A Tool For Real-Time Location Tracking of Mobile Nodes

Xiaotao Shu
December, 2004
Supervised by professor Michel Barbeau

Contents

1. Introduction

This project is about a tool for tacking in real time the geographical locations of mobile nodes. The purpose of this tool is to gather data regarding the mobility patterns of authorized nodes over a period of time. The data can subsequently be used for the purpose of user mobility profiling and simulations.

The Automatic Position Reporting System (APRS) is open-source system, developed by Bob Bruninga for tracking objects using amateur radio. It captures and reports on position, weather and other information for a geographical location. APRS exchanges information between a large number of stations covering a local area.

The tool described in this report is a substitute to APRS. It can receive the geographical position information of mobile nodes using amateur radio, plot the tracked nodes on a map, and log position data in files. APRS is a monolithic application. In this project, we address an additional feature for the location tracking software where the observer (interface) and data sources are not co-located. This tool, which consists of client programs and a server program, is distributed an application. A client program is lean and collects mobile node data from an amateur radio transceiver and transfers mobile node position information to a server. In the system, multiple clients may be distributed around the world. A server of the tool is able to keep track of mobile nodes in different regions on different map views.

The framework of this tool is illustrated in Figure 1.1

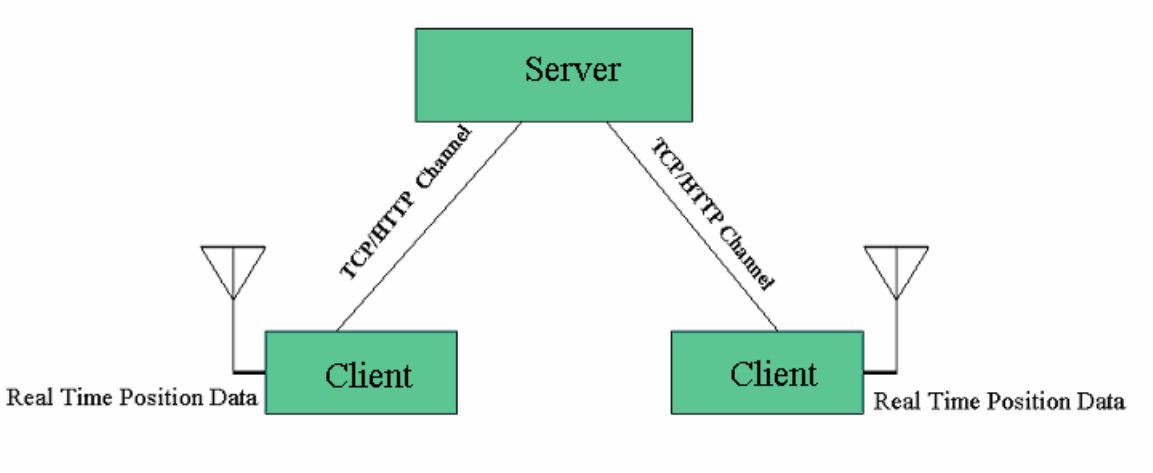

Figure 1.1 Framework of the Tool

The rest of this report comprises three sections. Section two describes the architecture of the tool. Section three discusses the implementation of the core components of the tool. Section four summarizes the report, reviewing the advantages of this tool and proposing improvements that may be done in the future.

2. Architecture

2.1 Hardware Architecture and Data Flow

Figure 2.1 represents the hardware architecture and data flow of user location-related information collection and processing.

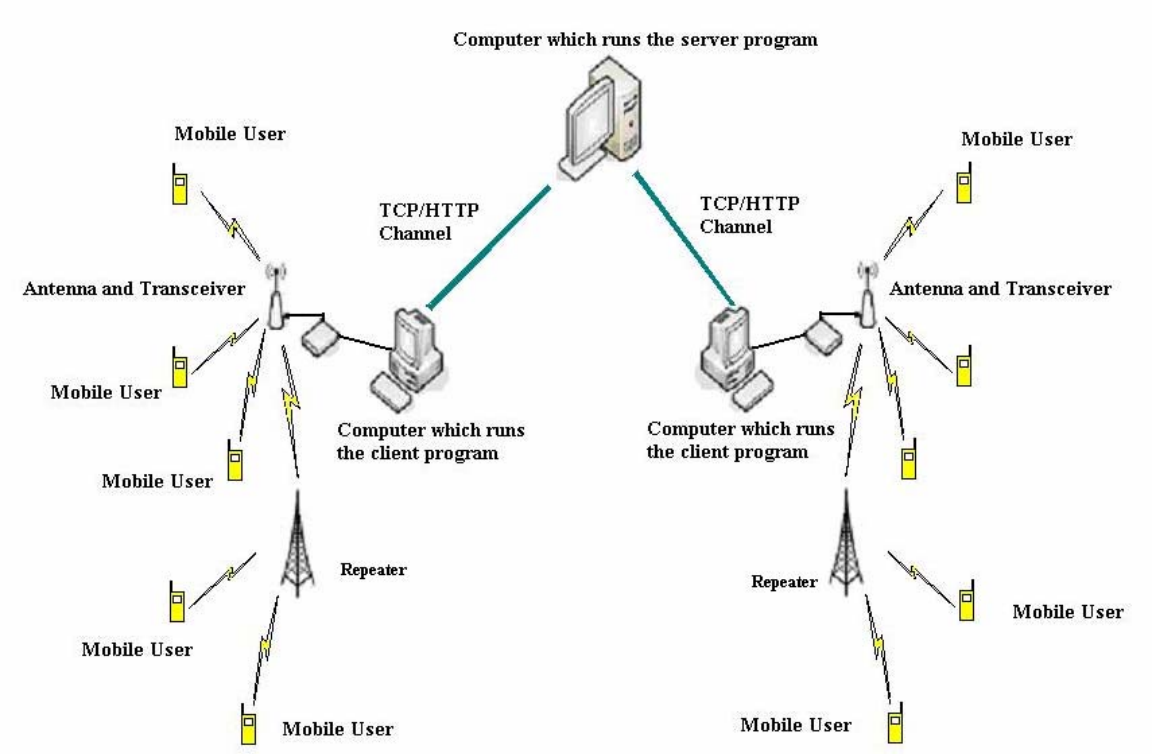

Figure 2.1 Hardware Architecture

1. Location-related data, transmitted by users within the range of the antenna, is received by a transceiver. Transmission from users outside the range of the antenna is first received by a repeater and then retransmitted to the transceiver.
2. The transceiver demodulates the location broadcast and transfers the data to a computer, in which a client program of our tool is running. The client program logs in real-time the geographical position data (e.g. latitude, longitude and time) in a file. It also transfers the data to the computer in which a server program of tool is running. The tools support multiple clients that can be located at different

geographical locations. The client and server communicate using either a TCP channel or a HTTP channel.

3. On the server part, the position coordinates are dynamically displayed on a map of a given region, e.g. city or province and recorded in the user position database that can be used for the purpose of the mobility profiling.

2.2 Software Architecture

The tool's software architecture is based on a distributed Model-View-Controller (MVC) model. Figure 2.2 shows the software architecture.

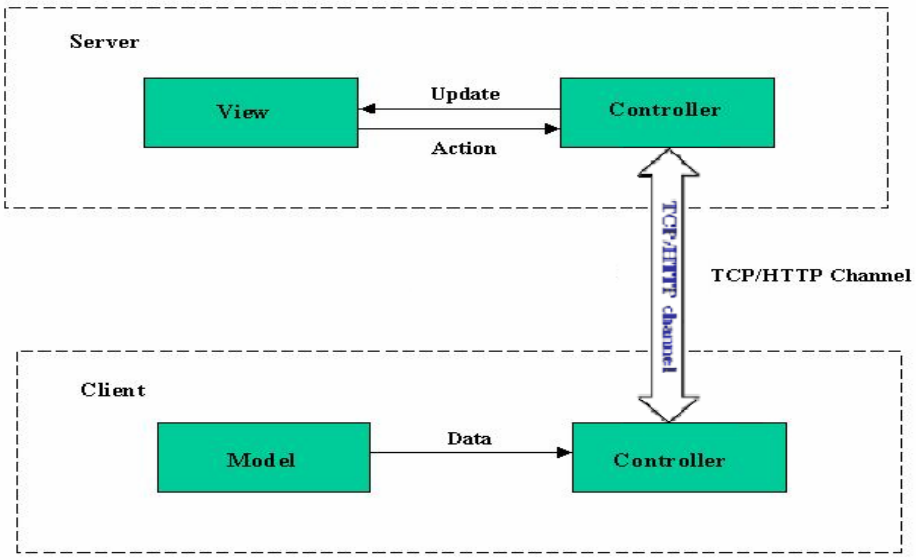

Figure 2.2 Software Architecture

1. The model is a source of data. It is the client part of the tool. It is responsible for receiving the mobile node position data from a transceiver. The data is transferred to a computer through a serial port.

2. The view is the graphical user interface of the tool. It keeps track of the moving objects on a map. The view is the server part. The server can display multiple views. Figure 2.3 shows the Ottawa region tracking map.

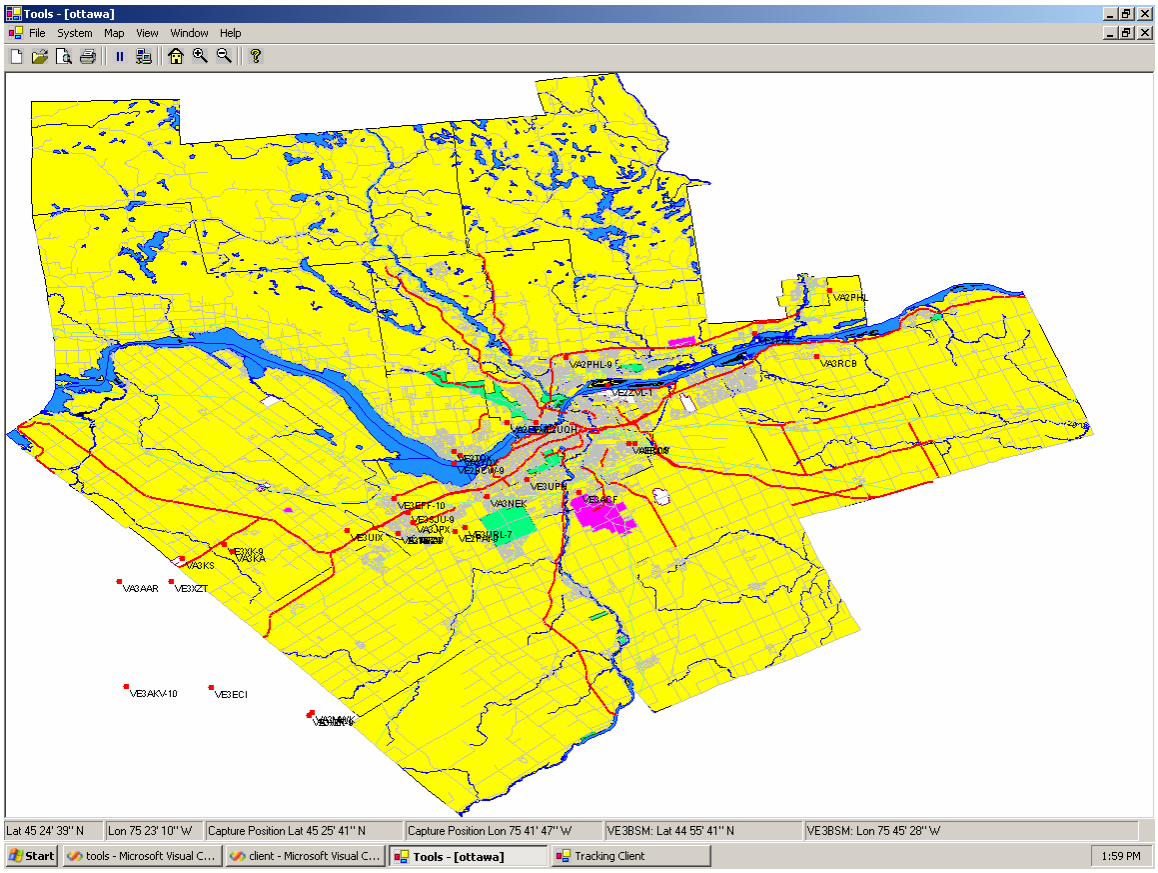

Figure 2.3 Ottawa region tracking map

3. The controller insures the consistency between the model and view. It gets data from the model reactively and updates the view according to the new data. The controller is distributed in both the client part and server part. A client and a server are connected through either a TCP channel or a HTTP channel.

3. Implementation of the Tool

The tool is implemented using C# under the Windows environment. It carries out the following tasks: real-time positions of mobile nodes collection and processing, plotting tracked nodes on maps, and logging tracking information into files. The tool consists of model (data source), view and controller parts, which are distributed among a client program and a server program. The main programming techniques employed in the tool include: serial port communication in C#, multiple threads, distributed objects, and vector map displaying. Figure 3.0 is a sequence diagram of the data collection and processing.

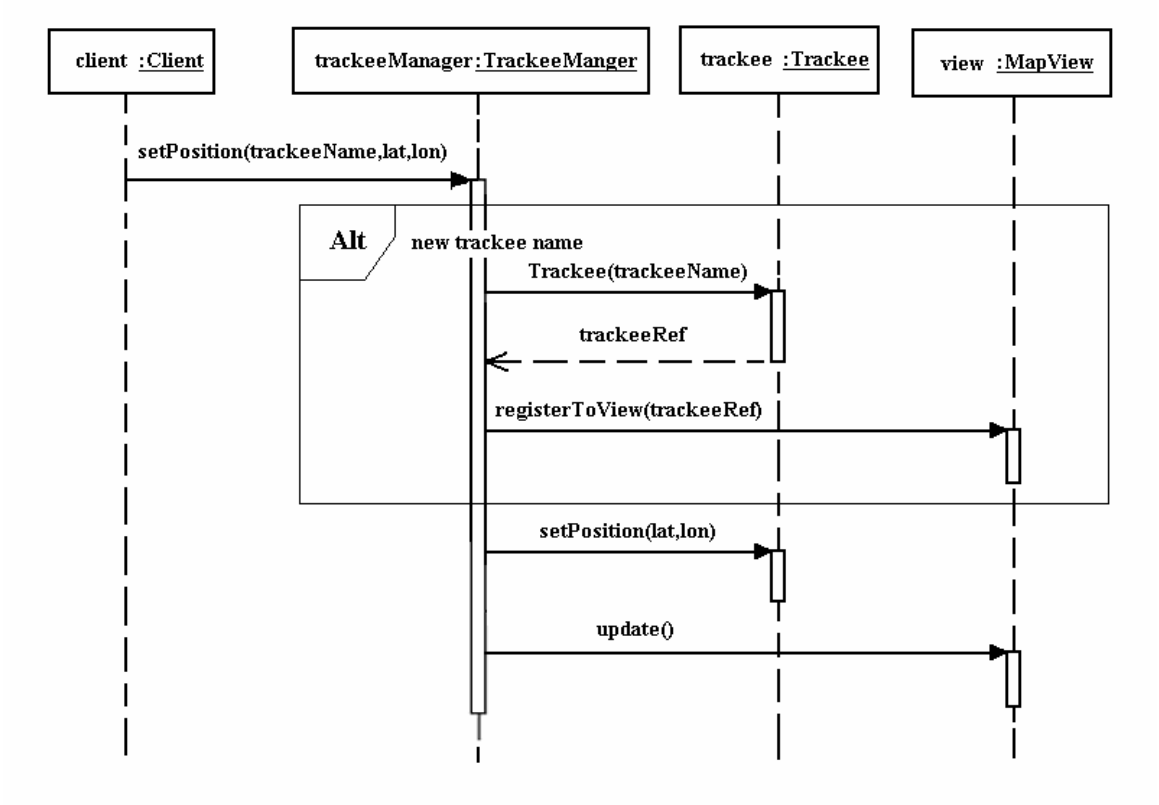

Figure 3.0 Sequence Diagram of the Data Collection and Processing

There are four entities “client”, “trackeeManager”, trackee” and “view” in this sequence diagram. The client is the client side object that receives position messages from a serial

port monitor. Then it invokes the *setPosition(trackeename, lat,lon)* operation on the trackeeManager, which is a remorting object that is created and published by the server. If the trackeeName is new, the trackeeManager create a trackee with this new trackeeName. The trackee is a server side object that corresponds to a mobile node with a call-sign, a geographic location, and a color. It is registered with the view.

When the trackee is alive, the trackeeManager invokes the trackee's *setPosition(lat,lon)* operation to set the new location of this trackee. The trackeeManager also requests the *update( )* operation on the view to update its graphical representation.

3.1 Model Implementation

Receiving of real time position of mobile nodes from a radio transceiver is the model part of the tool. It is implemented in a client program. The client program receives the mobile node location-related data from a transceiver through a serial port.

DOTNET offers no built-in class for the serial communications. We developed a class called SerialPort to implement the serial port communications using the Win32 API, which provides the low level functions that can be used to open, close, and manipulate serial ports, transmit and receive data, and manage connections. Code executed under the control of the common language runtime (CLR) is called managed code [8]. The Win32 serial communication function is unmanaged code that runs outside the CLR. We use P/Invoke to wrap the API functions, constants, and structure definitions as static members of the managed SerialPort class.

Since the messages that the mobile nodes send don't arrive at a regular time interval, a monitor is needed to listen to the serial port continuously. This monitor function is executed in a independent thread. Therefore, the client program is designed as a multiple threads application. There are two types of threads in the client part. One is the graphical user interface (GUI) thread, which provides an interface between a client program and an user. In the client GUI, users can set the parameters of connections between a client and a server, such as server's IP address, port number and communication channel type (TCP/HTTP), and set the parameters of a serial port communication, such as serial port number and baud rate. Users can also connect/disconnect to/from a server, start data collection and stop data collection through this GUI. The other thread is a serial port monitor thread that is responsible for serial port communication between a transceiver and a client program. Figure 3.1.1 shows the GUI of the client part.

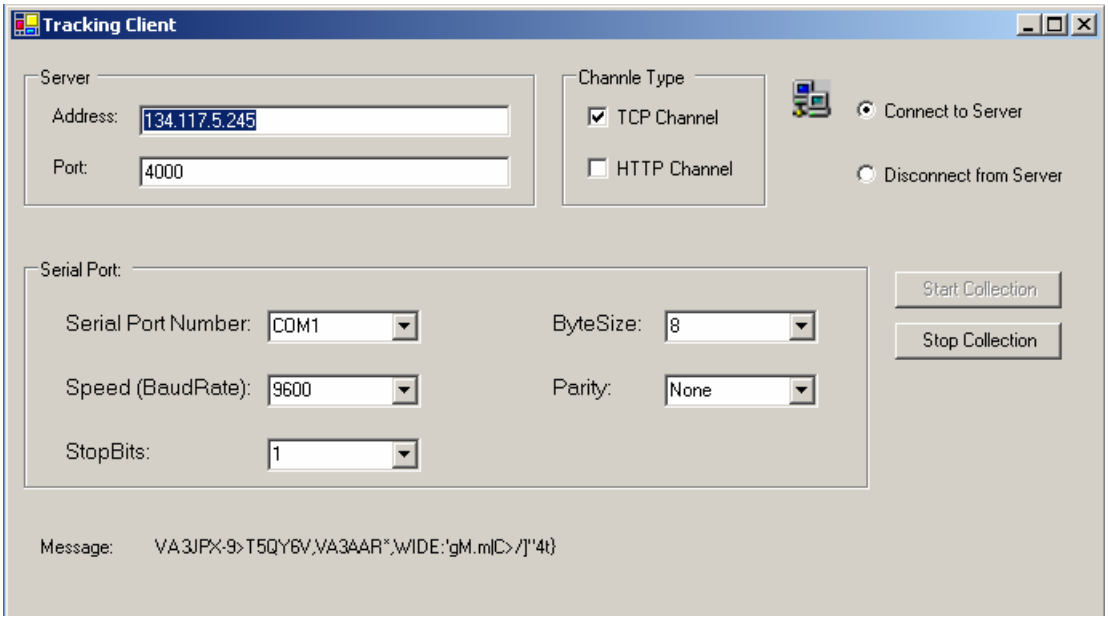

Figure 3.1.1 GUI of the Client Part

3.2 Distributed Controller Implementation

A controller insures the consistency between a model and a view. It gets data from the model reactively and updates the view according to the new data. Our controller is distributed in both a client process and a server process. The distributed controller is implemented using the DOTNET remoting object. DOTNET remoting object is a windows object-oriented middleware. It is a technology that allows for clients and servers to communicate with each other, in a location transparent manner. The client and server can be on the same machine, on different machines on the same network, or in different countries across the world.

Like in other distributed computing technologies, such as CORBA, DCOM and java/RMI, in DOTNET Remoting, a client object doesn't make a direct call to the remote object, rather it creates a proxy object and then uses the proxy object to invoke methods of the remote object. When the client object calls a method on the remote object via the proxy, the call is marshaled to the SOAP/Binary formatter. The SOAP/Binary formatter is transferred to the remote object via a channel, where the method is executed. There are two built-in channels in DOTNET, TCP channel and HTTP channel. A TCP channel is faster than a HTTP channel. It is good for binary data; whereas a HTTP channel is required for firewall traversal. A HTTP channel can be used in a WAN, and a TCP channel can be used in a LAN. Figure 3.2.1 pictures the connection parameter setting GUI of the tool on the server side.

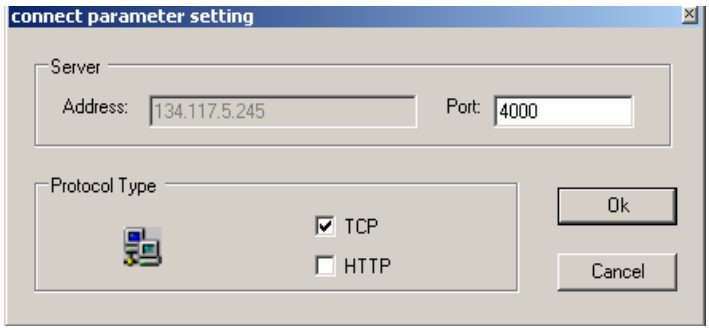

Figure 3.2.1 Connection Parameter Setting Form

All DOTNET remote objects are created by the server. There are two types of objects supported by DOTNET Remoting: Client-Activated Object (CAO) and Server-Activated Object (SAO). CAO is created by the server but its lifetime is managed by the client. A CAO is created as soon as the client calls the *New* method or any other object creation method. Both the default and non-default constructors can be used with CAOs. But a CAO is specific to a client and not shared among different clients; object instance exists until the lease expires or the client destroys the object. A SAO is created by the server. Its lifetime is managed by the server. A SAO is created when the client actually invokes a method on a proxy. Only default constructors (constructors with no parameters) are supported for SAOs.

There are three modes in which SAOs can be activated:

1) Singleton**:** Only one object will be created on the server to fulfill the requests of all the clients, which means the object is shared and the state is shared by all the clients. [6]

2) SingleCall: Such objects are created on each method call and objects are not shared among clients. State should not be maintained in such objects because they are destroyed after each method call. [6]
3) Dynamic publication. This is a type of server activation that offers more control over object construction by providing a programmatic publication mechanism. This allows for publishing a particular object at a particular URL, optionally with a parameterized constructor. Architecturally this should be viewed as a subtle variant of a server-activated Singleton mode. [3]

There is no naming service in DOTNET Remoting. Each published remote object should have an unique identity. The client activates a published remote object directly at the particular URL using its unique identity.

Our tool implements the distributed controller using a dynamically published remote object. Only one remote object called TrackeeManager is created and dynamically published at a defined URL by the server program. The TrackeeManager is responsible for creating one local “Trackee” object for each mobile node on the server side, setting and recording the new position of the “Trackee” object. Finally the TrackeeManager update the multiple views that have been dynamically registered to the controller.

3.3 Multiple View Implementation

View is the GUI of our tool. It presents the moving objects on a map. A view is in the server program. The server supports multiple views. The user can open multiple views on

the server side to monitor the moving objects on different region maps. Figure 3.3.1 is a snapshot of the GUI. The left view displays the portion of Ottawa area. The right view monitors the South-Ontario region. Locations of a number of nodes are pictured on each map.

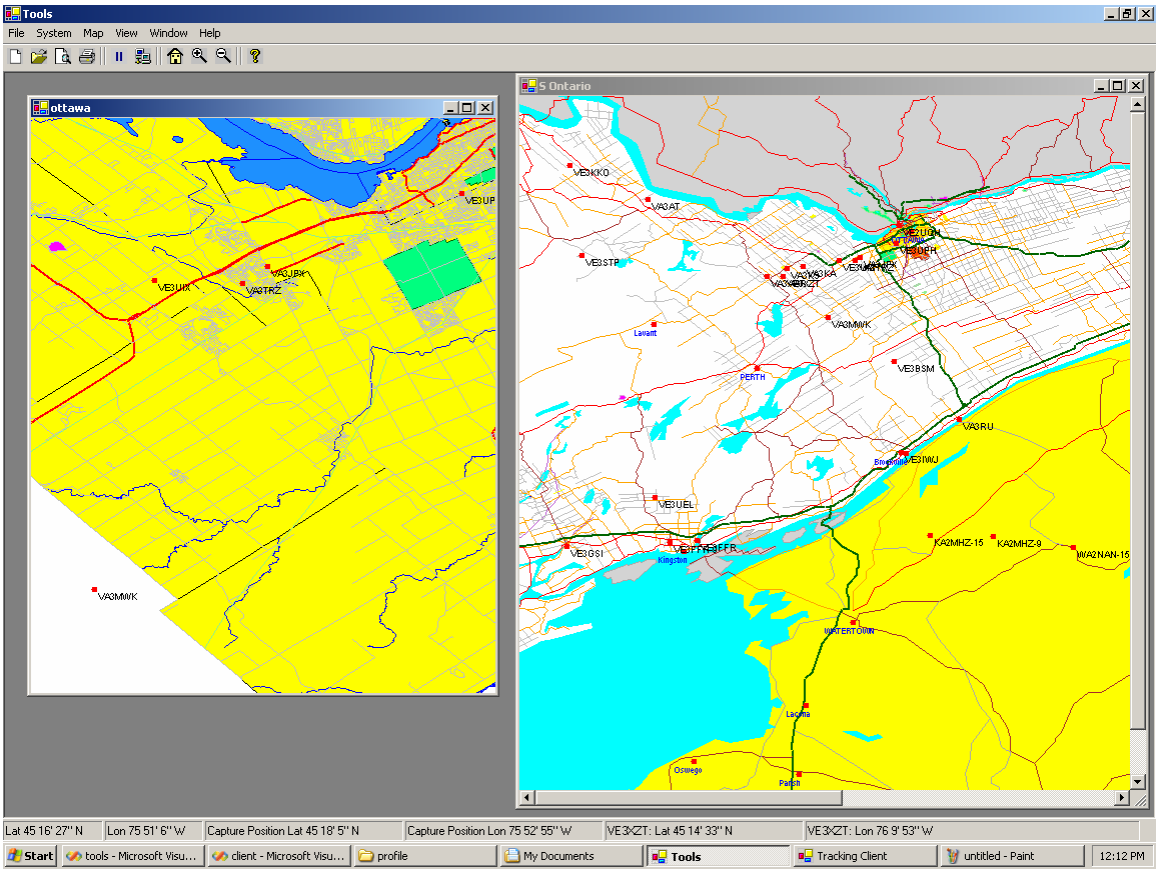

Figure 3.3.1 Multiple Views

Almost all the provincial maps of Canada are available in a database of the tool. The map database was built in the following way: maps are stored in map files under the map source directory. The related information including map title, range, size and map file name, is recoded in an XML file. The user can load an existing map from a map list or load a map file directly. The server program provides a map source management tool for

users to maintain the map source directory. Figure 3.3.2 is the GUI of the map source management.

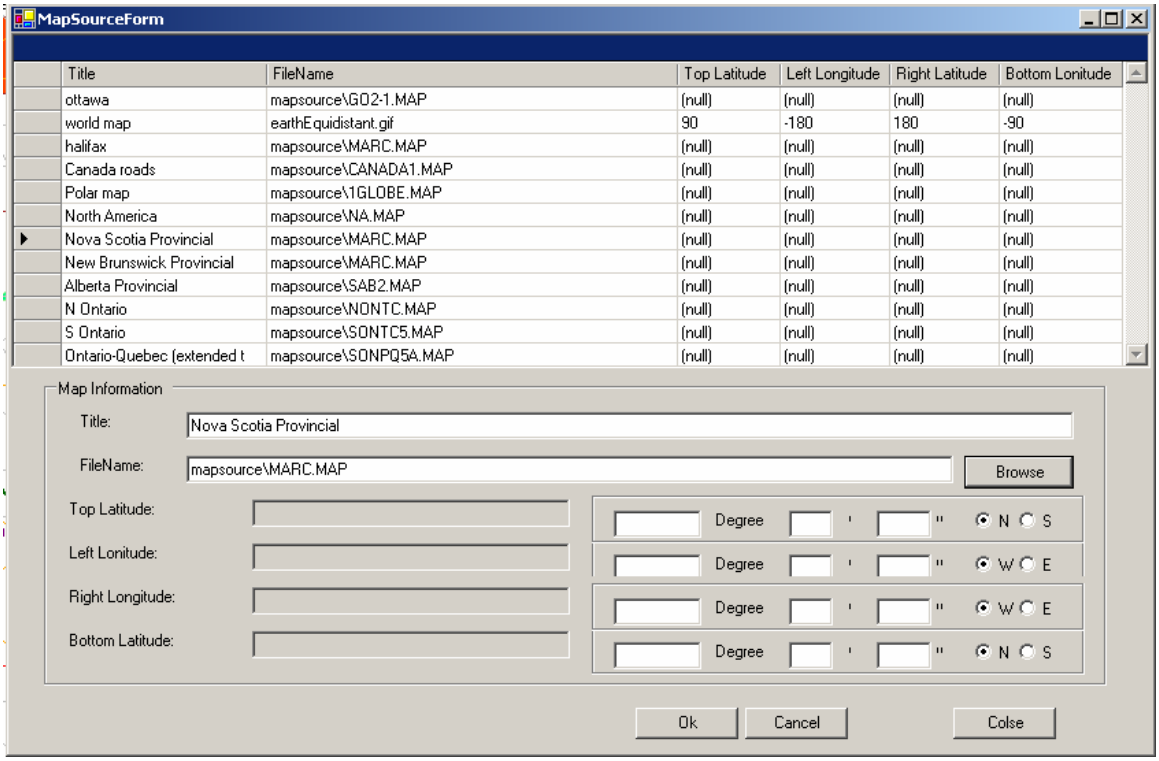

Figure 3.3.2 Map Source Management Form

Our tool supports both the bitmap file format (such as *.bmp, *.gif, *.jpg, etc) and vector map file format (*.map). A bitmap is composed of grid pixels. DOTNET GDI+, which is an advanced windows graphics design interface (GDI), supports the bitmap format. It has two built-in classes: Image and Bitmap. Loading and displaying a bitmap on GDI+ is as follows: create a bitmap object, load a bitmap from the bitmap map file, and invoke Graphics::DrawImage to draw a bitmap at a specific coordinate location on the view. The disadvantage of using the bitmap format is that it cannot be zoomed in/out smoothly. The vector map format is composed of vector objects such as points, lines,

polygons or text data (labels and symbols). It can unlimitedly be zoom in/out with smooth appearance.

In the tool, the view contains an Image object called m_backgroud and a VectorMap object called m_VectorMap. When the .MAP file is loaded, the tool parses all vector objects such as points, lines, polygons, colors, labels and symbols from the .MAP file, creates the VectorMap object, invokes the setBackground() method to draw the VectorMap object to m_backgroud with a defined scale and size and refreshes the view. Caching this m_background is more efficient than to draw the vector map every time we repaint the view. When the view is repainted, only the m_background is drawn on the view's GDI+. If zoom percentage or view size is changed, then the setBackground() method have to be invoked.

Vector map zooming in/out is done as follows:

1) Gets the view size, zoom percentage and geographical position of center of the view.
2) Calculates the offset of the latitude and longitude using the following formula:

```
m_latOffset = m_capturePos.Latitude-(m_vectorMap.m_bottom-m_vectorMap.m_top)/(2.0*m_currentZoom);
m_lonOffset = m_capturePos.Longitude-(m_vectorMap.m_right-m_vectorMap.m_left)/(2.0*m_currentZoom);
```

Figure 3.3.3 illustrates the above formula.

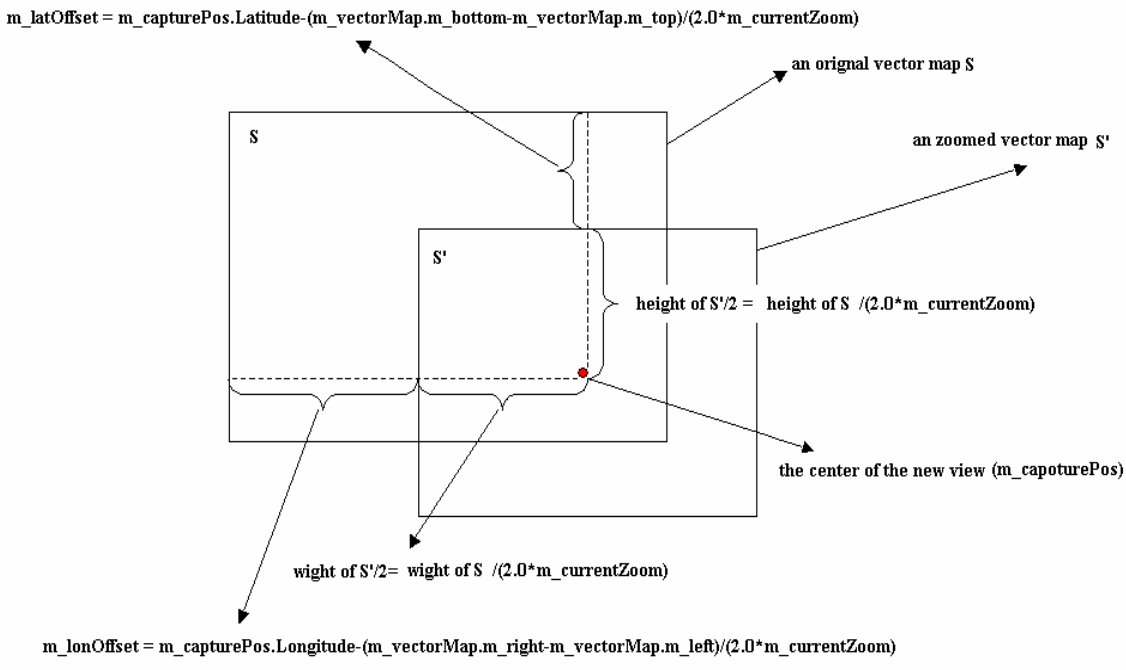

Figure 3.3.3

3) Invokes setBackground(): Figures out the geographical position on the vector map that corresponds the screen coordinate (0,0) depending on the offset of the latitude and longitude; draws all points, lines, polygons, labels and symbols to the m_background that is a Image object.
4) Refreshes the view. When a view repaint, the background and all Trackees, which have registered to the view, will be drawn on the view's GDI+.

Our tool supports multiple views. The controller keeps a list of its views, provides member functions for adding and removing views, and supplies the updateAllViews() function for letting multiple views know when the model's data have been changed.

4. Conclusions

The tool had been running continuously for almost one month. It has received a total 69661 location-related messages from 286 mobile nodes. The geographic location covers a portion of the Ontario-Quebec area. The latitude range is from 48.5891N to 56.7002N. The longitude range is from 68.4203W to 85.6993W. The testing results show that our tool is scalable and can be used as a substitute to APRS in mobile node location-related data collection and processing. Our tool possesses the following advantages:

1) Our tool is a distributed system and clients can be distributed all over the world.
2) Our tool supports multiple views.
3) Our tool records the position information of each mobile node in a file. The user position database can be used for the purpose of mobility profiling directly.

A few improvements remain to be addressed:

1) A few modifications to the tool are needed to support multiple servers such that a single client can communicate with multiple servers that are globally distributed.
2) Some mobile nodes only send one position message during a long time intervals. The position corresponding to these nodes don't need to be always displayed on a map. A solution for solving this problem is as follows:
    a) Create remoting trackee objects to substitute to local trackee objects.

b) Set the lifetime of Trackee remorting objects. When a remoting trackee object expires, deregister it from the view that it has been registered.

3) Modify the denotation of mobile nodes on the map. The tool can use abundant symbols to track the nodes. A symbol can denote the type of mobile nodes.

Reference

[1] Jeyanthi Hall. Shaoying Zou. Anomaly-based Intrusion Detection using Mobility Profiles of Wireless/Mobile Subscribers - Phase II. July 15, 2004.

[2] Michel Barbeau, Distributed Object Management and Transaction Processing System, Course Notes, Fall 2003.

[3] .NET Remoting Architectural Assessment, Pat Martin, Microsoft Corporation, May 2003,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetremotearch.asp

[4] Introductory Theory: Processes, Applications, Distributed Applications,
http://www.dotnetjohn.com/articles/articleid84.aspx

[5] Writing Distributed .NET Applications, Rick Ross, Pillar Technology Group,
http://rick-ross.com/papers/borcon2003/1207.html

[6]Tracking Services in .NET Remoting and Implementing Your Own Tracking Handlers, By Mansoor Ahmed Siddiqui, http://www.15seconds.com/issue/030225.htm

[7] How to Use .Net Remoting Using C#?, Mohamed Ashraf,
http://www.csharphelp.com/archives2/archive460.html

[8] An Overview of Managed/Unmanaged Code Interoperability, Sonja Keserovic and David Mortenson, Microsoft Corporation, October 2003
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/manunmancode.asp

[9] Use P/Invoke to Develop a .NET Base Class Library for Serial Device Communications, John Hind,
http://msdn.microsoft.com/msdnmag/issues/02/10/NETSerialComm/

[10] Serial Communications in Win32, Allen Denver, Microsoft Windows Developer Support,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfiles/html/msdn_serial.asp

[11] MacAPRS/WinAPRS Map File Format (Version 1.0 files) , Mike Musick
http://www.winaprs.org/MapFile.htm

[12] Introduction to the Global Positioning System for GIS and TRAVERSE
http://www.cmtinc.com/gpsbook/

[13] APRS Protocol Reference, protocol Version 1.0, The APRS Working Group, 29, August, 2000.