Abstract:

This report describes the implementation of short messaging and multicasting amongst ubiquitous hand held devices. The implementation is based on client server methodology, with a central server monitoring all transactions between devices that run the client application. The backbone of the system architecture is the wireless LAN in which these devices are connected through wireless ports. We also develop an interface for these hand held iPAQ devices with the Global Positioning System (GPS) to determine geographical positions of individual devices. The server interacts with the devices keeping a database of each of their parameters. The implementation is in Java for the server end and in embedded C++ for windows CE for the client end. The system provides for a unique position based messaging, where the sender does not require any prior knowledge of the recipient, except its co-ordinates. It is interesting to see that the framework also finds interesting applications, particularly in ad-hoc geographical addressing.

1. Introduction:

The goal of the project may be summarized in the following three points:

1. To create a client-server architecture for communication across mobile devices
2. To provide geographical short messaging
3. To implement efficient communication protocols between client and server for quality of service

The idea comes from the concept of building application of distributed data acquisition networks. Specifically, we require a network of sensors capable of obtaining, processing, and forwarding data to a central and potentially distant location. Such a network can be used to monitor events such as brush fires, hazardous chemical spills, and violent storms. The concept of independent and distributed collection and processing of sensor data in a geographically distributed topology is defined as a Distributed Smart Sensor Network (DSSN).
However, the processing power and efficiency of hand held devices are improving by the day. Hence, we take this idea one step ahead and build more capabilities into the devices through our client application. Besides being data collection nodes, clients also incorporate the ability to do on-site processing. Moreover, they provide for map rendering and interfacing with other devices. In this report we describe each of these functionalities that we built in to the system. We start with describing the basic client server methodology. In this section we discuss the partitioning of tasks between client and server and its implication on the overall performance of the system. It is obvious that the server should bear the brunt of the majority of processing and bookkeeping. The client should be freed from any load that involves multiple task handling or large data processing. Hence, we develop better protocols to minimize traffic across the client and server.

The section on Server implementation talks about the various algorithms used for handling queries from individual clients. It involves concurrent task handling, query processing and socket communication with devices. All these components of the server are optimized for maximum throughput and least perceptible response delays as seen by the client.

Client implementation talks mostly about the user interface for selecting message types. A major part of the client application is the user interface that allows for selecting recipient nodes from the map. The map also needs to be rendered by the client application and needs redrawing every update cycle. Another important feature of the client application is it’s interfacing with the Global Positioning System (henceforth referred to as GPS). For the purpose of our implementation, we chose the TeleType GPS device that interfaces with the COM4 serial IO port of the iPAQ. We also look at the device driver implementation for the same.

The report concludes with experimental results and discussion on possible future work, specifically the potential of this framework in serving as a platform for ad-hoc geographical addressing schemes for mobile devices.

2. Related Work:

The inspiration for this project comes from the existing QUSAR project at the Department of Information and Computer Science, UC Irvine. The QUASAR project tries to build a new paradigm for query processing with mobile devices. In the immediate future, the paradigm of storing data precisely in Database Management Systems (DBMS) and answering queries on them exactly will no longer be valid for a host of new applications. Such applications are emerging in a variety of fields, e.g., data analysis and mining, visualization/virtual reality and distributed systems including great numbers of mobile devices. In many of these areas, exact answering of queries is often not necessary, especially if it is associated with high response times required for accessing large volumes of disk-resident data, or the drainage of power supply in battery-operated devices. Additionally, the experience of the Web has taught us that data is by its nature
imprecise and hence cannot be easily incorporated in DBMS with their precise SQL-type semantics. Their goal is to create a novel database architecture that will allow the seamless integration of imprecise data without sacrificing the quality of answers to queries, and will provide facilities for the exchange of system resources for improvement in quality.

In dynamic spatio-temporal environments where objects may continuously move in space, maintaining consistent information about the location of objects and processing motion-specific queries is a challenging problem. The QUASAR project focuses on indexing and query processing techniques for mobile objects. Specifically, a classification of different types of selection queries that arise in mobile environments was developed and efficient algorithms to evaluate them were explored. Query processing algorithms are developed for both native space and parametric space indexing techniques. A performance study compares the two indexing strategies for different types of queries.

In our project, we extend the idea of ubiquitous data collection devices to build in more functionality into them. So, our approach essentially focuses at user involvement in the device operation through a client application. We also add GPS interface to determine positional information of the device. While exploring the idea of ad-hoc geographical addressing schemes, we came across some prior work done at Dartmouth University. This work also involves geographically distributed sensor and uses similar device addressing scheme as our framework.

In this research, they introduce an approach of addressing each module purely by its acquired GPS position. Each sensor unit uses a GPS receiver to acquire its current position at a regular interval. The unit's position, within some threshold factor, is then used to identify the unit, analogous to an IP address. Sensor units use their newly acquired address as an identifier for routing and communicating among each other. This idea differs from IP addressing in that the address is not fixed, since it will change as the unit's location changes. To demonstrate the value of this concept, consider the scenario where a number of sensors are scattered in the ocean, gathering localized temperature readings. Due to uncontrollable environmental affects, the sensor network topology is volatile. Therefore, information gathered by the sensors should be location dependent, not identity dependent. More information about the project can be found at http://www.cs.dartmouth.edu/CMC/projects/Corr/

3. System Architecture:

Figure 1. shows a top level overview of the system architecture. The client (running on mobile devices) and the server (on a workstation) are communicating through TCP IP sockets. Specifically, the mobile devices run Windows CE and are hooked up to the same LAN as the server through the wireless 802.11 port. Also note that each of the mobile devices has a GPS device interfaced with it. We will discuss this GPS interfacing in detail when we talk about the Client implementation. The primary job of the client is to provide a user interface for the variety of messaging services that are implemented. We will deal with these specific messaging services in greater detail in the section on messaging protocol. For now, it suffices to say that all these services are built
on top of the standard peer to peer SMS messaging service. For the server side, we have to implement a task handler to process incoming queries from various clients.

It should be noted that our implementation keeps the client’s clock and processing cycle times transparent to the server and vice versa. In essence, this is an asynchronous implementation in the sense that server does not know when the client query will arrive. This is typical of all network protocols. Also, we do not impose any hard real time deadlines on query processing or server response time. The implementation is aimed at best effort performance rather than real time performance. However, in certain applications that build upon this framework, it is quite useful to have real time deadlines. The current implementation can be easily extended to incorporate real time features. All we need to do is to parameterize the queries and synchronize the client and server clocks to achieve real time performance. However, for lack of time we did not choose to implement this feature.

4. Messaging Protocols:

In the scenario of our framework, it is important to clearly define messaging protocols to serve basically two major purposes. Firstly, we need to disambiguate messages or queries coming from various clients. As we saw earlier, the server maintains and manipulates client information in a database. Although the clients are ubiquitous and geographically addressed, we need to identify them uniquely in the database and keep log of their activity. So, a unique device Identifier is compulsory with every messaging or update service. We offer the following kinds of messaging services from the client UI:

1. SMS – This is the basic peer-to-peer short messaging service.
2. Multicast – This involves sending same message to more than one recipients
3. Broadcast – The message is sent to all active devices
4. KNN – The message is sent to K nearest neighbors of the sender

Each of these individual messaging services have some commonalities but they differ in the chosen message formatting. As mentioned earlier, all messaging services use a common backbone of SMS messaging. The individual services and their message formats are explained in greater detail in the following sub-sections.

4.1 Client Setup:

Every time a device is initiated into the system, it has to undergo a setup phase before it can start using the system services. The setup phase involves interacting with the server and also with the GPS interface. For the purpose of our current discussion, we will concentrate only of the server interaction.

In this scenario, with a single central server, the client setup is simple to perform. Basically, it does not involve any intelligence on the part of the client to determine dynamically which server it wishes to connect to. The scenario with multiple servers and distributed databases becomes more involved and requires server identification. Moreover, we have not built in any security features into this application yet. This means that the setup phase does not involve any authentication on the part of either the server or the client. For the purpose of our experiment, we used the machine “leapfrog.ics.uci.edu” to host the server application.

The client application determines the host ID from the given name and attempts to open a socket for furthering communication. If the host machine cannot be connected or the server application is not running, the client responds with an appropriate error and exits. However, if the socket is opened successfully, the client initiates its introduction to the network by sending a HI message. The format of the message is as follows:

\[
\text{Device ID} \mid \text{HI} \mid \text{Latitude} \mid \text{Longitude} \mid \text{precision}
\]

Here, \textbf{Device ID} is the identifier of the sender device. It’s important to keep unique device IDs in a given network, for obvious reasons. The server as a primary key for all data processing and management would use this ID. The important thing to note here is that the primary key is generated by the client and not the server. This means that it is the client’s responsibility to ensure the key’s uniqueness. In a more robust environment, the server must be responsible for allocating the device IDs. However, we overcome this shortcoming by keeping a long enough device ID, which is the 12 character long S/N of the device. Since the mobile devices we are using are homogenous and come from the same vendor, it is a fairly safe assumption that the S/N number would be unique.

The message format uses ‘|’ symbol as a delimiter. The “HI” string that identifies the message type follows the device ID. This is followed by the positional information of the sender. Note that the GPS initialization part of the setup phase is done before the device tries to introduce itself into the network. Therefore, we have prior information about the geographical position of the device. The latitude and longitude information is not exact as
per derived from the GPS device. The client to avoid redundant information normalizes the exact GPS data. Since we are operating in the spatial limitations of a local area network, it seems implausible that the position of two devices in the system would ever exceed 1 degree on the co-ordinate space. Therefore, we keep only the “minute” information from the GPS position data and ignore the rest. We will discuss the GPS interaction further in the client implementation.

The precision field of the message format tells the server that the location information would be updated only if the movement of the device exceeds some least count. This least count is imposed by the system depending on the amount of robustness we want from the system. Updating positional information in a periodic fashion on a timely basis is an inefficient implementation that only serves to increase traffic and load on the server. A smarter way is to build in movement checking within the client itself. So, the client maintains the following tuple:

\[
\text{<last_sent_latitude, last_sent_longitude, current_latitude, current_longitude>}
\]

The client checks its movement from last sent position every cycle and whenever this movement exceeds the precision, it must send a position update request to the server with its new normalized co-ordinates.

4.2 SMS messaging:

This service is by far the most important and basic one in our system. The messaging format looks as follows:

\[
\text{Sender_ID | SMS | Recipient_ID | Message_String | time}
\]

As can be seen by the label of the tokens, we need to send information about the Sender ID, so that the server uniquely identifies the devices making the forwarding request. The token “SMS” signifies that the message is meant only for a single recipient. Although, it is the same in principle as a multicast, this is just to notice the difference between the services provided by the client. The “Message String” is, of course, the content of the message to be sent. For practical purposes the length of this string should not be too long. That would increase traffic and consequently the message trip delays.

The idea behind adding the local device time is interesting to note. Although it does not serve us any purpose in the current version of the system implementation, it may come in handy once we extend this system to incorporate real time properties. Moreover, the “time” field helps us in the experimentation with calculating latencies. It should be noted that the server and all the clients are asynchronous to each other. This means that their clocks are not synchronized to each other. Hence, there is no use for the server to know the local time of the client because it has no basis for calculating latency based on the client time and its own local time. However, if a client sends an SMS message to itself, it can notice the local time of the incoming message (from itself) and thereby calculate round-trip delays. Of course, this delay does not account for the query processing time spent by the server. If the server is assumed to be fast enough and service time is negligible compared to data communication latency, we can get a fair estimate of trip delays.
The server reads these messages from the client and translates them to

\[ \text{Server_ID} | \text{SMS} | \text{Recipient_ID} | \text{Message_String} | \text{time} \]

Note that the server has replaced the device ID with its own ID. This message is now sent to the recipient ID. We will see in the server implementation how it retrieves the appropriate socket for the recipient client from its device ID. Also note that the original local time of the sender device is retained. This will be important for finding roundtrip delays as explained above.

4.3 Multicast/Broadcast:

Multicast messaging builds upon the support provided by SMS messaging. The concept is similar but the messaging format differs to avoid introducing redundant information. The format looks like:

\[ \text{Sender_ID} | \text{MCAST} | \text{Number_of_recipients} | \text{Message_String} | \text{recipient_id1|recipient_id2|…} \]

The Sender ID is the device ID of the sender, as usual. “MCAST” token denotes that this message is to be multicast. The new field “Number_of_Recipients”, as indicated by the token name, is the number of devices that should be receiving this message. We also send a list of the entire device IDs for the recipients. Thus we pack a set of individual SMS messages to several devices optimally into one MCAST message. The local time information may have been added to the message, but we did not need it for our current requirements, so we dropped that information to further compact the message.

Broadcast has a simpler message format as follows:

\[ \text{Sender_ID} | \text{BCAST} | \text{Message_String} \]

Here we need only the token BCAST for the server to identify that this message is to be sent to all active devices in the system. The other fields are same as before and have already been discussed.

The sender translates both the broadcast and multicast messages in a similar fashion. It starts with decoding the message and then prepares a list of recipient IDs. Thereafter the message string is sent to all these IDs in the same format as the SMS message. Thus the recipient is transparent to the fact as to whether the message was intended to be SMS, multicast or broadcast. This reduces the work on the client side, since it now has to deal only with one kind of message. So, the server upon receiving the broadcast or multicast message prepares a \(<\text{recipient_list}>\) and executes the following:

\[
\text{For each Recp_id in recipient_list}
\text{Do}
\text{Send} \quad \text{Server_ID | SMS | Sender_ID | Recp_id | Message_String | local_time}
\text{End For}
\]
4.4 K Nearest Neighbors:

A very common requirement in most temporal and spatial query systems is to find out the K nearest neighbors. This requirement has many interesting application specific implications. For instance, a device D1 witnesses an accident and wants to ask for help from the 10 closest devices. All the devices might not be in a position to respond, so the sender multicasts to closest neighbors and follows up with the closest neighbor that responds. This is a classical problem in database querying and is defined in both temporal and spatial frames of reference. In this project, we concentrate only on the spatial K nearest neighbor queries. The temporal queries may be supported but they are much more complicated to implement and beyond the scope of the current version.

The message format looks as follows:

\textit{Sender\_ID | KNN | K | Message\_String}

This format is similar to multicasting, with the difference that we do not provide any recipient Ids. It is the server’s responsibility to find out the K nearest neighbors in the current positional database and redirect the message to the relevant recipients. The generation of new SMS messages and their redirecting takes place in much the same way as we saw for the Multicast & Broadcast cases.

5. Server Implementation

The server performs all the core number crunching and database management activities in the system. The server may be divided into three major functional components viz. the task handler, the database manager and the message manager. The task handler deals with incoming queries, queuing the queries, allotting them execution time and servicing them concurrently. The database manger, implemented by the \texttt{dbase} class, does the bookkeeping for all active devices. Finally, the message manager keeps track of all incoming messages to be forwarded and services them. It also does the job of preparing custom maps for each client. In the following sub-sections, we will look at each of these functional components in greater detail and understand the underlying algorithms and performance improvement techniques.

5.1 The Task manager

The functionality of the task manager is elucidated in Figure 2. It consists of basically two parts. They are the concurrent threads and the FIFO queue that keeps the incoming queries. The queue is implemented in the class ConnectionQueue. For all clients that are connected to the server and are active, we assign a socket for reading in their requests and queries. The Task Manager follows the given algorithm for reading client messages, and placing them on the queue:
**While true**

**Do**

*Socket sock = serverSocket.accept(); // accept socket connection from client*
*ConnectionQueue.put (sock); // place client socket on the queue*

**End while**

As we can see from the above algorithm, the server iterates through all open client sockets. Whenever there is an incoming message at any of the sockets, it is placed in the queue to be processed by one of the task handlers. The task manager also initiates 15 parallel threads of Task Handlers. Each of these task handlers tries to read from the queue. If there is a message waiting to be processed, the task handler reads in that message and starts processing it. The pseudo code is shown below:

**While true**

**Do**

**If connectionQueue not empty**

*Socket sock = (Socket)core.getConnectionQueue().get();*
// get the client socket from queue

**process client request**

**End if**

**End while**

![Figure 2. – Task Handler Implementation](image-url)
The query handling is passed on to the message manager, as we shall see shortly. Note that this implementation uses a purely FIFO queue. Hence, there is no provision for assigning priorities to queries. If there were prioritized queries then we would have incorporated a priority queue or a heap data structure for keeping the incoming queries. In that scenario we would also have to provide for task suspension. Since this is a non-real time implementation, there are situations where an important client request might have to wait for a long time if all the task handlers are busy. Real time processing would require the ability to suspend low priority jobs and reassign task handlers to jobs that are more urgent.

5.2 The database manager

The other important functional component of the server is the database manager. This is a misnomer of sorts, since the device information is maintained in dynamic data structures, and not in file. As seen in the Figure 3 the database management keeps track of individual device parameters, like position, movement and neighbors. Specifically, the device information carries the following:

1. Device ID: this is the primary key for retrieving any device information
2. Device co-ordinates: these are the normalized latitude and longitude data, representing the current position
3. Last movement: the time of last movement
4. Neighbors list: this is the list of all active neighbors for device, sorted by distance

![Figure 3. – Device database organization](image-url)
For keeping track of all active devices we provided methods for adding and removing devices. Server, as a response to clients “HI” message, allocates new data structure for keeping clients information. If capacity of server is not exceeded, server reads data sent by the client in initiation stage. It creates data structure for the device, initializes its attributes, updates lists of neighbors for all active devices, adding new device into it. Also, it adds device to list of active devices. We implemented this as following:

\[ \text{AddDeviceToDbase}(device \ d) \]

\[ \text{If } \text{NumberOfDevices} \text{ not greater than Database capacity} \]
\[ \quad \text{And Device is not already in Database} \]
\[ \quad \text{Do} \]
\[ \quad \quad \text{deviceList.put}(d.\text{GetDeviceId}(), \ d); \quad \text{//creates and initializes device entry} \]
\[ \quad \quad \text{For all active Devices} \]
\[ \quad \quad \quad x.\text{neighbors.add}(x); \quad \text{// adds current device to neighbors list} \]
\[ \quad \quad \text{End for} \]
\[ \quad \quad \text{// update number of devices in the database and device Id list} \]
\[ \text{End if} \]

Removing device implements inverse procedure. Server removes entry for the device, and removes it from the list of active devices. For all active devices, it updates list of neighbors, removing device from it.

\[ \text{RemoveDeviceFromDbase}(device \ d) \]

\[ \text{If } \text{deviceList} \text{ is not empty} \]
\[ \quad \text{Do} \]
\[ \quad \quad \text{deviceList.remove}(d.\text{GetDeviceId}()); \]
\[ \quad \quad \text{// removes corresponding data structure for device d} \]
\[ \quad \quad \text{For all active Devices} \]
\[ \quad \quad \quad x.\text{neighbors}.\text{remove}(x); \quad \text{// removes current device to neighbors list} \]
\[ \quad \quad \text{End for} \]
\[ \quad \quad \text{// update number of devices in the database and device Id list} \]
\[ \text{End if} \]

An important feature of the database management is to keep the list of neighbors sorted by distance for each the devices. This is important since we have to answer KNN queries and if we maintain a sorted list of neighbors, it may be done efficiently. All that we need to do is to send the SMS message to the first K devices in the neighbor list. Also note that this neighbor list is an individual copy for each of the client. For coherence purposes, this list needs to be resorted and modified every time we notice movement of any device. Once a device moves a distance beyond its set precision, it sends an update position to the server. The server responds by updating its position and follows up with updating the sorted list of all neighbors of all devices. This may be a computationally intensive task if the system is volatile, which means that the devices move around very often thereby
flooding the server with update position requests. Assuming that there are N devices in the system. The movement of 1 device may cause re-sorting of all lists. If this re-sorting is done efficiently, it can be performed in O (N) time for each of the N lists. The O (N) complexity comes from the fact that only one item needs to be placed in the correct position. Therefore the total complexity for bringing the server database to coherent state is O (N^2). If we have several devices and frequent movements, this is a big bottleneck in server performance.

5.3 The message manager

The message manager primarily takes care of incoming requests and handles them on a need by need basis. The client is first introduced into the system by the HI message, as we have seen earlier. On receiving the HI message, the server opens a new socket for the new client connection. The registry phase of the set up process involves putting the client socket address in the hash table indexed by the device ID. Once the device is registered, the server would reopen this socket for reading or writing every time there is a new message from this client.

The message manager handles other messages from the clients by either forwarding them to appropriate recipients or answering the queries therein. If the message is an SMS type, it is forwarded to the single recipient by adding the server ID field to the message. For Multicast and Broadcast messages, the server generates several new messages each of the aforementioned SMS type. For multicast case, the server retrieves sockets of recipient clients from the registry hash table and forwards them as individual SMS. For broadcast messages, the server with the same SMS message writes all the sockets in the registry.

In the case of KNN messages, the server has additional job of finding the appropriate device IDs from the database. Therefore it sends a request to the database class to reply with the ID list of the K nearest neighbors of the sender device. Once these IDs are obtained, the following job of retrieving them from the registry and forwarding the message is done just as in the case of Multicasting.

Another responsibility of the message handler is to generate custom Map for each of the client devices. When a client requires the latest copy of the map, it sends a request to the server. It would be inefficient for the server to send the locations of all the devices in the database since this would mean too much redundant traffic overhead. Therefore the server only sends the positions of those clients whose positions have changed since the last map request was made by the sender device. This ensures that only the required update is performed for the client.

6. The Client Implementation

The client implementation consists of mainly two parts viz. User and GPS interface. All of client implementation is done with embedded visual C++ on windows
CE. Before the set up phase on the client side, the device needs to start interacting with the GPS device. We shall discuss both these interfaces in detail in the following sections.

6.1 GPS Interfacing

TeleType develops the GPS device used for our experiments and they provide a tool along with the device to develop application and gather GPS data etc. The TeleType application is primarily used for transportation control and navigation. The application allows for reading in GPS receiver data and logging it to a file. There are certain NMEA (The National Marine Electronics Association) standards that are observed while formatting GPS data. NMEA standard defines an electrical interface and data protocol for communications between marine instrumentation. The simple sentence standards are enumerated below:

19 Interpreted sentences

$GPBOD - Bearing, origin to destination
$GPBWC - Bearing and distance to waypoint, great circle
$GPGGA - Global Positioning System Fix Data
$GPGLL - Geographic position, latitude / longitude
$GPGSA - GPS DOP and active satellites
$GP GSV - GPS Satellites in view
$GPHDT - Heading, True
$GPR00 - List of waypoints in currently active route
$GPRMA - Recommended minimum specific Loran-C data
$GPRMB - Recommended minimum navigation info
$GPRMC - Recommended minimum specific GPS/Transit data
$GPRTE - Routes
$GPTRF - Transit Fix Data
$GPSTN - Multiple Data ID
$GP VBW - Dual Ground / Water Speed
$GP VTG - Track made good and ground speed
$GPWPL - Waypoint location
$GPXTE - Cross-track error, Measured
$GPZDA - Date & Time

Out of this sentence, there are only three that are of particular interest to us. $GPGGA gives us the position fix data that is an indication of when the device fixes position. GPGSV tells us about satellites in view. Most important however is the positional data provided by the $GPRMC message. Each of the relevant message codes are discussed in detail here:

$GPGGA

Global Positioning System Fix Data
$GPGGA,hhmmss.ss,llll.ll,a,yyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx*hh

hhmmss.ss = UTC of position
llll.ll = latitude of position
a = N or S
yyyy.yy = Longitude of position
a = E or W
x = GPS Quality indicator (0=no fix, 1=GPS fix, 2=Dif. GPS fix)
xx = number of satellites in use
x.x = horizontal dilution of precision
x.x = Antenna altitude above mean-sea-level
M = units of antenna altitude, meters
x.x = Geoidal separation
M = units of geoidal separation, meters
x.x = Age of Differential GPS data (seconds)
xxxx = Differential reference station ID

$GPGGA,hhmmss.ss,llll.ll,a,yyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx*hh

1 = UTC of Position
2 = Latitude
3 = N or S
4 = Longitude
5 = E or W
6 = GPS quality indicator (0=invalid; 1=GPS fix; 2=Dif. GPS fix)
7 = Number of satellites in use [not those in view]
8 = Horizontal dilution of position
9 = Antenna altitude above/below mean sea level (geoid)
10 = Meters (Antenna height unit)
11 = Geoidal separation (Diff. between WGS-84 earth ellipsoid and mean sea level. -=geoid is below WGS-84 ellipsoid)
12 = Meters (Units of geoidal separation)
13 = Age in seconds since last update from diff. reference station
14 = Diff. reference station ID#
15 = Checksum

$GPGSV

GPS Satellites in view
$GPGSV, 3, 1, 11, 03, 03, 111, 00, 04, 15, 270, 00, 06, 01, 010, 00, 13, 06, 292, 00*74
$GPGSV, 3, 2, 11, 14, 25, 170, 00, 16, 57, 208, 39, 18, 67, 296, 40, 19, 40, 246, 00*74
$GPGSV, 3, 3, 11, 22, 42, 067, 42, 24, 14, 311, 43, 27, 05, 244, 00, , , , , *4D
$GPGSV, 1, 1, 13, 02, 02, 213, , 03, -3, 000, , 11, 00, 121, , 14, 13, 172, 05*62

1 = Total number of messages of this type in this cycle
2 = Message number
3 = Total number of SVs in view
4 = SV PRN number
5 = Elevation in degrees, 90 maximum
6 = Azimuth, degrees from true north, 000 to 359
7 = SNR, 00-99 dB (null when not tracking)
8-11 = Information about second SV, same as field 4-7
12-15 = Information about third SV, same as field 4-7
16-19 = Information about fourth SV, same as field 4-7

$GPRMC

Recommended minimum specific GPS/Transit data

$GPRMC, 081836, A, 3751.65, S, 14507.36, E, 000.0, 360.0, 130998, 011.3, E*62

$GPRMC, 225446, A, 4916.45, N, 12311.12, W, 000.5, 054.7, 191194, 020.3, E*68

225446 Time of fix 22:54:46 UTC
A Navigation receiver warning A = OK, V = warning
4916.45, N Latitude 49 deg. 16.45 min North
12311.12, W Longitude 123 deg. 11.12 min West
000.5 Speed over ground, Knots
054.7 Course Made Good, True
191194 Date of fix 19 November 1994
020.3, E Magnetic variation 20.3 deg East
*68 mandatory checksum

$GPRMC, 220516, A, 5133.82, N, 00042.24, W, 173.8, 231.8, 130694, 004.2, W*70

1 2 3 4 5 6 7 8 9 10 11 12
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>220516</td>
<td>Time Stamp</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>validity - A-ok, V-invalid</td>
</tr>
<tr>
<td>3</td>
<td>5133.82</td>
<td>current Latitude</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
<td>North/South</td>
</tr>
<tr>
<td>5</td>
<td>00042.24</td>
<td>current Longitude</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>East/West</td>
</tr>
<tr>
<td>7</td>
<td>173.8</td>
<td>Speed in knots</td>
</tr>
<tr>
<td>8</td>
<td>231.8</td>
<td>True course</td>
</tr>
<tr>
<td>9</td>
<td>130694</td>
<td>Date Stamp</td>
</tr>
<tr>
<td>10</td>
<td>004.2</td>
<td>Variation</td>
</tr>
<tr>
<td>11</td>
<td>W</td>
<td>East/West</td>
</tr>
<tr>
<td>12</td>
<td>*70</td>
<td>checksum</td>
</tr>
</tbody>
</table>

```$GPRMC,hhmmss.ss,A,llll.ll,a,yyyy.yy,a,x.x,x.x,ddmmyy,x.x,a*hh```

1  = UTC of position fix  
2  = Data status (V=navigation receiver warning)  
3  = Latitude of fix  
4  = N or S  
5  = Longitude of fix  
6  = E or W  
7  = Speed over ground in knots  
8  = Track made good in degrees True  
9  = UT date  
10  = Magnetic variation degrees (Easterly var. subtracts from true course)  
11  = E or W  
12  = Checksum

The client application is responsible for gathering GPS data at regular intervals. Our original design was to run the TeleType application in parallel. TeleType provides for Serial IO port interface and logs GPS data into a file. This file may be used by the client in a producer – consumer mode and decrypt the GPS message codes. However, there are several problems with this kind of setup. Firstly, the file needs to be closed every time the client wants to read it. This is because the Teletype application is also writing to it and hence it is a critical section. This has to be done manually and defeats our objective of collecting and sending GPS data to server without manual interference. Moreover, file IO is very slow and the TeleType application is also a resource hogger. To avoid these complications, we decided to write a device driver to read data directly from the COM4 port that the GPS device writes to.
Thereafter, the client normalizes this data and updates its positional information. The raw GPS data looks as follows:

$GPRMC,225446,8,4916.45,N,12311.12,W,000.5,054.7,191194,020.3,E*68$

This is decrypted to

Latitude 49 degrees 16.45 mins. NORTH
Longitude 123 degrees 11.12 mins. WEST

As we discussed earlier, the system constrains us to operate only in a LAN area. Thus the degree information is irrelevant to us. We therefore normalize this co-ordinate information to carry only the minute information. The precision is kept at a minimum of 0.01 minutes, so if the position changes the client sends an update position request to the server in the following format:

Device ID | POS | Latitude | Longitude | time

6.2 User Interface

The user interface is made with Microsoft foundation classes. The implementation details are not relevant to be talked about here, although this effort was the most time consuming part of the project. The User interface provides several views to the user to access various messaging services provided by the client application. One of the views is used to render the map of all devices in the system. As mentioned earlier, the server generates an optimal customized map for the client. The co-ordinates of devices on the map are normalized further at the client end so that the map shows the relative positions of other devices. The code for the GUI implementation is attached in the appendix section of this report.

7. Future work: Geographical Ad-hoc Addressing and Routing

In the near future, Global Positioning System (GPS) cards will be deployed in each car and possibly in every user terminal. A user's location will become information that is as common as the date is today, getting input from GPS, when outdoors, and other location providing devices, when indoors. Availability of location information will have a broad impact on application-level as well as on network-level software. Possible new services and functionalities include geographic messaging, advertising, and resource discovery. Geographic messaging constitutes the ability to send a message selectively only to specific sub areas defined by latitude and longitude. For example, sending an emergency message to everyone who is currently in a specific area, such as a building, train station, or a highway. Once given the ability to send a message to a distinct geographical area, it would then be possible to perform geographically targeted advertising over the Internet. For instance, a business may wish to advertise a given service only to clients who are within a certain geographic range (say within 2 miles). Conversely, users could use geographic messaging to discover services or resources
within a geographical region, such as in the direct proximity of the user. One can imagine a "who is around" service which finds out who is currently present in a specific geographic area. If we assume that terminals are also equipped with a camera, a user may point his terminal in a specific direction and obtain annotation (links) to the objects displayed by a camera viewer. In this way the whole external world can become one large "web page". Thus, a building may have a link explaining its function. Links can also be attached to mobile objects appearing on the camera viewer.

To support such applications, location should become a first class citizen in networking protocols, such as IP or ATM, and in the application layer. Routing protocols for geographic messages should be developed that would allow routing to a specific area defined by a polygon of geographic coordinates. Location should also be a parameter in World Wide Web (WWW) access protocols to provide pages located on servers within a specific distance from the user. *Distance-based web bookmarks* can be used to restrict the relevance of web pages by using distance as additional relevance criterion when accessing material on the web.

We address the question of protocol support for location aware services such as geographic messaging, geographic service discovery, and geographical service advertising. Geographic routing is a key support that is necessary. Thus, the exact routing mechanisms to make it happen are discussed in this paper.

We will first discuss **geocasting** - broadcasting to geographical areas defined as arbitrary polygons. Next, we will follow with a brief discussion of **geo-multicasting**, multicasting to geographic areas.

**Addressing Model**

Two-dimensional geographic positioning offers latitude and longitude information as a two dimensional vector:

\[ \text{latitude, longitude} \]

where longitude ranges from -180 (west) to 180 (east), and latitude ranges from -90 (south) to 90 (north). Thus \(< 40.48640, -74.44513 \) is an example of the geographic coordinates for the town of New Brunswick, New Jersey, U.S.A.

Assuming the use of single precision floating-point numbers, four bytes of addressing space are necessary to store latitude and four bytes are also sufficient to store longitude. Thus a total of eight bytes are necessary to address the whole surface of the earth with precision down to 0.1 mile!

A destination geographic address would be represented by some closed polygon such as:

- point
- circle (center point, radius)
- polygon

where each vertex of the polygon is represented using geographic coordinates. This notation would be used to send a message to anyone within the specified geographical area defined by the closed polygon.

For example, if we were to send a message to city hall in Fresno, California, we could send it by specifying the geographic limits of the city hall as a series of connected lines that form a closed polygon surrounding it. Therefore the address of the city hall in Fresno could look like:
In a hypothetical usage scenario, a user will interact with a zoom-able map with a graphical user interface. The address of the message will be specified as a polygon on the map. Then, the polygon will be translated into geographic coordinates and the message will be sent to all clients who are located within the bounds of that polygon.

Routing Geographically

Let us now describe the three suggested solutions for delivering a message to any geographical destination. The solutions described are the Geographic Routing Method, the Geographic-Multicast Routing Method, and the Domain Name Service Method. The solutions were chosen so that the necessary geographic routing infrastructure in the Internet varies from very little (Domain Name Service Method) to medium (Geographic-Multicast Method), to significant (Geographic Routing Method). Currently, we are in the process of evaluating a prototype implementation of the Geographic Routing Method and setting up an experimental network capable of routing the geographic messages. This is a DARPA-sponsored Integrated Technology Demonstration (ITD) within the GloMo (Global Mobile Information Systems) program.

All of these methods assume that a user is able to determine his location in some way. While outdoors, the user can make use of the Global Positioning System to determine his location. When indoors, a different method needs to be used. One possible solution is to have each room contain a radio beacon placed on the ceiling. Each radio beacon will have its own geographic address associated with it which it will broadcast. The geographic address of the mobile hosts will be set to be the same as the beacon's. In this manner, the mobile user can have a geographic address associated with him even though he is indoors and his GPS module is useless.

Geographic Routing Method

The Geographic Routing Method (GEO) uses the polygonal geographic destination information in the geographic message header directly for routing. Geographic routing is going to be implemented in the Internet Protocol (IP) Network layer and the Application layer in a manner similar to the way multicast routing was first implemented. That is, a virtual network which uses geographic addresses for routing will be overlayed onto the current IP internetwork. We would accomplish this by creating our own geographic address routers. These routers would use IP tunnels to transport data packets through areas which do not support geographic routing.

The system is composed of three main components: GeoHosts, GeoNodes, and GeoRouters.

The GeoHost is located on all computer hosts which are capable of receiving and sending geographic messages. Its role is to notify all client processes about the availability of geographic messages, the host computer's current geographic location, and the address of the local GeoNode.

A GeoNode is an entry/exit point for the routing system. The main function of the GeoNode is to store incoming geographic messages for the duration of their lifetimes and to periodically multicast them on all of the subnets or wireless cells to which it is attached. Each subnet and each wireless cell will have at most one GeoNode. The lifetime
of a geographic message is specified by the sender of the message. Message lifetimes are necessary because the receivers of geographic messages may be mobile and may possibly arrive at the message destination just after the geographic message first arrives.

Since, most likely, there will be several geographic messages residing in a GeoNode at one time, the multicasting of the various messages will be scheduled. The scheduling algorithm will take into account the size of the message, the priority of the message, and the speed of the subnet's transport medium. Clients wishing to receive geographic messages would then tune in to the appropriate multicast group to receive them.

Geographic routers (GeoRouter) are in charge of moving a geographic message from a sender to a receiver. GeoRouters are essentially routers which are geographically aware. Each router is charged with performing geographic routing functions for a small number of GeoNodes whose geographic areas of operation are contiguous. GeoRouters keep track of the geographic area that they service (called its service area) by calculating the union of the geographic areas covered by its GeoNodes. Its service area is represented as a single simple closed polygon whose vertices are denoted by geographic coordinates. GeoRouters build their routing tables by exchanging service area polygons. For scalability reasons, in order to reduce the size of the routing tables, GeoRouters are arranged in a hierarchical fashion with each layer corresponding to a distinct geographic area, such as a state or a city.

Sending a geographic message involves three steps: sending the message, shuttling the message between routers, and receiving the message. In order to send a geographic message, the programmer would use the Geographic Library routine SendToGeo(). The function will first contact the local GeoHost Daemon and query it for the IP address of the local GeoNode. It will then send the message directly to the GeoNode which, in turn, will simply forward the message to the local GeoRouter.

Once it receives a geographic message, a GeoRouter must first determine if it services any part of the area of the destination polygon. To do this, the router determines if the destination polygon and the router's service area polygon intersect each other. If not, then the router simply sends the message to its parent router. However, if the polygons intersect or contain each another, then the router does service the area described by the intersection polygon.

The router keeps a cache of the next-hop destinations of the most recent geographic message packets. When a router receives a geographic message packet, it will use the incoming packet's Message Id as a key into the cache. If this is not the first packet to arrive for this destination and if the timer on the cache entry has not yet expired, then the cache will return a list of all of the next hop addresses to which copies of the packet must be sent.

Once a geographic message has been sent to a GeoNode from a geographic router, the receive process can begin. The GeoNode will store the message locally and assign a multicast group to it. Periodically it will multicast the message to its multicast group and advertise its presence using ICMP messages on a well-known channel. The GeoHost
daemons will receive the ICMP messages and determine if the host computer is located inside the message's destination polygon. When a client process executes a `RecvFromGeo()` call from the Geographic Library, the function will join the appropriate multicast address and receive the geographic message itself.

**Domain Name Server Solution**

In this subsection we sketch a solution which relies heavily on the Domain Name Service (DNS). Here, the geographic information is added to the DNS servers. These will provide the full directory information down to the level of the IP address of each GeoNode and its area of coverage.

A new first level domain - `.geo` is added to the set of first level domains. The second level domain names represent states, the third counties, and, finally, the fourth polygons of geographic coordinates. We can also allow polygons to occur as elements of second or third-level domains to enable the sending of messages to larger areas. Thus a typical geographic address can look like `city-hall-Palo-Alto.San-Mateo-County.California.geo` or `Polygon.San-Mateo-County.California.geo` where Polygon is a sequence of coordinates. This geographic address is resolved into a set of IP addresses of the GeoNodes which cover that geographic area. Depending on the size of the message, it may now be transported to the GeoNodes in one of two ways. If the message size is small, then it will be sent as a set of unicast messages to all of the GeoNodes corresponding to the addresses returned by the DNS. Alternatively, given a large message size, it is more efficient to first ask all of the GeoNodes to join a temporary multicast group for the geographic area specified in the message. The message content is then sent to that multicast group.

**Geographic Email**

The Geographic Email (GeoMail) application demonstrates the use of geographic routing. It allows a user to send a text message to any geographic destination. The GeoMail program also displays all available geographic messages in relation to the user's current position. When the user's position intersects the destination polygon of a geographic message, then the GeoMail program will receive the message and display its contents.

**Geo-multicasting**

While geocasting is an important service, it is more likely that we will multicast rather than broadcast into the geographical areas. For example, we will be interested in reaching all *motorists* on a specific highway, or all *police cars*, rather than reaching *everybody*. Geographically directed multicast will accomplish this. Both geocasting methods described before can be modified to accommodate geo-multicasting. The hierarchy of geo-routers can be used also to maintain information about multicast group memberships. Our multicasting solution can also be easily extended to handle arbitrary multicast groups in conjunction with partitions and atoms. Finally, other solutions are also possible, based on a concept of *area codes* analogous to the ones used in telephony today. Geo-multicasting will be described in more details in a forthcoming paper.

**Geographic-Based Service Querying and Advertising**

21
The future mobile users will need information that is pertinent to their location. Such information could include maps of the locale area, traffic and tourist information, as well as what restaurants or other establishments are available locally. With the rapid growth of the volume and diversity of data, the user of the future will find it increasingly difficult to discover or know in advance the correct servers to go to in order to obtain the location-dependent information he seeks. As businesses become directly connected to the Internet, the sought-after information sources will be collocated with the business that is providing it. In such a future, a geographic routing enabled Internet would allow for the existence of services based on geography or distance.

Geographic-based services would entail distributing or finding information within a geographical area. Note that since we assume that the information servers will be collocated with the individual or business that provides the information, we assume that the information contained in those geographically-close servers will contain information that is of greater relevance to the user's current location. Therefore, geographic-based services optimize the relevance of the information gathered rather than network resources used in order to obtain that information since the geographic proximity of the information does not imply that the information servers are nearby in terms of the network topology.

Such geographic-based services would include advertising to a specific geographical region such as only within a certain distance from the server. A restaurant or store could advertise itself or its sales to travelers on nearby streets and roadways. Clients could also request services in a specific geographic region such as only within a certain distance from their current location. For example, a user could request local tourist information such as local maps or directions to monuments and buildings.

8. Conclusions and Acknowledgements

In this report we have described our implementation of geographic messaging and multicasting. A lot of interesting opportunities lie ahead as seen in the future work section. The basic shortcoming of our implementation is the spatial constraint. Since we restricted ourselves to a single central server, the utility of the system was limited. However, this is a good starting point for a more extensive work. The spatial coverage of the system is easily extensible by adding hierarchy to the system. Instead of having a central server, we may have several distributed servers and a hierarchy of servers. This poses problems like coherence. Also, since the bandwidth of wireless communication is limited, the large size of the maps and prolific devices would add to the challenges to quality of service. As mobile devices get more and more processing power, we may transfer some of the functionality to the client end thereby leaving the server with mostly bookkeeping work to do.

The current implementation does not bother about security issues. It would be useful to incorporate security features into the system like message blocking, anonymity etc. Also, there is a plethora of application level functionality that may be added. The ability to service temporal queries is one such desirable feature.

In conclusion, we implemented Client Server architecture for “Instant” messaging across mobile devices. We built provision for sending messages with only positional knowledge.
of the recipient. And finally we built a system that may be used as a framework for building several geographical knowledge based applications. We would like to thank Prof. Rajesh Gupta for his insightful ideas and the proposition of this project. We would also like to thank the QUASAR team at UC Irvine for their help with introducing us to the technical details of their implementation.

Appendix – A

Here we present relevant parts of the server code. The implementation is done in Java. More information and actual source code may be obtained by sending a mail to either of the authors.

A.1
This section of the appendix lists the task handler part of the server. The important classes shown here are those for the FIFO queue, the individual task handlers and the system task manager.

A.1.1 The FIFO queue class

```java
import java.util.*;
import com.sun.java.util.collections.ArrayList;

public class ConnectionQueue {
    private ArrayList connections;
    private boolean available = false;

    public ConnectionQueue() {
        connections = new ArrayList();
    }

    public synchronized Object get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { } 
        }
        Object retval = connections.remove(0);
        if (connections.size()==0)
            available = false;
        notifyAll();
        return retval;
    }

    public synchronized void put(Object conn) {
        connections.add(conn);
        available = true;
        notifyAll();
    }
}
```
A.1.2 Task handler code:

```java
public class TaskHandler extends Thread {

    private String id;
    private Core core;
    private BufferedReader in;
    private PrintWriter out;

    public void run() {
        System.err.println("TaskHandler " + id + " is running");
        while (true) {
            Socket sock = (Socket) core.getConnectionQueue().get();
            try {
                in = new BufferedReader(new InputStreamReader(sock.getInputStream(), "UnicodeLittle"));
            }
            catch (IOException e) {
                System.err.println("Could't get BufferedReader");
                e.printStackTrace();
                try {
                    out.close(); sock.close();
                }
                catch (Exception e1) { }
                continue;
            }
            String line="";
            try {
                line = in.readLine();
                System.err.println(""sends
 &&	" + line + "");
            }
            catch (IOException e) {
                System.err.println("Trouble Reading Message");
                e.printStackTrace();
                try { out.close(); sock.close(); } catch (Exception e2) { } continue;
            }

            out = getPrintWriter(sock);
            if (out==null) {
                try {
                    out.close(); sock.close();
                }
                catch (Exception e) { }
                continue;
            }
        }
    }
```

```java
StringTokenizer st = new StringTokenizer(line, "|");
String firstToken = st.nextToken();

if (firstToken.equals("CLIENT_CONNECT")) {
    if (!sock.getInetAddress().equals(sock.getLocalAddress())) {
        out.println("You are not allowed to connect to this server"); out.flush();

        System.err.println(id + " : Unauthorized CLIENT_CONNECT received from " + sock.getInetAddress());

        try {
            out.close(); sock.close();
        } catch (Exception e) {
        }
        continue;
    }
    String assigned = core.getAndIncrementClientNumber();
    out.println(assigned);
    out.println(core.getId());
    out.flush();
    core.registerClient(assigned, sock.getInetAddress(), out);
    try {
        out.close(); sock.close();
    } catch (Exception e) {
    }
    continue;
}
String senderID = firstToken;
String msgType = st.nextToken();

// Add code for handling different message types here
// example shown for handling SMS type message

if (msgType.toUpperCase().equals("SMS")) {
    String destID = st.nextToken();
    String msg = "|" + core.getId() + "|SMS";
    while (st.hasMoreTokens())
        msg += "|" + st.nextToken();
    if (!core.db.isActive(destID))
    {
        System.err.println("Inactive recipient device : Id = " + destID);
        continue;
    }
    InetAddress destAddress = core.getSourceAddress(destID);
    PrintWriter destOut = getPrintWriter(core.getSourceSocket(destID));
    try {
```
destOut.println(msg);
destOut.flush();
destOut.close();
}
catch (Exception e)
{
    System.err.println("Sending SMS from"+senderID+" to "+destID+" failed");
e.printStackTrace();
    continue;
}
System.err.println("Message "+msg+" from "+senderID+" was sent to "+destID);
}

A.1.3 Top level task manager

class Core extends Thread {
    private TaskHandler taskHandler[];
    public void run() {
        // instantiates the FIFO request queue
        connectionQueue = new ConnectionQueue();
        // initiates the task handler threads
        for (int i=0; i<nThreads; ++i)
            taskHandler[i].start();
        try
        {
            serverSocket = new ServerSocket(portNumber);
        }
        catch (IOException e)
        {
            System.err.println("Trouble initializing server socket on port "+portNumber);
            e.printStackTrace();
            System.exit(1);
        }
        System.err.println("Server Core "+getId()+" is up");
        System.err.println("Current Source Version is "+currentSourceVersion);

        while (true)
        {
            try
            {
                // waits for incoming requests
                Socket sock = serverSocket.accept();
                // puts incoming request on FIFO queue
                connectionQueue.put(sock);
            }
            catch (IOException e)
            {
                System.err.println("Trouble accepting connection");
            }
        }
    }
}
A.2

This section of the appendix lists the database handling part of the server. The relevant classes and their primary methods are enlisted here.

A.2.1 The dbase class for maintaining device information

```java
import java.util.*;
import com.sun.java.util.collections.LinkedList;

public class dbase {
    private int NumberOfDevices;
    private String d_ids[];
    private Hashtable deviceList;
    private static Hashtable knn;
    private Hashtable mapTable, LocalMapTable;

    public void dbase()
    {
        NumberOfDevices = 0;
        deviceList = new Hashtable();
        mapTable = new Hashtable();
        LocalMapTable = new Hashtable();
        knn = new Hashtable();
    }

    public boolean AddDeviceToDbase(device d)
    {
        Position p;
        if(NumberOfDevices > 99)
        {
            System.err.println("Exceeded database capacity");
            return false;
        }
        if(deviceList.contains(d.GetDeviceId()))
        {
            System.err.println("Trying to duplicate database entry for device"+d.GetDeviceName());
            return false;
        }
        deviceList.put(d.GetDeviceId(), d);
        p = new Position(d.GetPosX(), d.GetPosY(), d.GetDeviceId());
        mapTable.put(d.GetDeviceId(), p);
        d_ids[NumberOfDevices] = d.GetDeviceId();
        for (int i = 0; i <NumberOfDevices;i++)
        {
            device x = (device)(deviceList.get(d_ids[i]));
            x.neighbors.add(x);
        }
    }
```
public boolean RemoveDeviceFromDbase(device d) {
    if (NumberOfDevices == 0 || deviceList.isEmpty())
        return false;
    deviceList.remove(d.GetDeviceId());
    mapTable.remove(d.GetDeviceId());
    for (int i = 0; i < NumberOfDevices; i++)
    {
        device x = (device)(deviceList.get(d_ids[i]));
        x.neighbors.remove(d);
        if (d_ids[i] == d.GetDeviceId())
        {
            d_ids[i] = d_ids[NumberOfDevices - 1];
            d_ids[NumberOfDevices - 1] = null;
            break;
        }
    }
    NumberOfDevices --;
    return true;
}

public Enumeration GetLatestMap(String caller_id) {
    device caller = (device)(deviceList.get(caller_id));
    Position temp;
    device d;
    LocalMapTable.clear();
    if (NumberOfDevices == 0)
    {
        System.err.println("No devices in database");
        return null;
    }
    for (Enumeration en = mapTable.keys(); en.hasMoreElements();)
    {
        d = (device)(deviceList.get((String) en.nextElement()));
        temp = new Position(d.GetPosX(), d.GetPosY(),
                          d.GetDeviceId());
        mapTable.put((String) en.nextElement(), temp);
        if (!d.equals(caller) && d.getLastMovement() <
            caller.getLastRequest())
        {
            LocalMapTable.put((String) en.nextElement(),
                                temp);
        }
    }
    return LocalMapTable.elements();
}
public void SortNeighbors(String device_id) {
    long temp=0;
    int largest=0;
    device d = (device)(deviceList.get(device_id));
    device tempd;
    LinkedList nbrs = d.neighbors;

    for (int i=0; i<NumberOfDevices; i++) {
        temp = d.Distance((device)nbrs.get(i));
        for(int j = i; j<NumberOfDevices; j++) {
            if(temp < d.Distance((device)nbrs.get(j))) {
                temp = d.Distance((device)nbrs.get(j));
                largest = j;
            }
        }
        tempd = (device)nbrs.remove(largest);
        nbrs.addFirst(tempd);
    }
}

// get the spatial K nearest neighbors
public Enumeration getKNN(String device_id, int k) {
    device d = (device)deviceList.get(device_id);
    SortNeighbors(device_id);
    knn.clear();

    for(int i = 0;i<k;i++) {
        Integer temp = new Integer(i);
        knn.put(temp, ((device)(d.neighbors.get(i))).GetDeviceId());
    }
    return knn.elements();
}

Appendix – B

Here we provide listing of the client end of our system. Again, this is only a partial listing of only important classes and their non-trivial methods. For more information and complete code, please contact the authors.

B.1
This listing is for the device driver that retrieves data from the COM4 serial IO port of
the iPAQ. Remember that the TeleType GPS device writes raw GPS data to the COM4
port.

```c
BOOL PortInitialize (TCHAR * portName)
{
    DCB PortDCB;
    COMMTIMEOUTS CommTimeouts;

    //lpszPortName = _T("\\Temp\\GPSlog.txt");

    // Open the serial port.
    hPort = CreateFile
            (portName, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_ALWAYS, 0, NULL);
    // Handle to port with attribute
    // to copy

    // If it fails to open the port, return FALSE.
    if ( hPort == INVALID_HANDLE_VALUE )
        {
            // Could not open the port.
            unsigned short msg [1000];
            dwError = GetLastError ();
            wsprintf (msg, _T("Unable to open the port %s, error %u"),
                      portName, dwError);
            MessageBox (g_hMainWnd, msg,
                        TEXT("Error"), MB_OK);
            return FALSE;
        }

    PortDCB.DCBlength = sizeof (DCB);

    // Get the default port setting information.
    GetCommState (hPort, &PortDCB);

    // Change the DCB structure settings.
    PortDCB.BaudRate = CBR_4800;              // Current baud
    PortDCB.fBinary = TRUE;                  // Binary mode; no EOF check
    PortDCB.fParity = TRUE;                  // Enable parity checking
    PortDCB.fOutxCtsFlow = FALSE;            // No CTS output flow control
    PortDCB.fOutxDsrFlow = FALSE;             // No DSR output flow control
    PortDCB.fDtrControl = DTR_CONTROL_DISABLE;
              // DTR flow control type
    PortDCB.fDsrSensitivity = FALSE;          // DSR sensitivity
    PortDCB.fTXContinueOnXoff = TRUE;        // XOFF continues Tx
    PortDCB.fOutX = FALSE;                   // No XON/XOFF out flow
    PortDCB.fInX = FALSE;                     // No XON/XOFF in flow
    PortDCB.fErrorChar = FALSE;               // Disable error replacement
    PortDCB.fNull = FALSE;                     // Disable null stripping
    PortDCB.fRtsControl = RTS_CONTROL_DISABLE;
            // RTS flow control
    PortDCB.fAbortOnError = FALSE;             // Do not abort reads/writes
```
PortDCB.ByteSize = 8;            // Number of bits/byte, 4-8
PortDCB.Parity = NOPARITY;       // 0-4=no,odd,even,mark,space
PortDCB.StopBits = ONESTOPBIT;   // 0,1,2 = 1, 1.5, 2

// Configure the port according to the specifications of the DCB
// structure.
if (SetCommState (hPort, &PortDCB) == 0)
{
    unsigned short msg [1000];
    dwError = GetLastError ();
    wsprintf (msg, _T("Unable to configure serial port %s, error %u"),
portName, dwError);
    MessageBox (g_hMainWnd, msg,
       TEXT("Error"), MB_OK);
    return FALSE;
}

// Retrieve the time-out parameters for all read and write
// operations
// on the port.
GetCommTimeouts (hPort, &CommTimeouts);

// Change the COMMTIMEOUTS structure settings.
CommTimeouts.ReadIntervalTimeout = MAXDWORD;
CommTimeouts.ReadTotalTimeoutMultiplier = 0;
CommTimeouts.ReadTotalTimeoutConstant = 0;
CommTimeouts.WriteTotalTimeoutMultiplier = 10;
CommTimeouts.WriteTotalTimeoutConstant = 1000;

// Set the time-out parameters for all read and write operations
// on the port.
if (!SetCommTimeouts (hPort, &CommTimeouts))
{
    // Could not create the read thread.
    MessageBox (g_hMainWnd, TEXT("Unable to set the time-out
parameters"),
       TEXT("Error"), MB_OK);
    dwError = GetLastError ();
    return FALSE;
}

// Direct the port to perform extended functions SETDTR and SETRTS
// SETDTR: Sends the DTR (data-terminal-ready) signal.
// SETRTS: Sends the RTS (request-to-send) signal.
EscapeCommFunction (hPort, SETDTR);
EscapeCommFunction (hPort, SETRTS);

// Create a read thread for reading data from the communication
port.
if (hReadThread = CreateThread (NULL, 0, PortReadThread, 0, 0,
       &dwThreadID))
{
    CloseHandle (hReadThread);
}
else
{
// Could not create the read thread.
MessageBox (g_hMainWnd, TEXT("Unable to create the read thread"),
    TEXT("Error"), MB_OK);
dwError = GetLastError ();
return FALSE;
}

return TRUE;
}

// read module

DWORD PortReadThread (LPVOID lpvoid)
{
    BYTE Byte;
    DWORD dwCommModemStatus,
        dwBytesTransferred;

    // Specify a set of events to be monitored for the port.
    SetCommMask (hPort, EV_RXCHAR);  // | EV_CTS | EV_DSR | EV_RLSD | EV_RING);

    while (hPort != INVALID_HANDLE_VALUE)
    {
        // Wait for an event to occur for the port.
        if (WaitCommEvent (hPort, &dwCommModemStatus, 0) != 0) {
/*
    unsigned short msg [1000];
    dwError = GetLastError ();
    wsprintf (msg, _T("Error WaitCommEvent %u"), dwError);
    MessageBox (g_hMainWnd, msg,
        TEXT("Error"), MB_OK);
*/
        }

        // Re-specify the set of events to be monitored for the port.
        SetCommMask (hPort, EV_RXCHAR);  // | EV_CTS | EV_DSR | EV_RING);

        if (dwCommModemStatus & EV_TXEMPTY) {
            MessageBox (g_hMainWnd, TEXT("BINGO"), TEXT("Info"), MB_OK);
        }

        if (dwCommModemStatus & EV_RXCHAR)
        {
            // Loop for waiting for the data.
            do
            {
                // Read the data from the serial port.
                ReadFile (hPort, &Byte, 1, &dwBytesTransferred, 0);

                // Display the data read.
                if (dwBytesTransferred == 1)
                {
                    void ProcessChar (BYTE byte);
                    ProcessChar (Byte);
                }
B.2

Here we give listing of the client side socket programming. This is implemented in embedded Visual C++ for Windows CE.

    // connecting to server

BOOL CNWSocket::ClientConnect(SOCKET * psock, HWND hwndDlg)
{
    BOOL            fSuccess = FALSE;
    int             nRetries = 0;
    struct sockaddr_in  addrT;
    int             nRet;

    memset(&addrT, 0, sizeof(addrT));
    addrT.sin_family = AF_INET;
    addrT.sin_port = htons(iSockPort);
    addrT.sin_addr.s_addr = INADDR_ANY;

    nRet = bind(*psock, (PSOCKADDR)&addrT, sizeof(addrT));
    if ( nRet != SOCKET_ERROR )
    {
        if ( listen(*psock, 0) != SOCKET_ERROR )
        {
            while ((fSuccess == FALSE) &&
                   (++nRetries < MAX_RETRIES))
            {
                struct fd_set  fds;
                struct timeval tv = { DELAY_TIME / 1000,
                                      DELAY_TIME % 1000 };  
                FD_ZERO (&fds);
                FD_SET (*psock, &fds);

                if (select(0, &fds, NULL, NULL, &tv) == 0)
                    continue;
                if (FD_ISSET(*psock, &fds))
                {
                    // accept the connection and close the
                    // socket we were listening to
                    SOCKET oldsock = *psock;

                    *psock = accept(oldsock, NULL, 0);
                    closesocket(oldsock);
                    fSuccess = (*psock != INVALID_SOCKET);
                }
            }
        }
    }
    return fSuccess;
}
// client sending data to server. This method is called for making
queries

BOOL CNWSocket::ClientConnectSend()
{
    BOOL fSuccess = FALSE;
    int nRetries = 0, nRet;
    struct sockaddr_in addrT;
    PHOSTENT phe;
    //int error_no;
    //char * error;
    //LPCTSTR error_msg;
    // ppp_peer returns the IP adress of the host machine
    // if CE Services is running or NULL otherwise
    //if ( !(phe = gethostbyname("wedge.ics.uci.edu")) )
    if ( s_SocketSend == INVALID_SOCKET )
    {
        return FALSE;
    }
    for (fSuccess = FALSE;
         nRetries < MAX_RETRIES && ! fSuccess;
         nRetries++, Sleep(DELAY_TIME)
    )
    {
        memset(&addrT, 0, sizeof(addrT));
        addrT.sin_family = AF_INET;
        addrT.sin_port = htons(iSockPortSend);
        addrT.sin_addr.s_addr = *(long *) phe->h_addr;
        nRet = connect(s_SocketSend, (PSOCKADDR)& addrT,
                       sizeof(addrT));
        if ( nRet != SOCKET_ERROR) {

            return fSuccess;
        }
    }
    if ( !(phe = gethostbyname("leapfrog.ics.uci.edu")) )
    {
        AfxMessageBox(_T( "connection error"));
        return FALSE;
    }

    return TRUE;
}

for (fSuccess = FALSE;
     nRetries < MAX_RETRIES && !fSuccess;
     nRetries++, Sleep(DELAY_TIME)
)
{
    memset(&addrT, 0, sizeof(addrT));
    addrT.sin_family = AF_INET;
    addrT.sin_port = htons(iSockPortSend);
    addrT.sin_addr.s_addr = * (long *) phe->h_addr;
    nRet = connect(s_SocketSend, (PSOCKADDR)& addrT,
                   sizeof(addrT));
    if ( nRet != SOCKET_ERROR) {
fSuccess = TRUE;

AfxMessageBox(_T("socket connection successful"));

return (TRUE);

//MessageBox(hwndDlg, _T("success"),
//_T("Success connected "), MB_OK);

//TCHAR test[20]=_T("hello");
//SendText(*psock,(LPCTSTR)test);

} else {

AfxMessageBox(_T("socket connection error"));

return (FALSE);
}

} s_SocketSend = INVALID_SOCKET;

return fSuccess;