In: Proceedings of 15th Annual Small Satellite Conference, Logan, Utah, 2001.

SSC01-VIIIa-4

A Virtual Ground Station Based on Distributed Components for Satellite Communications

Steve Bernier
Communications Research Centre
3701 Carling Avenue
Ottawa, ON, Canada K2H 8S2
steve.bernier@crc.ca

Michel Barbeau
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON, Canada K1S 5B6
barbeau@scs.carleton.ca

Abstract. Communication with Low-Earth Orbit (LEO) satellites requires the set up of a ground station which is a complex and costly installation. Moreover, a LEO satellite is accessible only during certain time slots from a given ground station. In other words, access to LEO satellites is on an intermittent basis and is constrained by the availability of ground stations.

The work presented in this paper aims at augmenting the accessibility to the services offered by LEO satellites. We have devised a concept of virtual ground station available over the Internet. A virtual ground station can be used by any client with a computer attached to the Internet which augments the degree of accessibility. Besides, a virtual ground station and its clients don't have to be collocated. A client can access a satellite as long as a remote virtual ground station has access to it. As long as there are several virtual ground stations distributed at several locations, this architecture augments the degree of accessibility to satellites.

The design of the virtual ground station is based on CORBA distributed components. The virtual ground station has been developed using an application framework we have created, hence reducing the amount of programming required to obtain it. Moreover, we have developed a client satellite tracking software that uses our virtual ground station.

In this paper, we review the design and implementation of our virtual ground station concept. We also present the companion client satellite tracking software that we have developed.

1. Introduction

Since the launch of Sputnik in late 1957, satellite communications (satcom) have come a long way. From the transmission of a simple beacon, years of improvements brought satcom to the era of multimedia. Satellites are now used for many applications ranging from scientific earth and atmospheric studies to military tactical communications. Over the years, the number of satellites has constantly increased and as a consequence there are now more satcom services available.

However, the relative abundance of satcom services does not necessarily mean easy access. In fact, LEO satellites services have always been rather difficult to use. One of the reasons for this is the intricacy of the satellite tracking process. This process requires the simultaneous coordination of frequency tuning, antenna pointing and data acquisition. Only training can provide the skills required to manually track a LEO satellite. Consequently, helped by the emergence of personal computers, software applications were developed in order to address this issue. Such applications facilitate the tracking process by interfacing with the tracking equipment in order to automate all of the manipulations. The SatSy application[1] developed by our group represents an example of such an application. Nowadays, most tracking applications offer sophisticated graphical user interfaces (GUI) and they effectively reduce the tracking process to a few mouse clicks.

Nevertheless, those tracking applications don't address the most important problem in using satcom services, which is the limited number of ground stations. The basic reason why there isn't more ground stations is that they are rather expensive and complex to install. In addition, satcom antennas are quite cumbersome in that they require adequate space and have to be properly located. As of now, like SatSy, the vast majority of tracking applications are single computer applications. This type of application can only be used from the computer it is installed on. Since tracking applications need to be installed on a computer linked to tracking equipment, they become just as much inaccessible as the ground stations themselves. However, it is possible to overcome this limitation by connecting the tracking computers to a network and by using a network-enabled tracking application. This fairly new type of application enables users to start tracking sessions remotely from any computer connected to the same network as the ground station. When coupled to the Internet, this solution significantly increases ground stations accessibility.

To the best of our knowledge, our group has created the very first Internet-enabled tracking application. This software was devised in 1996 through a project called WATOO[2]. More recently, similar applications were created[3,4,5]. The WATOO software enables users to start tracking sessions through the Internet. For example, during demonstrations, WATOO was often used to track the Russian Space Station Mir from remote cities. Actually, it was even used to record radio amateur communications with MIR cosmonauts. As opposed to a single computer tracking application, a network-enabled version is divided into two parts: a server and a client. The server is very similar to the single computer version except it doesn't provide a user interface. The interface is provided by the client which is usually installed on remote computers. In this design, the client is responsible for sending commands to the server (through the network) based on user actions. The server is responsible for interpreting the commands to control tracking sessions. All the user needs is a computer connected to the network and the client application.

However, networking one single ground station isn't enough. The true benefit of this solution comes from networking many ground stations from all around the world. This enables a user to track a satellite no matter where it is, as long as there is a ground station in its footprint. Furthermore, with a sufficient number of networked ground stations, it is even possible to address another problem related to LEO satcom services: the intermittent nature of the services. In effect, on average, satcom services are only available a few times a day and usually for a maximum of approximately 15 minutes. Thus, even when an uninterrupted download of a file requires less than one hour, it can take several hours of elapsed time to download using a LEO satellite. In other words, several passes may be required to download a single file from a LEO satellite. One way to improve this situation is to consecutively use many different remote ground stations. We call this an extended tracking session.

The implementation of this solution requires a widespread use of a network-enabled tracking application. The first version of the WATOO server is an example of an application that could have been used to achieve this goal. However, because it was platform dependent, lots of modifications would have been required for each different ground station. Consequently, the WATOO software was completely redesigned in 1998. The WATOO server was developed such that it would be fairly simple to customize it for different ground stations. Also, technologies such as CORBA and JAVA were used to provide as much platform independence as possible. As of the GUI client, it was redesigned to take advantage of the new tracking server.

This paper presents the new design of the WATOO software. The first section offers a short overview of CORBA and distributed objects. The following section presents the design and implementation of our virtual ground station. The last section presents the GUI tracking client through a few screen shots. Finally, this paper ends with a conclusion and a couple ideas for future works.

2. CORBA Basics

The Common Object Request Broker Architecture (CORBA), is a specification produced by the Object Management Group (OMG) that addresses interoperability in distributed heterogeneous environments. The CORBA standard represents industry consensus from more than 800 companies. CORBA assumes a heterogeneous environment in which clients and servers implemented in different languages on different platforms can interoperate. There are many implementations of the CORBA standard, some of them in the form of commercial products that have demonstrated strong market acceptance.

CORBA enables clients and servers to interact together through a middleware called an Object Request Broker (ORB). The ORB is the mediator responsible for brokering interactions between clients and servers. Its job is to provide object location and access transparency by facilitating clients' use of servers' services. Since CORBA is object-oriented, clients and servers are represented as objects and services as methods. The set of public services of a server object is represented as an interface specification that must be specified using the CORBA standard Interface Definition Language (IDL).

Through the use of an IDL compiler, an interface specification is compiled to generate code into native language such as JAVA, C++, SmallTalk and others. Among other things, the IDL compiler generates a client

proxy class and a server skeleton class. Both classes are generated such that they can interact together through an ORB. To be more precise, an ORB is a set of classes that are provided by the ORB vendor.

Figure **1** illustrates the process through which an IDL interface specification (specification for short) is transformed into objects.

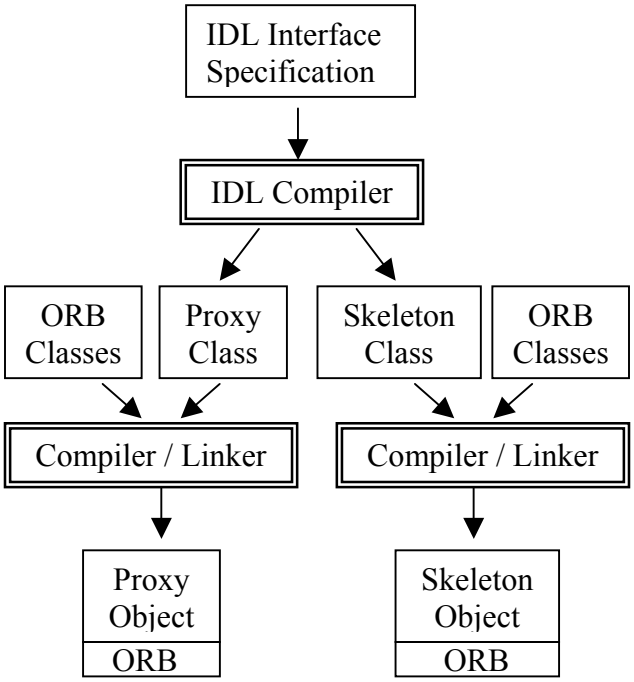

Figure 1. Generation of the Proxy and Skeleton Objects.

The proxy object represents a remote object locally. For example, if a client object needs to use the services of a remote object called FOO, it actually uses the services of a local FOO proxy. When accessed, the local proxy forwards all method calls to the remote FOO object using the ORB communication layer (see Figure 2). The client object doesn't know where the remote object is located. To create the proxy object, the generated proxy class is compiled with no modifications whatsoever. For the proxy to be accessed by the client object, it has to be generated in the same programming language. Therefore, if the client object is implemented using JAVA, the proxy class will have to be generated in JAVA as well. This local proxy can then be used to access a C++ remote object for which a C++ skeleton is needed as depicted in Figure 2.

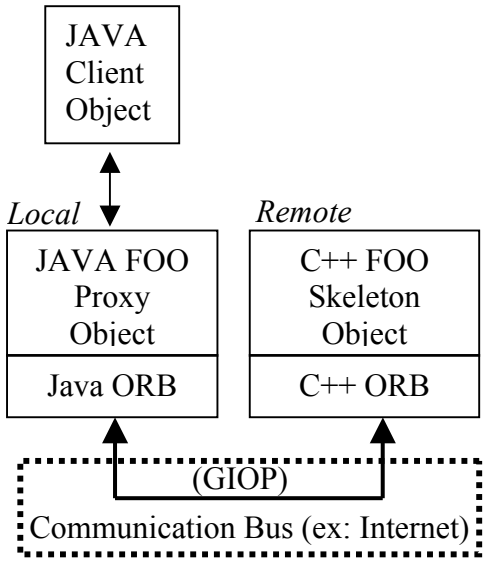

Figure 2. Communications Between a Client and a Remote Object.

To generate the JAVA FOO proxy class and C++ skeleton class, different IDL compilers are used. Figure 3 illustrates how this works. As mentioned before, both the proxy and the skeleton are generated such that they use the ORB services to interact together. Conceptually, an ORB is a communication layer. To communicate, it uses a standard protocol called General Inter-ORB Protocol (GIOP). This protocol is completely independent of programming languages and of execution environments (it takes care of byte-ordering issues). The ORB is responsible for mapping the generic constructs of GIOP (ex: method calls) to the native programming language and execution environment it is written for. Consequently, as shown in Figure 2, two ORBs are needed to enable communications between a JAVA and a C++ object.

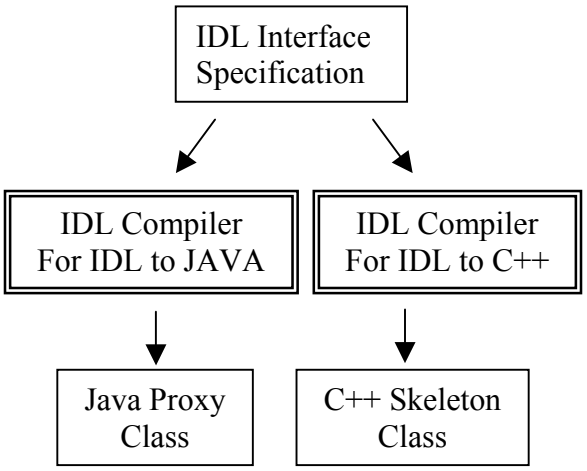

Figure 3. Using Different IDL Compilers to Offer Heterogeneity.

The skeleton class is where the developer implements the services (methods) of the server object. A generated skeleton is only partially implemented and needs to be filled in. As with the proxy, the implementation language dictates which IDL compiler will be used to generate the skeleton. In the FOO example, the skeleton is implemented in C++. The portion of the skeleton that is already implemented pertains to all the ORB interactions. This way the developer doesn't need to interact with the ORB at any time. All that needs to be done is to implement the methods representing the object's services. The generated code takes care of everything else.

The only thing that remains to be explained is how the local proxy finds the remote object. This happens in three steps. First, the remote object registers to a naming server. It does so by using an ORB service. ORB vendors provide naming servers as separate applications. They keep track of the name and location of every registered object. The second step consists of the client finding the remote object. The client object does this by querying a naming server through an ORB service. When the object is found, its reference is returned to the client object where it is used to create a proxy object. From there on, the client object can invoke methods on the remote object through the local proxy. There are many ways to use naming servers. The easiest way is to have clients and servers use the same naming server. It is also possible to use many different naming servers all connected together into a federation but this falls out of the scope of this basic.

In summary, using CORBA objects starts with an IDL interface specification. The specification is then compiled to generate specific language bindings for the clients (proxies) and the server (skeleton). The client object is written such that it uses the local proxy to get access to the server object's services. The services are implemented within the skeleton and compiled into a server object. The server object is registered to a naming server so that the client object can find it using a well-known name. The client object creates a local proxy from a remote object reference and starts invoking methods.

CORBA is fairly simple to use and provides numerous advantages. It is available for many different platforms, operating systems, and programming languages. This technology has matured over the last 10 years and it has been used for many types of application ranging from military real-time communications to secure online banking.

The main reason why CORBA was used in the second version of WATOO is its programming language and execution environment independence. Thus, the new WATOO server is now composed of a collection of CORBA objects that could all be implemented with a different language. For example, if one of our JAVA CORBA object is considered not to be performing fast

enough, it could be reimplemented using C++ without any impacts on the remaining objects.

3. Creating a Virtual Ground Station

This section presents the concept of a virtual ground station and offers a detailed description of its design. The framework we have developed to facilitate the creation of virtual ground stations is described with an example.

A virtual ground station is a software representation of a real life ground station. It is equipped with virtual equipement as a transceiver, an antenna and a rotor controller. Virtual equipment offers the same services its real life counter part offers. For example, the virtual transceiver can be turned on or off and its mode and frequency can be set. Like a ground station equipped with a tracking application, the virtual ground station offers services to start and stop tracking sessions. In addition, it has an owner and it knows where it's located in terms of latitude, longitude, and altitude.

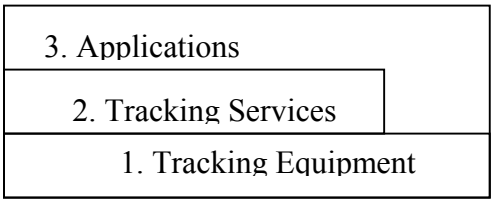

Figure 4. Three Layers of Services

The services offered by a virtual ground station can be grouped into two superimposed layers corresponding to layers 1 and 2 of Figure 4. Services contained in layer 1 pertain to the control of tracking equipment. Layer 2 services deal with the computation of satellite position, pass predictions, and such. The virtual ground station does not provide services of layer 3. These services are implemented by end-user applications. The WATOO GUI provides layer 3 services like the plotting of an orbital trace on Earth projections. The fact that both layers 1 and 2 are accessible from layer 3 offers a great degree of flexibility for the creation of new services.

The segregation of services into layers also helps for the creation of an application framework. This type of framework is a full implementation of an application that can easily be customized by the replacement of specialized components only. For example, the services of layer 1 will most likely have to be customized for different ground stations as opposed to services of layer 2. Consequently, we have created a framework that dictates how services of layer 2 interact with services of layer 1. This way, services of layer 1 can easily be replaced without changing anything in layer 2. Moreover, the components providing the layer 1 services have been designed such that they only need to be partially customized. As you will see in the implementation section, the use of an application framework coupled with technologies like CORBA and JAVA augments platform independence and simplifies customization when necessary.

3.1 Analysis and Design

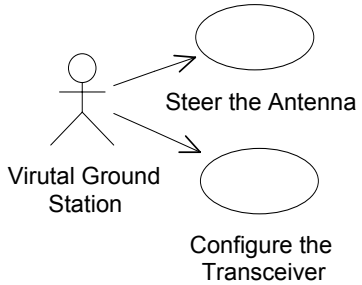

Figure 5. Layer 1 Use Cases

This subsection starts with a high-level description of the use-cases regarding services a virtual ground station must provide. Then, it describes a simplified diagram of a physical ground station. This diagram is helpful in

defining the initial classes shown at the end of this section.

Services of layer 1 are illustrated with two use-cases shown in Figure 5. It points out that a virtual ground station needs to be able to steer an antenna and configure a transceiver. Here's a description of the functionality provided by the services.

+ *Antenna Steering*: this service is used to change or obtain the direction where the antenna is pointing. Two angles define this direction: azimuth and elevation. The azimuth varies between –180 and 180 degrees. The elevation varies between 0 and 180 degrees.

+ *Transceiver Configuration*: this service is used to communicate with a Satellite. The transceiver must offer services to be turned on and off as well as services to select a communication frequency and mode.

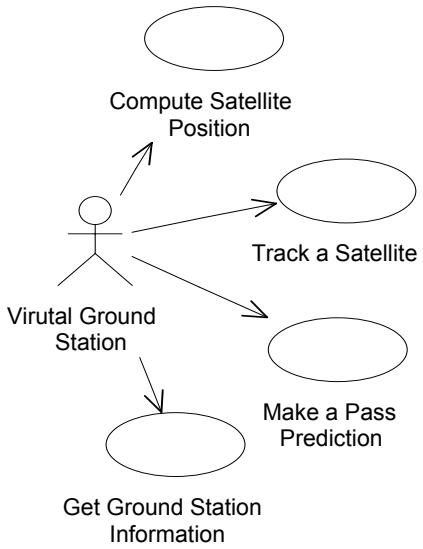

Figure 6. Layer 2 Use Cases.

Figure 6 illustrates the four use-cases of layer 2 services. It shows that a virtual ground station needs to be able to compute a satellite's position, track it, make pass predictions and provide general information on itself. Here's a description of the functionality provided by the services.

+ *Satellite Position Computing*: this service is used to obtain a satellite's position at a specific time. This position is defined by a longitude, a latitude, and an altitude relative to the center of the Earth.

+ *Satellite Tracking*: this service is used to start and stop tracking sessions. To do so, it uses the name of a satellite. The communication mode and frequency are automatically chosen. During a tracking session, it is responsible for using other services to compute the satellite's position, adjust the radio frequency to cope for Doppler shift, and steer the antenna based on the ground station location.

+ *Pass Prediction*: this service is used to create a pass prediction for a satellite and a ground station location. It can be used to determine the next pass or any pass during a specified period.

+ *Ground Station Information*: This service is used to provide the name of the owner of the ground station and to provide the location of the ground station in terms of longitude, latitude, and altitude. In addition, it provides the minimum elevation angle the antenna must positioned to before a line of sight can be established with satellites. For example, it is possible that a mountain next to the ground station blocks the first 4 degrees of elevation of the antenna. In this case, the minimum elevation is set to 4.

All of the above services are provided by the virtual ground station and must be available through the Internet. End-user applications can use these services and the WATOO GUI client is an example of such an application. Figure 7 shows how a virtual ground station is viewed from a network.

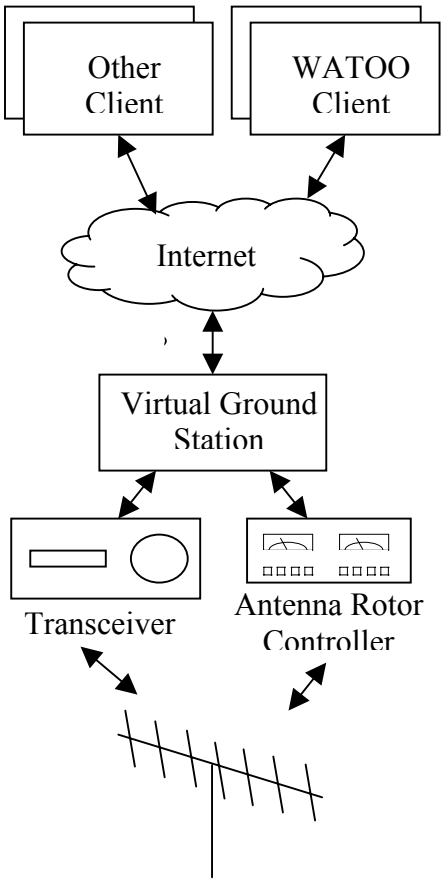

Figure 7. A Virtual Ground Station Component View.

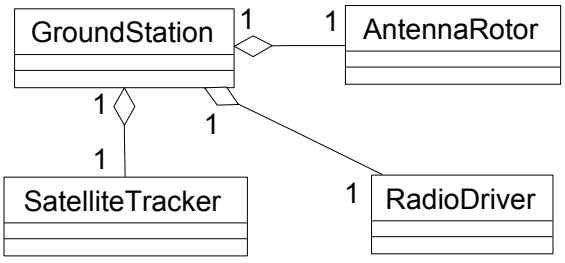

Figure 8. Initial Class Diagram of the Four Main Components.

Based on Figure 7, it is easy to derive three of the four main classes of our design. Those classes are named *Transceiver*, *AntennaRotor*, and *GroundStation*. The first two classes will provide the layer 1 services. The *GroundStation* class provides the tracking and information services of layer 2. The fourth main class is called *SatelliteTracker* and is responsible for both computing the satellite's position and for making pass predictions. Thus, the *SatelliteTracker* provides the remaining services of layer 2. The reason why they're not provided by the GroundStation is that there are many different algorithms for computing satellites position. Therefore, isolating those services in a separate class makes them easier to customize. The initial UML[6] class diagram is shown in Figure 8. This diagrams shows that the *GroundStation* is composed of one *AntennaRotor*, one *SatelliteTracker*, and one *RadioDriver*. The cardinality also indicates that the classes part of the *GroundStation* can only be part of one *GroundStation* at the time.

During the analysis phase, the main classes and their initial methods are defined. In the design phase, the definition of these main classes is refined and all utility classes are identified. To avoid repetitive diagrams, the refining process will not be described in this paper. A detailed description of all the classes is included in the first author's master's thesis[7]. This paper will only introduce the four main classes and their important utility classes.

Figure 9 presents a detailed definition of the *GroundStation* class and its utility class *Position*. As shown in the class diagram, the *GroundStation* provides two methods to start and stop tracking sessions. To start a tracking session, it is necessary to provide a *Satellite* object as well as communication modes and frequencies. Other methods provide information like the name of the ground station, its position and its minimum elevation angle. Finally, additional methods provide access to the tracking components. As shown in the class diagram, all three classes *SatelliteTracker*, *Transceiver*, and *AntennaRotor* are declared "Abstract". For the

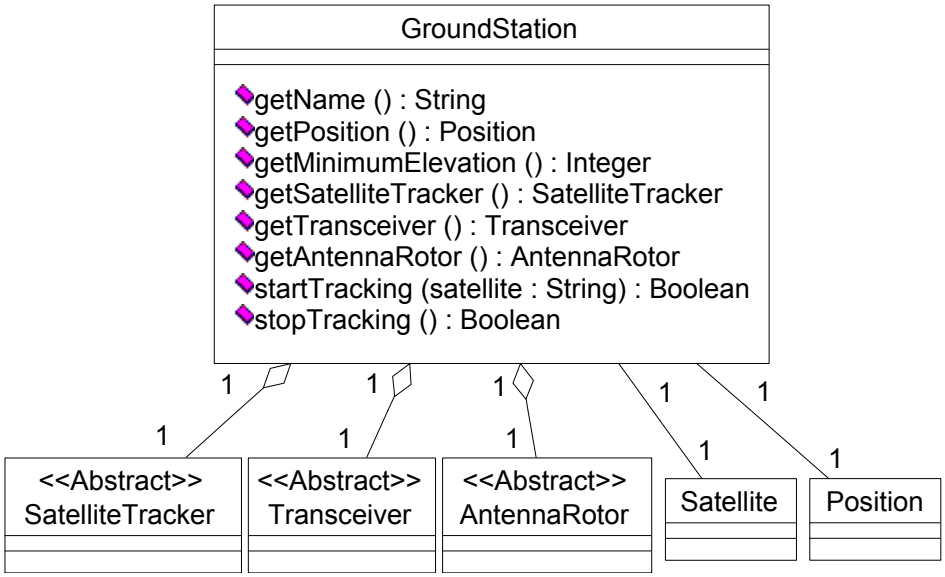

Figure 9. GroundStation Class Diagram.

GroundStation, this represents a guaranty that these classes will always have the same interface. It is also a guaranty to the developers that any of these three classes can be changed without modifying the *GroundStation* class. That is, if the customized class extends the appropriate abstract class.

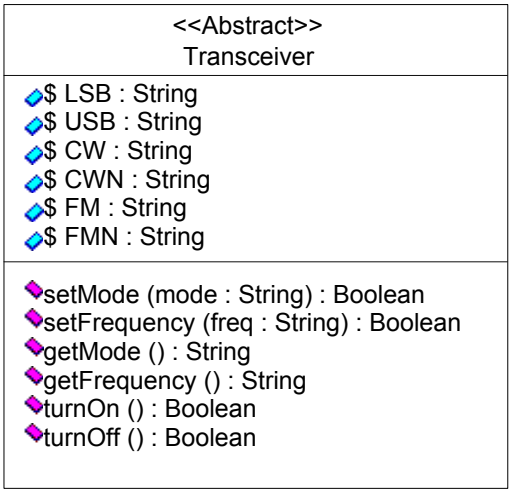

Figure 10. Tranceiver Class Definition.

The *Tranceiver* class, as shown in Figure 10, provides services to configure both frequency and the mode. The frequency is specified in MHz as a string of the following format: "xxxx.yyyyy". The mode is also specified as a string for which the following values are accepted: "LSB", "USB", "CW", "CWN", "FM", and "FMN". Predefined class attributes are provided for this purpose.

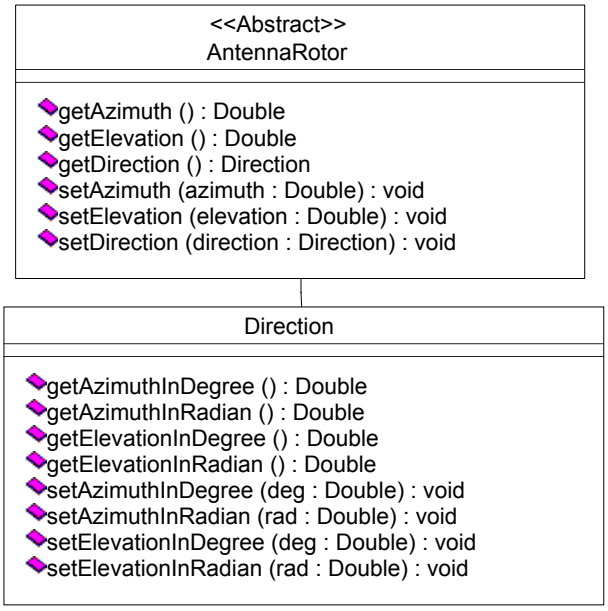

Figure 11. AntennaRotor Class Definition.

The *AntennaRotor* class is defined as shown in Figure 11. It provides services to obtain and change the antenna pointing through the use of

two angles: azimuth and elevation. Also shown in the figure is the *Direction* class which is simply a helper class to simplify the manipulation of the two angles.

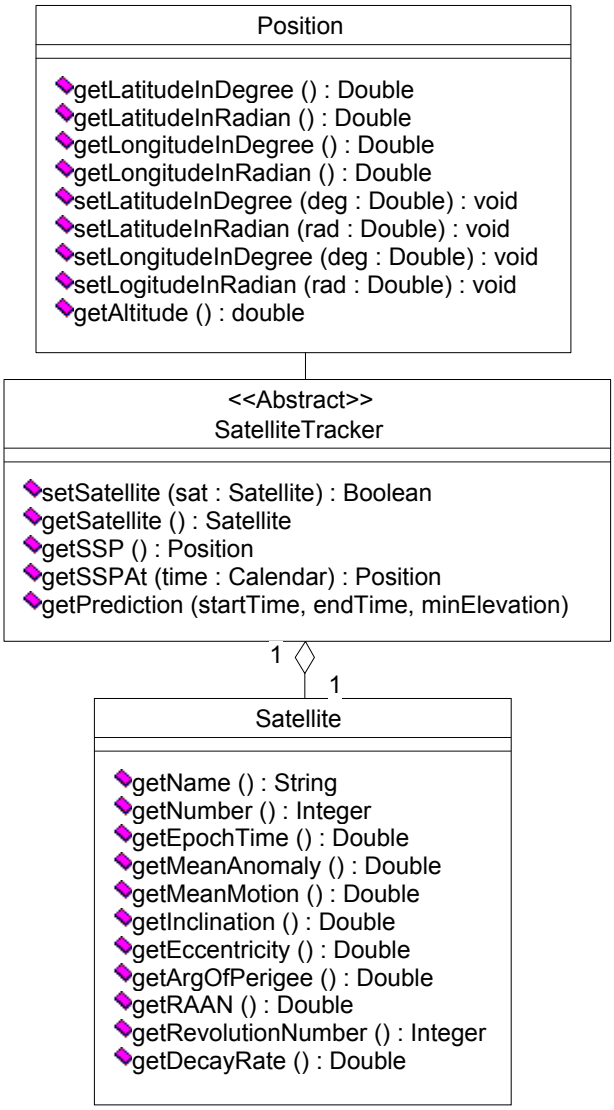

Figure 12. SatelliteTracker Class Definition.

Figure 12 shows the *SatelliteTracker* class. It is used to obtain a satellite's position based on either real time or a specific time. The satellite selection is done by calling the *setSatellite* method using a *Satellite* object. The *Satellite* class is a helper class. It is a placeholder for the Keplerian elements describing a past location of the satellite. The algorithm implemented within the *getSSPAt* method uses that information to compute the position of a satellite. The *getPrediction* method returns a pass prediction in a string table containing the rise and set time of the satellite, the antenna steering angles and Doppler shift for each minute during the pass. Finally, the *Position* class is another helper class very similar to the *Direction* class.

```
#include "SatelliteTracker.idl"
#include "Transceiver.idl"
#include "AntennaRotor.idl"
#include "Position.idl"

interface GroundStation {
string getName();
Position getPosition();
long getMinimumElevation();
SatelliteTracker getSatelliteTracker();
Tranceiver getTransceiver();
AntennaRotor getAntennaRotor();
void startTracking(in string sattellite);
void stopTracking(); };
```

Figure 13. IDL Specification for the GroundStation Component.

As it was explained in the CORBA Basics section, objects that have to be accessed remotely need an IDL interface specification. Consequently, the virtual ground station needs an IDL specification (Figure 13) since it will be used by remote end-user application such as our WATOO GUI client. Other components that need to be distributed also require an IDL specification. For conciseness, their IDL specification will not be presented in this paper.

3.2 Implementation

This subsection starts with a description of the tracking equipment of our real ground station. Then it explains how we created a virtual ground station using the application framework described earlier.

Our ground station is equipped with a transceiver, an antenna rotor controller and a personal computer. The transceiver is a Yaesu FT-736R model. The PC can control it through the serial interface. The rotor controller is also from Yaesu and can be controlled by the PC using a Kansas City Tracker (KCT) interface. The PC we used runs both Windows NT version 4 and RedHat Linux version 5.

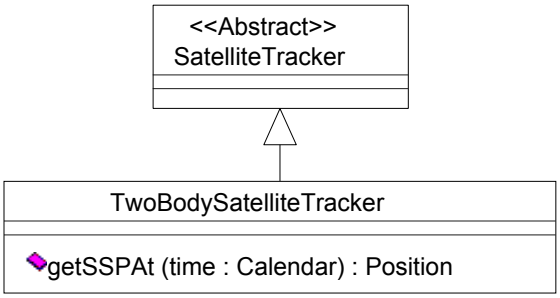

Figure 14. TwoBodySatelliteTracker Class Definition.

The first step in developing a virtual ground station is to implement the *SatelliteTracker* class. This class provides some layer 2 services. However, since all layer 2 services are not related to the tracking equipment, they are platform independent when implemented using JAVA. Consequently, once a *SatelliteTracker* is implemented, it can be used as-is for any virtual ground station. For the WATOO project, a two-body[8] satellite tracker has been implemented using JAVA. Since this *SatelliteTracker* is provided with our WATOO software, this first step isn't necessary where JAVA is available. Figure 14 illustrates the class definition of the two-body satellite tracker we developed. Not shown on the diagram are the private utility methods implemented to support *getSSPAt* which is the only method that must be implemented.

The second step is to implement the two classes that provide the layer 1 services: *Transceiver* and *AntennaRotor*. In our case, the *Tranceiver* needs to control a Yaesu FT-736R through a serial interface. The control is done using a specific protocol that defines a packet format, some commands and a byte order. The *Transceiver* we implemented is called *YaesuFT736R* and it uses JAVAComm[9] to access the serial interface. JAVAComm is a platform independent way if accessing serial interfaces. Figure 15 shows the class definition of our specialized *Transceiver* with the methods that had to be implemented.

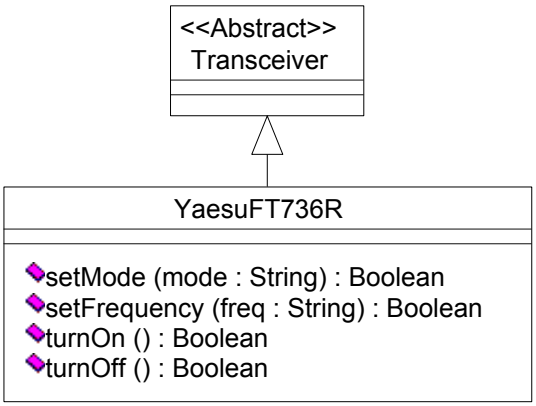

Figure 15. YaesuFT736R Class Definition.

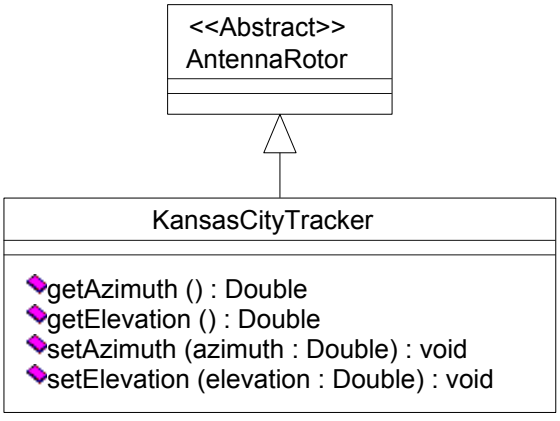

Figure 16. KansasCityTracker Class Definition.

The *AntennaRotor* we implemented is called *KansasCityTracker* as shown in Figure 16. It lists the only four methods that need to be implemented. Communication with the KCT interface is done through a memory mapped port using a bit-wise protocol. JAVA allows

direct memory access but it has to be done with the JAVA Native Interface (JNI)[10] which breaks the platform independence. Consequently, our *AntennaRotor* specialization is platform dependent as opposed to our *Transceiver*. However, we designed the *KansasCityTracker* such that most of it is written in pure JAVA. The platform dependant code is limited to only three lines of C code for Linux and approximately 30 lines for WindowsNT4.

```
groundStation.name = Sherbrooke1
groundStation.position.latitude = 45.24
groundStation.position.longitude = -71.46
groundStation.position.altitude = 0.225
groundStation.minimumElevation = 3

satelliteTracker.className=
        TwoBodySatelliteTracker

antennaRotor.className = KansasCityTracker
antennaRotor.instantiationParms = 03e0

transceiver.className = YaesuFT736R
transceiver. instantiationParms = COM2
```

Figure 17. Virtual Ground Station Configuration File.

The last step in developing a virtual ground station is to create a configuration file to be read by *GroundStation*. This file (Figure 17) supplies ground station information such as its name, its location and its minimum elevation angle. The file also dictates which *AntennaRotor* class, *Transceiver* class, and *SatelliteTracker* class the virtual ground station should use. These classes can be changed and used without recompiling the *GroundStation*. This is made possible by the JAVA's dynamic loader[10].

Once the layer 1 one classes are crated, all that needs to be done is load the *GroundStation* object. This object starts by reading the configuration file and instantiates the classes that will be providing the layer 1 and 2 services. Then, uses them during tracking sessions.

Thanks to the framework design of the virtual ground station, only two classes need to be implemented for each different station: *Transceiver* and *AntennaRotor*. In fact, because the *Transceiver* class we implemented (ie: *YaesuFT736R*) is platform independent, we were able to use it as-is on both Linux and Windows. Thus, when we created the windows version of our virtual ground station, all we had to do is adapt the *KansasCityTracker*. Everything else, including the *SatelliteTracker*, remained unchanged.

4. GUI Client Application

The client application also had to be redesigned because it wasn't CORBA capable. Also, the GUI was migrated from a developers testing tool to a more aesthetic end-user application. This section describes the new WATOO GUI client through different screen shots.

First, lets start with the tracking window shown in Figure 18. This window contains an equidistant projection of the earth on which the position of each selected satellites is plotted. In this example, the satellites are from the Iridium constellation. To add a satellite to the projection, its name must be typed in the upper-left corner text field. After a satellite has been added, the plotting needs to be started with the time-related buttons show in Figure 19. The slider shown on the same figure is used to make time go forward or backward. As you can see in Figure 20, the projection contains a ground station named Sherbrooke. Although the GUI client doesn't provide a way of adding more ground stations, the projection can take any number of them.

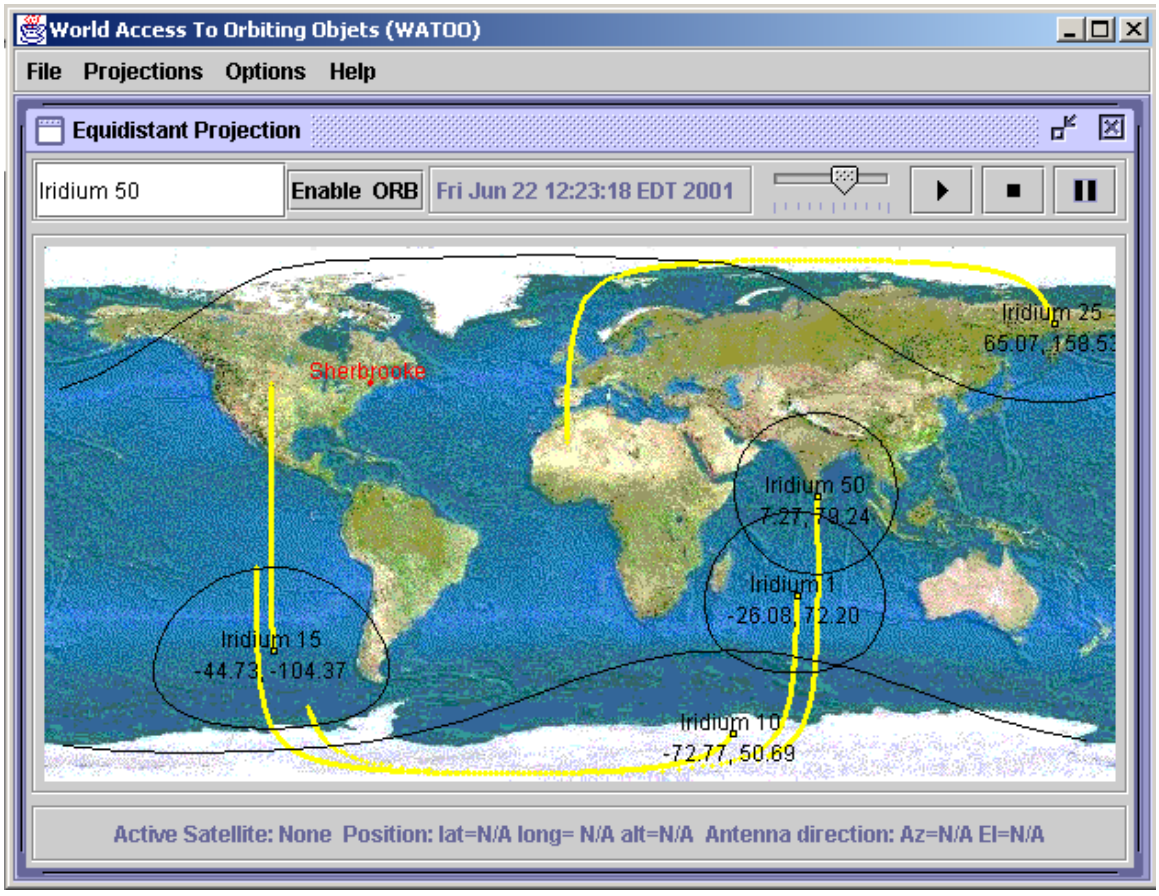

Figure 18. Equidistant Projection Window of the WATOO Tracking Application.

Figure 19. Tracking Window Time Related Controls.

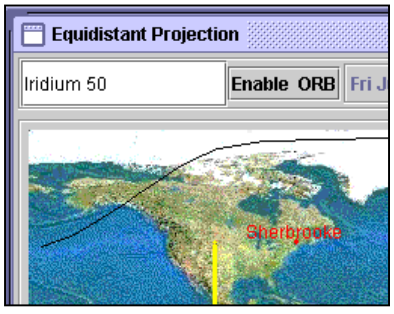

Figure 20. Sherbrooke Ground Station.

To start a tracking session, both a satellite and a ground station must be selected. Clicking on them in the projection window does this. To show that a selection has been made, the status bar at the bottom of the projection window displays tracking information. When the satellite's footprint reaches the ground station, the tracking session begins. From that moment, and until the satellites sets, the information of the status bar is displayed in a different colour.

However, until the ground station's naming server hasn't been specified, the tracking session doesn't actually occur physically. The reason for this is that the GUI client doesn't know where to get the ground station's CORBA services. Therefore, the address of a naming server must be provided as shown in

Figure 21. Clicking on the "Enable ORB" button of the tracking window opens that configuration window. One thing to mention is that the GUI client allows more than one projection window to be opened at one time. This means that it is possible to start more than one tracking session concurrently.

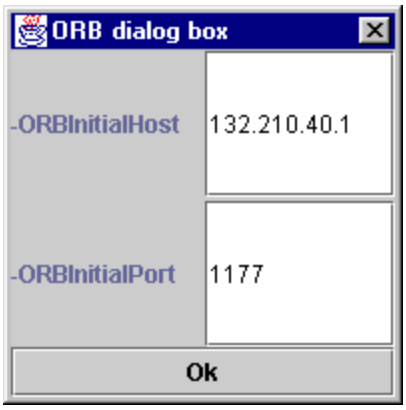

Figure 21. CORBA Naming Server Configuration

5. Conclusion

The number of LEO satellites increases each year. They offer increasingly sophisticated services that are used by scientists from all around the world. Some radio amateur satellites even provide packet communication services. The utilisation of LEO satcom services requires ground stations that are expensive and complex to employ. As a result, ground stations are quite uncommon and inaccessible which limits the use of LEO satcom services.

The contributions presented in this paper address the accessibility of satcom services through the concept of a virtual ground station connected to the Internet. We have demonstrated that this concept increases accessibility of tracking equipment since we were able to launch tracking sessions from remote cites. Furthermore, we realized during the course of our project that it would be possible to launch extended tracking sessions using many different ground stations successively. This new type of tracking session offers a great potential in terms of reducing the intermittence of satcom services.

The virtual ground station presented is designed such that it is easy to adapt to different tracking equipment. In addition, it has been implemented using technologies such as JAVA and CORBA for maximum platform independence. Only three days were required to create a windows version of our Linux virtual ground station. An example of a client tracking application capable of using a virtual ground station was presented as well. This GUI is also platform independent and illustrates how satcom services could be used through the Internet.

In summary, the results of our research can be used to address some of the problems related to LEO satcom services, which was the goal of project WATOO.

5.1 Future Work

To better support extended tracking, the GUI client could be modified in order to automate the successive selection of ground stations. In addition, the client could be improved by enabling users to save tracking session configurations, select more than one satellite to track, etc.

The virtual ground station could be enriched with new tracking equipment such as a virtual terminal node controller (TNC). This would enable the end-user application to get a hold of the digital information transmitted from a satellite.

Finally, prior to its public released, the virtual ground station would require modifications regarding security and access control. To this

end, CORBA's access control and security services would have to be investigated.

6. Acknowledgments

Special thanks to Denis Thibault for his implementation of the two-body satellite tracking algorithm.

7. References

1. Normandeau, M. and M. Barbeau, "Object-Oriented Modeling of a Satellite Tracking Software" Proceedings of the 15th ARRL and TAPR Digital Communications Conference, Seattle, September 1996.

2. Normandeau, M. and S. Bernier and J.-M. Desbiens and M. Barbeau, "WATOO: an Internet Access Software to a Satellite Tracking Session" Proceedings of the 15th AMSAT-NA Space Symposium, Toronto, October 1997.

3. Baker, M.S., "SBSat: LX200 Satellite Tracking" www.sb-software.com .

4. Schretter, S.J., "Remote Operation of a Ground Station" www.sjsi.com/w4mq .

5. Wilkinson, M., and Dr. C. Swenson, "Design of a Satellite Tracking Station for Remote Operation and Multi-User Observation" Proceedings of the 13th Annual AIAA/USU Conference on Small Satellites, 1999.

6. Muller, P.-A., Instant UML, WROX Press Inc., 1997.

7. Bernier, S., "Conception et Implantation Basées sur des Composants Répartis d'une Station Terrestre Virtuelle de Communication Satellite" Master's thesis, Université de Sherbrooke, 2000.

8. Davidoff, M., The Satellite Experimenters handbook, The American Radio Relay Ligue Press, 1994.

9. JAVAComm, Sun Microsystems Inc., java.sun.com/products/javacomm .

10. Topley, K., Core Java Foundation Classes, Prentice Hall, 1998.