Next, how we created the application, including the RESTful protocol, is detailed in "The Implementation." In "The Proposal" we identify how the implementation differs from what was originally planned. The last three sections, "Future Improvements," "In Retrospect," and "Time Restraints" provide many of our reflections on our finished product. Here, we discuss how we could improve on what we already have given more time and any mistakes we may have made. Finally, an appendix of screenshots is provided.

The Problem:

NDIM is an instant messenger client. It allows users to send messages to each other through a server and supports both one-on-one and group chat. The program aims to allow users to easily create conversations, add people to conversations, and send messages. Users should be able to see other users who are online and available for chat. They should be able to send messages to other users, even if they are offline. In the case where the other user is online, they should receive this message very quickly. In the case where the other user is offline, they should receive this message upon logging into the system. A system like this should be kept secure, so that users cannot pretend to be other users and gain access to their information and messages. In order to give the user control over whom they can talk to, a robust friend system was included in the design. Users can friend, un-friend, block, and un-block other users. Requests to add users to any friends lists must be approved by both parties. Because many accounts will be created, there should be some way to save account information to a file, in case the server goes offline.

The Implementation:

NDIM was implemented using a client-server model. The client portion of the application was a GUI written in PyQt, while the server portion was written in CherryPy. When the program runs, one server is launched, and many clients are launched, connecting to that server.

The server was designed as a RESTful web service. The protocol for this service is defined in the table below:

Resource	GET	PUT	POST	DELETE
/users/	Return a list of all usernames.		Add a new user.	
/users/[username]			Attempt to login.	

/friends/		Change the friend status for a pair of users.	Add friend.	
/friends/[username]	For a given user, return a list of tuples of the form (username, friend status).			
/messages/		Add content to a given message.	Create a new message.	
/messages/[username]	Return a list of messages that a given user can access.			
/messages/[message id]		Return the contents of a given message.	Add or remove a given message from a user’s message list.	
/statuses/[username]	Return the status of a given user.	Change the status of a given user.		
/backups/	Saves users and friends to users.dat and friends.dat, respectively.			

The service accepts and returns request in JSON format. The field ‘result’ should contain ‘success’ if the request was successful and ‘failure’ otherwise.

GET /backups/:
{"result": "success"}

GET /statuses/phillips:
{"status": "offline", "result": "success"}

PUT /statuses/phillips with {"status":"online"}:
{"result": "success"}

PUT /messages/0 with {}:
{"content": "robert: What’s up", "usernames": ["phillips", "robert"], "result": "success"}

POST /messages/0 with {"username":"robert","action":"add"}
{"result": "success"}

GET /messages/Phillips:
{"message_ids": [0,1,2], "result": "success"}

POST /messages/ with {"usernames":"[robert,phillips]"}:

{"id": 1, "result": "success"}

PUT /messages/ with {"username":"robert","id":"1","content":"What's up?"}:
{"result": "success"}

GET /friends/phillips:
{"usernames": [["robert", "friends"], ["douglas", "friender"]], "result": "success"}

POST /friends/ with {"username":"robert","friendname":"douglas"}:
{"result": "success"}

PUT /friends/ with {"username":"robert","friendname":"douglas","action":"unblock"}:
{"result": "success"}

POST /users/patrick with {"password":"123456aB"}:
{"result": "success"}

GET /users/:
{"usernames": ["phillips", "robert", "jamarquis", "douglas", "patrick"], "result": "success"}

POST/ users/ with {"username":"patrick","password":"123456aB"}:
{"result": "success"}

Using this API, the client application can communicate with the server to achieve all the functionality previously described. When the user launches the program, they get a login screen. If they choose to login with an existing account, the /users/:user POST method is used to attempt to authenticate the session. If it succeeds, the user is logged in. If not, the server returns an error message, which is displayed to the user in a QMessageBox widget. If the user does not have an account, they can create one via the login screen. When they attempt to create an account, the /users/ POST method is called. This method checks if the user name is available and if the password meets the criteria specified by the program. The user will then receive a message telling them whether or not the account creation was successful.

When the user successfully logs in, their status is set to indicate that they are online, through the /statuses/:user PUT method. A QMainWindow subclass is instantiated, containing a few menus and a list of the user's friends. There is also a label telling the user which friends are online. The list of user friends is retrieved through the /friends/:user GET method. The status of each friend is got by calling the /statuses/:user GET method. On the server, each user has a list of tuples that represents their friends and their relationships with other users. The first part of the tuple is the friends username, and the second part of the tuple is the relationship between the friends. The value 'friender' means the current user has an outgoing friend request to the other user. The value 'friended' means the current user has an incoming friend request from the other user. The value 'friends' means that one user has sent the other user a friend request and that request was accepted. The value 'blocker' means the current user has blocked the other user, and the value 'blocked' means the current user has been blocked by the other user. These values are set by functions in the /friends/ controller.

The three menus in the main window are called "Buddies", "Chat", and "Logout". Under the "Buddies" menu, the user has several options pertaining to the management of their friends. Each of these options causes the creation of a modal QDialog that handles the user input. There is an option to add a friend called "Add Buddy". In this window, the user has a text box that they can enter the friend's username into. When the "Add" button is clicked, the /friends/ POST method is called, which sends the user a friend request. There is also an option to remove a friend called "Remove Buddy". This dialog allows the user to select a friend from a list of their friends for removal. Here, the /friends/ PUT method is called to modify their relationship.

There are also options called "Block Buddy" and "Remove Buddy" which allow the user to block and unblock friends. These options allow the user to designate people that they don't want to talk to ever again. Each of these options is handled by a modal dialog and call the /friends/ PUT method for modifying relationships. The last option on the "Buddies" menu is the "View Requests" option. This option opens a modal dialog that shows the user's incoming and outgoing friend requests. They can cancel outgoing requests, provided they haven't yet been accepted, and accept incoming requests. Again, the /friends/ PUT method is used to modify the relationships here.

The second menu, "Chat", allows users to set up a chat. Chats can either be setup via this menu, or by double clicking on the name of a friend in the Buddy List. If the chat is setup through the menu, a dialog allows the user to select multiple friends to add to the chat. When chats are started, non-modal dialogs are opened. Through various buttons and widgets, these dialogs allow the user to see the current content of the conversation, send messages, add new users to the conversation, and leave the conversation. In the QMainWindow subclass, a timer is setup so that the server is queried for message updates every half of a second. If the user has received a new message, a new dialog is opened, if one does not already exist for that particular message. If a dialog is already opened, the conversation content is updated. This timer is also used to update the statuses of users. Therefore, if a user's friends have come online since they logged in, their statuses will be updated. New messages are created through the /messages/ POST method and content is added through the /messages/ PUT method. When the client requests the contents of the message from the server, the /messages/:id PUT method is used. The /messages/:id POST method is called when the user requests to add someone else to the conversation or remove themselves from the conversation.

The last menu in the main window is the "Logout" menu. This menu allows the user to end their session. The /statuses/:user POST method is used to set the user's status to offline here. Then, the /backup/ GET method is called, backing up server data on users and friends to text files for future use.

Finally, a note on how we implemented encryption. The client, ndim.py, instantiates a global instance named encryptor of class Encryptor. In sendMessage, before the string text is included in the dictionary messagedata and sent via request to the server, it is encrypted. The following command handles the encryption: “text = encryptor.caesar_cipher_encrypt(7,text).” The member function, caesar_cipher_encrypt(shift,string), of class Encryptor, implements the Caesar Cipher. The Caesar Cipher, “is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed positions down the alphabet” (Wikipedia). Thus, the message is stored encrypted on the server. In receiveMessage, before the string is displayed in the GUI, it is decrypted. The following command handles the decryption: “text = encryptor.caesar_cipher_decrypt(7,text).” The member function, caesar_cipher_decrypt(shift,string), of class Encryptor, implements reversing the Caesar Cipher. We wrote the Encryptor class ourselves.

The Proposal:

Our original proposal in many ways was abstract. We proposed to make an instant messenger service, but only identified four specific features. Those features were the ability to send/receive messages, maintain a contact list, use emoticons and encrypt messages. We said to implement emoticons we would use regular expressions. For the rest of the features, we did not give implementation details. This was probably for the best as the only way in which our project differs from our proposal is how we implemented emoticons. We did not use regular expressions. Instead, we used a built in python pattern matching function. Finally, we indicated that the server would be developed with CherryPy and the client with PyQT. The final implementation uses both.

Professor McMillian reviewed our initial proposal and had several suggestions to increase the complexity of the project by adding more features. Those features were private and group messages, setting and querying the statuses of users, adding and removing buddies, logging in and out, adding and deleting users, and implementing encryption as a plugin. For the most part, the specifics on how to implement each feature were left out. In the end, we implemented each feature thus fulfilling all our responsibilities as indicated in both the original and revised proposals. If there is one exception, it is with how we implemented encryption. Instead of building encryption into a plugin system, we created a single class called Encryptor. The class has two methods, one for encrypting a message and another for decrypting. The algorithm we used was the Caesar cipher. It is a very basic encryption, but it gets the job done.

Future Improvements:

Our solution is not very scalable. Each client pings the server at least once per second. Imagine trying to launch hundreds or thousands of clients at once with our service. It would fail instantly. To improve our project, we could make it massively scalable. Instead of one server, have many that the clients can connect to. A major issue with this theoretical improvement is data replication across all the servers. This conveniently leads to another talking point about how our service can be improved: data management. Currently, all of the data on the server is stored into five dictionaries.

Name	Key	Value
users	username	password
friends	username	list of tuples of the form (username, friend status)
users_messages	username	list of message ids
messages	message id	message content
statuses	username	status

The username is the key for 4 of the 5 dictionaries. How these dictionaries are setup is very similar to how a relational database would work with primary-foreign key mappings. If the server implemented data storage with a SQL database instead of dictionaries, chances of successfully scaling our project would be significantly improved.

In Retrospect:

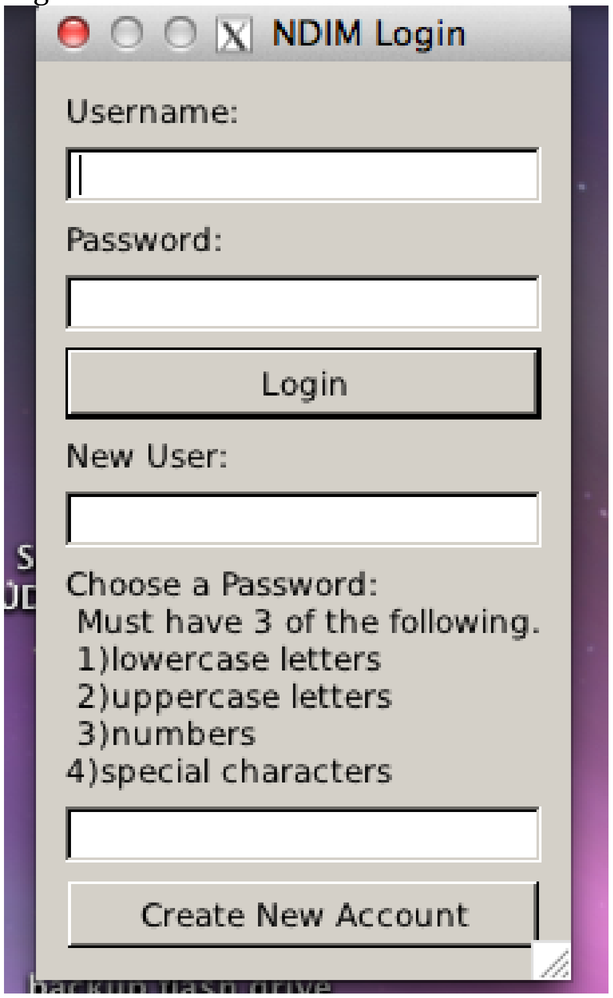
This project is perfect for a two person team. One can implement the client, or frontend, and the other can implement the server, or the backend. This is how we decided to divide the work. Throughout the development process we were both satisfied with the division of labor. It allowed us to feel as if we were both working on independent projects that we had complete control over. Even so, it may have been beneficial if both of us sat down and agreed upon the RESTful protocol before a single line of code was written. How we actually did it was the partner developing the server gave the other partner the protocol when it was finished. Then, as the other partner developed the client needed additional data, the protocol was changed on the fly. Although these changes were minimal, they could have been avoided entirely.

Time Restraints:

Currently, our solution is fully functional. With more time, we could continue to add features. We think our project is similar to Facebook. Today, Facebook works great, but that doesn't mean their developers will stop innovating and rolling out new updates. During development, we discussed a wish list of features that just weren't possible given the time restraints. There were some that were incredibly ambitious like video chat. Then there were more reasonable ideas like improving security. Ultimately, more time would equate to more features.

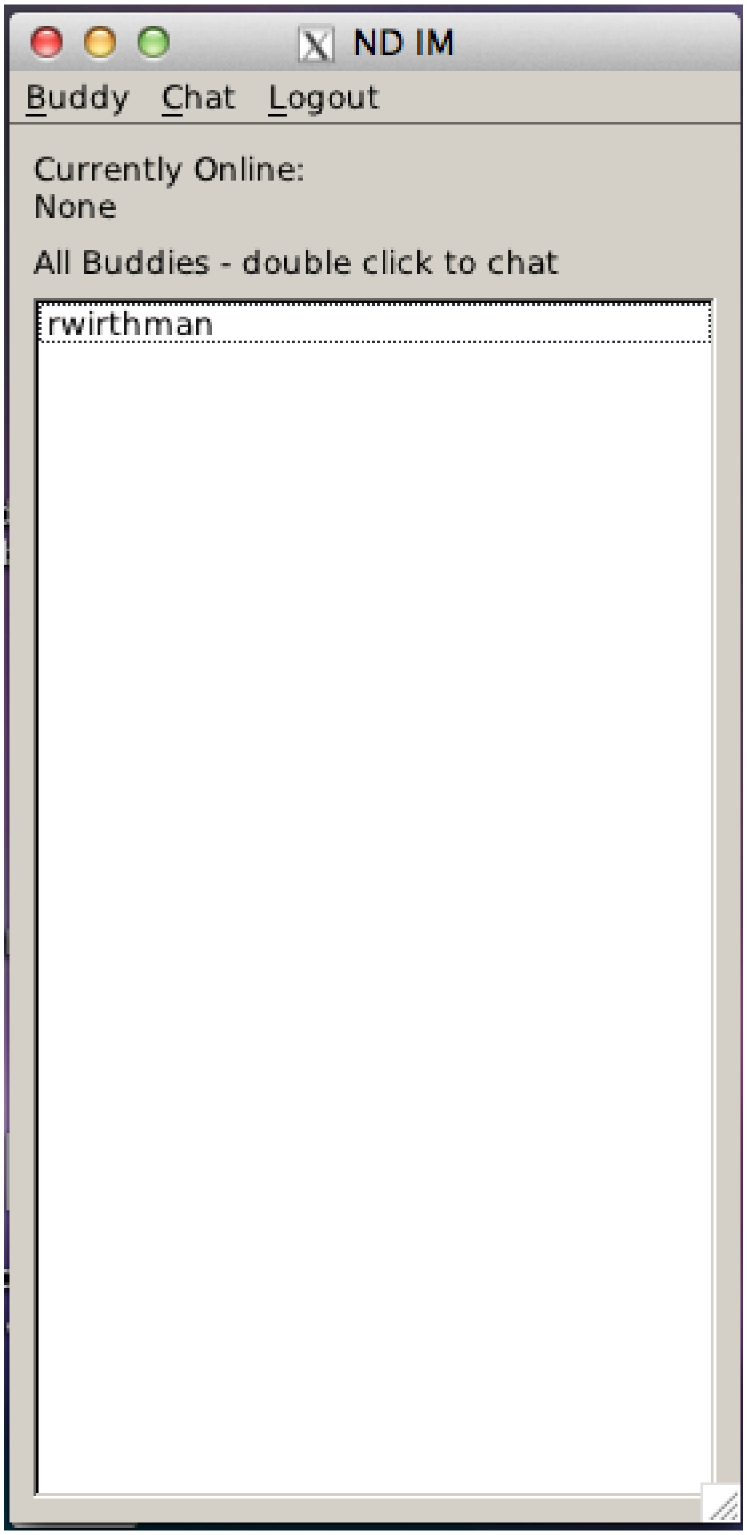
Appendix (Screenshots):

Login Window:



Choose a Password:
Must have 3 of the following.
1)lowercase letters
2)uppercase letters
3)numbers
4)special characters

Main Window:

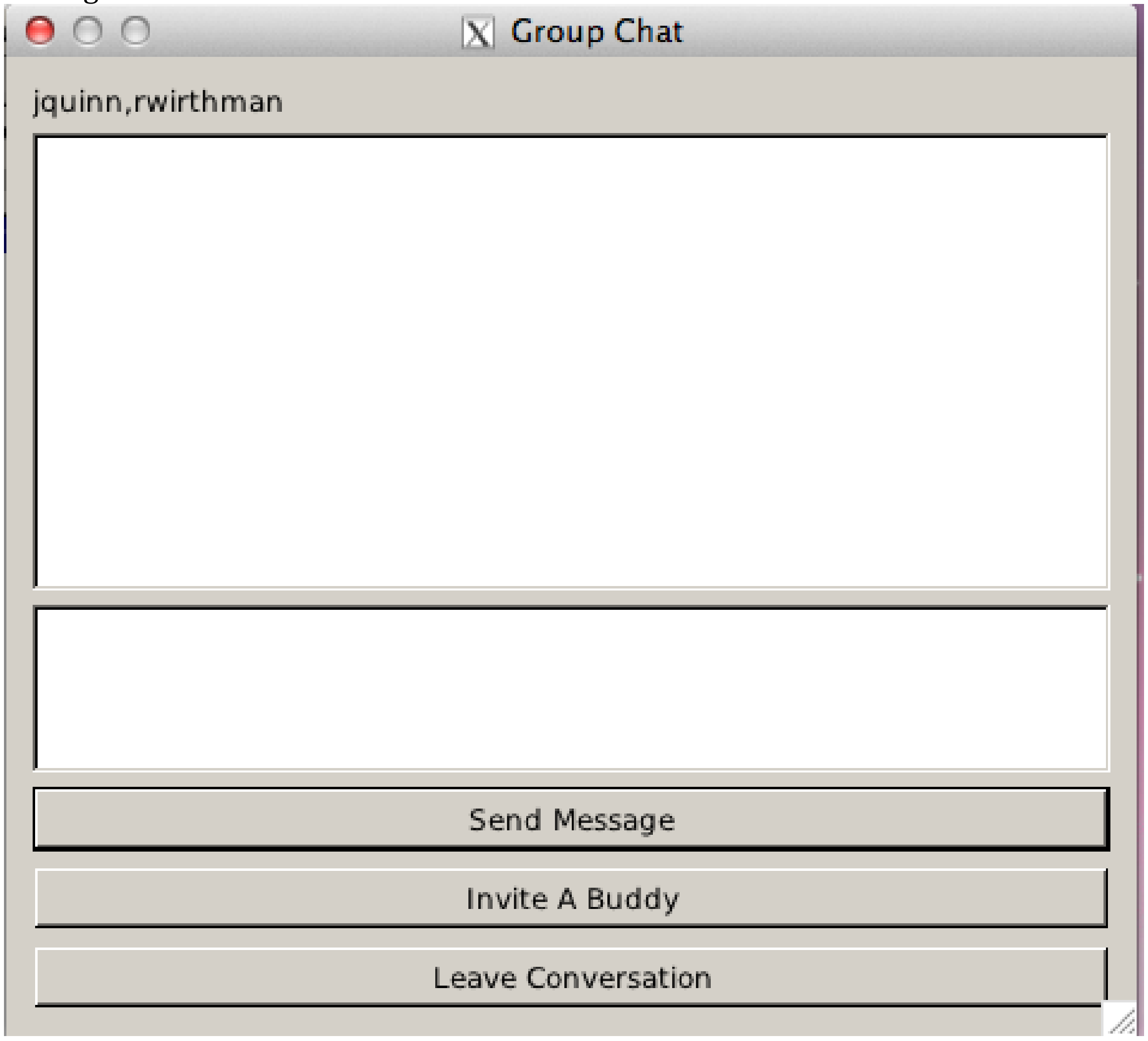


Currently Online:
None

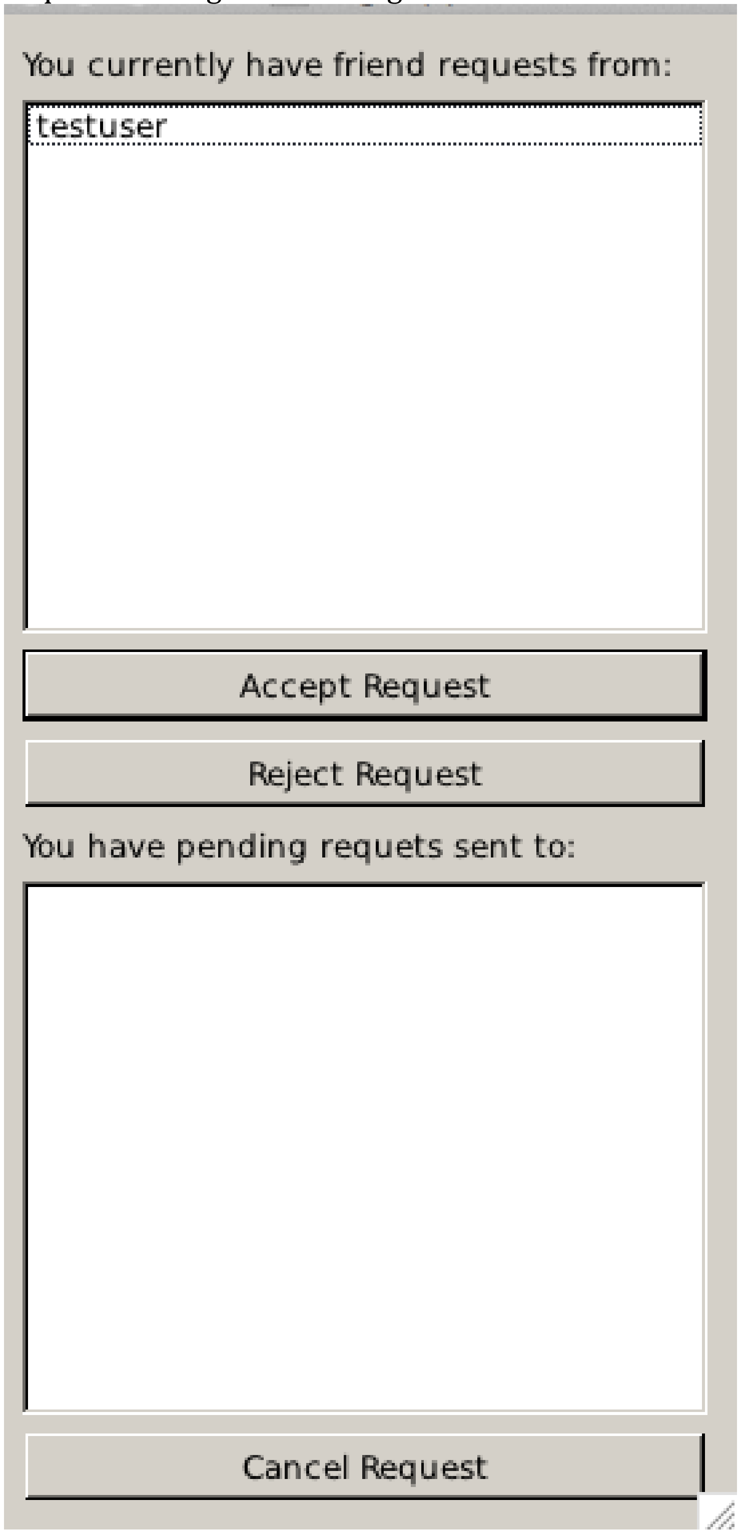
All Buddies - double click to chat

rwirthman

Message Window:



Requests Management Dialog:


You currently have friend requests from:
testuser
Accept Request
Reject Request
You have pending requets sent to:
Cancel Request