James Gieszelmann
Brian Fontana

Final Paper Abstract
For our project we decided to make a mobile message server and client, with the capability to search the collection of messages on the server for any word in a message. To accomplish this, we set up server/client programs that would remotely connect through HTTP Channels and SOAP, and would share objects that were hosted by the server. We decided to give the client a Graphical User Interface created in a Visual Studio C#.NET environment. For the search function, we decided to implement a double hash table, which stored the key, and a list of messages in which the keyword is found.
James Gieszelmann  
Brian Fontana  
Team Autastic: Autastic Remoter

1. Keywords  
- Remoting  
- Message and findMessage Classes  
- Double Hash Table  
- Searching for a key  
- GUI

2. Introduction  
For our project we decided to make a mobile message server and client, with the capability to search the collection of messages on the server for any word in a message. To accomplish this, we set up server/client programs that would remotely connect through HTTP Channels and SOAP, and would share objects that were hosted by the server. We decided to give the client a Graphical User Interface created in a Visual Studio C#.NET environment. For the search function, we decided to implement a double hash table, which stored the key, and a list of messages in which the keyword is found.

3. Body  
To write server/client applications that would interface with each other, we decided to use .NET remoting, a remoting interface included in the System.Runtime.Remoting.dll included in Visual Studio.NET. The very basic idea behind the server and the client and how to connect the two
together using .NET remoting, was taken from http://gsraj.tripod.com/dotnet/remoting.html, a tutorial built and maintained by Gopalan Suresh Raj.

We chose to use HTTP channels over TCP channels because HTTP has a broader application; TCP remoting is primarily used for intranet applications. The way our client/server system works is the server is started, and proceeds to listen on a specified port (in our case, port 1099).

```csharp
    int listeningChannel = 1099;
    // Create a New HTTP Channel that listens on Port 1099
    HttpChannel channel = new HttpChannel(listeningChannel);
    // Register the channel with the runtime
    ChannelServices.RegisterChannel(channel);
    // Expose the Object from this Server to any possible clients
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(Topic),
        "Topic",
        WellKnownObjectMode.Singleton);
```

The following piece of code creates a new HTTP channel that allows the server to listen on port 1099, and then registers that channel with the machine it is on. So, when any other program attempts to use port 1099, the machine knows there is already a program running and listening on that port. The RemotingConfiguration.RegisterWellKnownServiceType allows clients to access the Topic object that is on the server. Making the object Topic available to others is what allows our client
to post messages to the server, and to search for words inside of that message.

It is easy for the client to connect to the server, all the client does is first to specify with which port he is going to connect, then detail where to connect, and then create a reference to the object that the server has registered, as a well known service type.

    //create an instance of the server's serialized object locally
    int listeningChannel = Int32.Parse(CPORTBox.Text);
    HttpChannel channel = new HttpChannel(listeningChannel);
    ChannelServices.RegisterChannel(channel);
    Topic = (Topic) Activator.GetObject(typeof(Topic),"http://" + PORT + ":1099/Topic",channel);
    if (topic == null)
    {
        SVRTTextBox.Text="Unable to obtain a Reference to the Remote Topic Server component ...\n" ;
    }

To be able to store the data efficiently, we created two data storing classes. The first class we created was the Message Class. This simple class is basically an organized data store, which holds the User ID of the person who has posted a message, the Category of the message, the title of the message, and the actual text of the message.

    public class Message : MarshalByRefObject
    {
        public Message()
        {
        }
this.cat = "";
this.tit="";
this.txt="";
this.uid="";
}
public Message(string us, string ti, string tx, string ca)
{
this.cat = ca;
this.tit=ti;
this.txt=tx;
this.uid=us;
}
public string uid;
public string txt;
public string tit;
public string cat;
}
The class findMessage is used in the search algorithm; findMessage is another data container. Because there can be multiple occurrences of one word, we must store an array of messages per each key. The size of the array is dependant upon how many times the key occurs in the given number of messages (including multiple occurrences in a single message).

public class FindMessage
{
public FindMessage()
{
this.key = "";
this.mess = new ArrayList(0);
}
public FindMessage(string keys, Message messa)
{
    this.key = keys;
    this.mess = new ArrayList(1);
    this.mess.Insert(0,messa);
}
public string key;
public ArrayList mess;
}

To store our keys and messages, we developed a double hash table. The way a double hash table works is there is a table with m possible slots. You pass the table a key, and what to store with the key (an array of messages in this case). That key is then passed to a hash function, which turns it into a number between zero and m-1, for indexing purposes. If the slot in the table given by hashing the key is full and not the same as the key, then you pass the key back to another hash function. The return of the second hash function serves as an increment amount that is to be added to the current position; this increment amount is continually added until an open index is found. Once an open index is found, the key along with its message is inserted into the hash table.

For example, if we wanted to insert the key “here” into the double hash table stored on our server, we would pass it to the first hash function.

    public int hf_one(string key)
    {
        int sum = 0;
for (int i = 0; i<key.Length; i++)
{
    sum +=(int)(key[i]);
}
return sum%tbl_size;
}

The hash function turns the string into an integer number, which is then modulated with the size of the table to ensure that the index is a valid index in the table. It then returns the index into which the key is to be inserted into the hash table. Now, if that index is already taken with a different key, then we increment the current index by the value determined by our second hash function.

    public int hf_two(string key)
    {
        int sum = 0;
        for (int i = 0; i<key.Length; i++)
        {
            sum +=(int)(key[i]);
        }
        return sum%((tbl_size/4)-1);
    }

Once the insert function finds an open spot in the hash table, it sets the key from empty to the current key that it is inserting, and then fills the ArrayList of messages with the message in which the key is contained. In the interest of keeping the insert algorithm fast we double the size of the hash table if the table fills up half way and rehash/insert all the values.
When the client wishes to search our hash table, he accesses the find() function of the object that he serialized when he connected to the server, and passes with that function call the key that he is searching. When the server receives that function call, it calls the double hash table’s Find() function. The Find function uses the same hashing procedure as in the Insert. What the find function looks for, though, instead of a blank space, is if the key that it was passed is the same as the key that is stored in the index returned by the hash functions. If it is not the same key, it keeps on looking through the table until it either finds it, or finds a blank space. It then returns the ArrayList of Messages in the current position to the server, where the key is stored, or if the search is for a key that is non-existent, the search returns an empty ArrayList. The client never gets the full ArrayList of returned Messages, but rather requests them one at a time from the server.

The GUI that we created was done in the Microsoft Visual Studio.NET environment, using C#.NET as the language. Controls such as Labels, Textboxes, Listboxes, buttons, and tabs were used. All of the bugs that were encountered during the GUI stage of development were resolved by using the resource http://www.experts-exchange.com as the prime place of research.

4. Summary
The Autastic Remoter was a success. It works very well, is fairly robust, and achieves all of the goals that
were set for it. The server houses a collection of messages as well as a double hash table to search those messages for words/keys. The client is easy to use, and the GUI provides a professional, aesthetic appearance.

For future enhancements, we could add administrative capabilities, with options to ban people or IP addresses from connecting to the server, or limit users access to certain categories. The GUI could be expanded to make it slightly more user friendly, and we could somehow take out the need to specify a port you are trying to connect to, and the IP address as well – this can be put into a preferences tab. We could also modify the search to be able to search for more than just a word in a message. For instance, we could have it search for all the messages that a certain user has posted, or all messages posted on a certain date.
5. Biographies

For Biographies, check out http://www.nd.edu/~jgieszel/bio.html As clearly noted from the biographies (and more so, pictures) these two were unknowingly bound to meet each other, a perfect match, and accomplish great things. The University of Notre Dame provided that meeting place and it was the start of Team Autastic.

6. Appendix

Screen Shots
for Screen shots, see http://www.nd.edu/~jgieszel/screenshots.html
7. Works Cited / References


