Combining Ada 95, Java Byte Code, and the Distributed Systems Annex

Brad Balfour
Objective Interface Systems, Inc.
1892 Preston White Drive • Reston VA 20191
brad.balfour@ois.com • 703/295-6533 (voice) • 703/295-6501 (fax)

Abstract

This paper describes a prototype client/server system which combines two technologies: the ability to compile Ada 95 code into Java Byte Code (JBC) running on the Java Virtual Machine via Intermetrics AppletMagic™ product and the Ada 95 Distributed Systems Annex. The paper sets out the goals of the prototype effort and then illustrates the concepts, design, and techniques used to create the system. Finally, the trade-offs, alternatives, benefits and conclusions from the prototype effort are presented.

Overview

Recently, two new and interesting — but separate — technologies have emerged in the Ada community. The first is the ability to compile Ada 95 code into Java Byte Code (JBC) running on the Java Virtual Machine via Intermetrics AppletMagic product. This brings Ada developers the many benefits of Internet based client/server applications. The second technology is the Ada 95 Distributed Systems Annex (Annex E). This also allows the creation of client/server software, but is even more flexible in allowing many different configurations of distributed software. As of Spring 1997, these technologies had not yet been combined nor even shown to be able to work together.

The AJPO sponsored* the author's previous company to produce a software prototype which combined Ada 95, the Java Virtual Machine (JVM), and the Distributed Systems Annex (DSA) in one demonstration. In addition, the AJPO directed that the software prototype be representative of the type of applications which will be a part of the forthcoming Defense Information Infrastructure (DII) and its Common Operating Environment (COE).

This paper will illustrate the concepts, techniques and benefits which arise from the successful combination of Ada 95, Java Byte Code and the Distributed Systems Annex.

The Prototype's Goals

The use of Java Byte Code technology inspired the project to strive for three goals:

- *Demonstrate Client Neutrality:* The client software must run (without change) on Sun/Solaris, Macintosh/System 7.5.5, PC/Windows NT 4.0, and PC/Windows 95 environments.
- *Ensure that no physical distribution and/or installation of client software is needed:* The demonstration application must run on any machine preconfigured with a web browser. There will be no need to supply users with a disk containing client software and, therefore, no user run installation of client software.
- *Provide instant update of clients:* If the demonstration application is updated from v1.0 to v1.1, then all clients must automatically run the latest version the next time they launch their web browser.

The use of the Distributed Systems Annex inspired the project to strive for an additional goal:

- *Ensure Scalability:* The server software must be able to be configured and run on more than one Sun system. Repartitioning the distributed server system must *not* require any modification of the Server's Ada 95 software. Achieving this goal will provide the ability to distribute the server over multiple machines to achieve corresponding performance gains *without* Ada 95 source code modifications.

Specific Tools Used in Prototype Demonstration

All new software written for the prototype was written in Ada 95. The client software was compiled using Intermetrics AppletMagic v1.38 compiler for Macintosh. The client software makes extensive use of the standard Java toolkits and libraries via the Ada 95 bindings supplied with AppletMagic. The resulting JavaByteCode (or J-code) has been run on Java Virtual Machine (JVM) clients on a Macintosh (using four different JVM implementations), a PC running Windows 95 and Windows NT, and on a Sun**. The server software was compiled with ACT's GNAT v3.09 for Sparc/Solaris 2.5.1 and GLADE v1.01 (the GNAT implementation of the Ada 95 Distributed Systems Annex). The server software has been run on a series of Sparcstation 4 and Sparcstation 2 machines under Solaris 2.5.1.

Specific Software Requirements

Basic Concept

The prototype client/server pair represents a hypothetical Command, Control, Communications, Computers and Intelligence (C4I) application which could be built on top of the DII COE using Ada 95/JBC and the DSA. For this effort, the server would simulate a series of sensors which were monitoring the locations of enemy troops on the battlefield. The client would collect the data reported by all of the sensors and display it on a map on the user's screen. The server's three (3) sensors would simulate the reporting of simplified information on enemy troop locations and troop types with which a commander might be presented during a battle. This data would be broadcast in near real-time and displayed as military icons on a simulated texture map by the client software.

Server Capabilities

The server software simulates three independent sensors all observing a common Battlefield. The sensors are:

- Ground Sensor (e.g., an Artillery Forward Observer [FO]),
- Air Sensor (e.g., an Unmanned Aerial Vehicle [UAV]), and
- Satellite Sensor.

Each of these sensors is located in a different place on/over the common Battlefield. Each sensor looks at the Battlefield and returns observations to main sensor server (which then relays them to the client software via a socket connection). Each sensor introduces

* The work described in this paper was sponsored by the Ada Joint Program Office under the CIM/SETA contract and performed by the author while an employee of CACI, Inc. — FEDERAL of Fairfax, VA.

** Observed differences in client software behavior for identical .class files appear to be due to differences in JVM implementations. The behavior of the Ada 95 client is no different than a Java client — both run identically on different machines within the known differences/bugs present on those JVM implementations.

map window, disconnect from the server, and return to the login screen. The server software remains active in the background and awaits additional client connections.

Overall Architecture

This section describes the software and hardware architecture for both the client and server pieces. In addition, the flow of information among the client(s) and distributed servers is illustrated for a typical user scenario (or use case).

The client(s) and server are connected via a TCP/IP network (e.g., the Internet). The client machine is running some implementation of the Java Virtual Machine (JVM) (e.g., a Java enabled web browser). Once the Sensor Server has been launched, one or more clients may connect to it. Both the HTML server (providing the web page and the downloadable JBC client applet) and the Sensor Server application *must* be hosted on the same machine*.

Figure 3 shows the overall architecture for both the client and server machines and a typical use-case interaction. Details on this architecture are provided in the next two sections. The remainder of this section provides an architectural description of the components base on the most typical Use Case.

*This is due to a "limitation" of the security software which is part of the Java Virtual Machine. A web-based applet (client) is only allowed to open up a network connection to the same host machine from which the applet was downloaded (i.e., the html server).

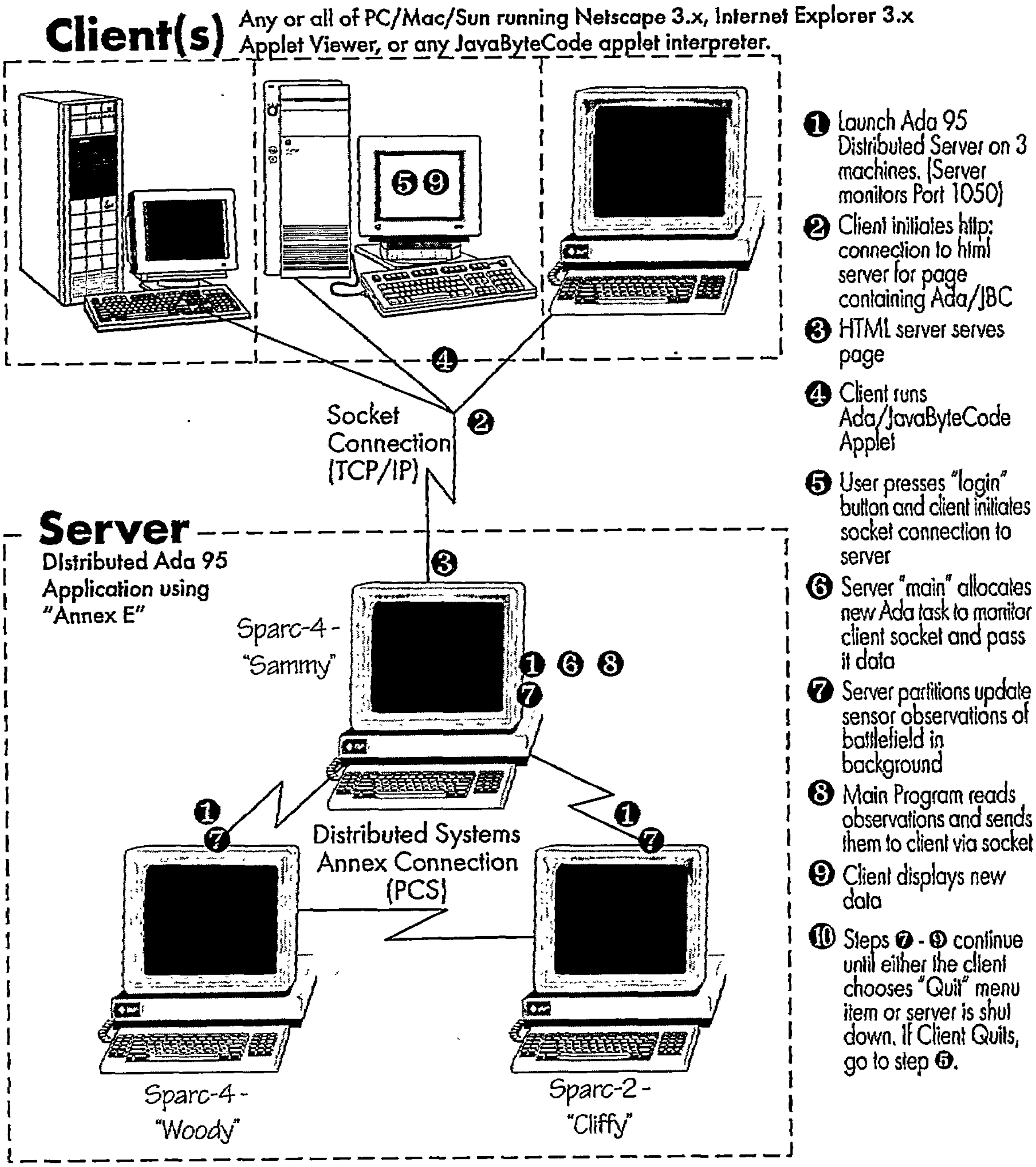

Figure 3: Overall Client/Server Architecture

Upon launch, the Sensor Server main program begins to monitor socket port 1050 for a client connection. At the same time, the separate server partitions for the Sensors and the Battlefield begin to monitor the battlefield and record troop positions.

The client connects to the web server html page that contains the Sensor Demo applet. The html server downloads the applet to the client machine as part of the web page. The Ada 95/JBC client run and then initially displays the screen that was shown in Figure 1.

When the user presses the "login" button, then the client software to initiate a TCP/IP Socket connection to port 1050 onto the Server. Upon detecting a new connection from a client, the server spawns off a new Ada task dedicated to polling each Sensor Partition for its readings and sending sensor updates to the tasks client. In parallel, the Sensor Server main program then awaits either a new client connection or control-C from the server console. The client software displays a blank map in a new window and then begins to monitor socket port 1050 for new sensor data to read. As the server sends new data, the client software displays updated icons in the new map window on the client machine. An example of this map was shown in Figure 2.

The user may select "Quit" from the "File" menu at any time in order to close the map window and return to the login screen. The user quits the browser session to terminate all client/server interaction. The server software continues to await additional client connections. Once the server receives a control-C, it is interrupted and performs an orderly shutdown of all distributed partitions.

Several elements of this architecture are dictated by the use of the Java Virtual Machine. These include the co-location of the HTML server and the sensor server, the use of a socket connection back to the server machine, and the use of a new window on the client to contain the map and the menu bar (only JVM Frames may have menus). The automatic startup and shutdown of the server software and all associated partitions is enabled by functionality provided by the Glade implementation of the DSA. Most other partitioning decisions enumerated in this section and the following two sections were design decisions by the author.

Client Design

The client software consists of nine (9) new Ada 95 packages. These packages make use of many pieces of the standard Java Library. The basic architecture for the client software is shown in the dependency diagram in Figure 4.

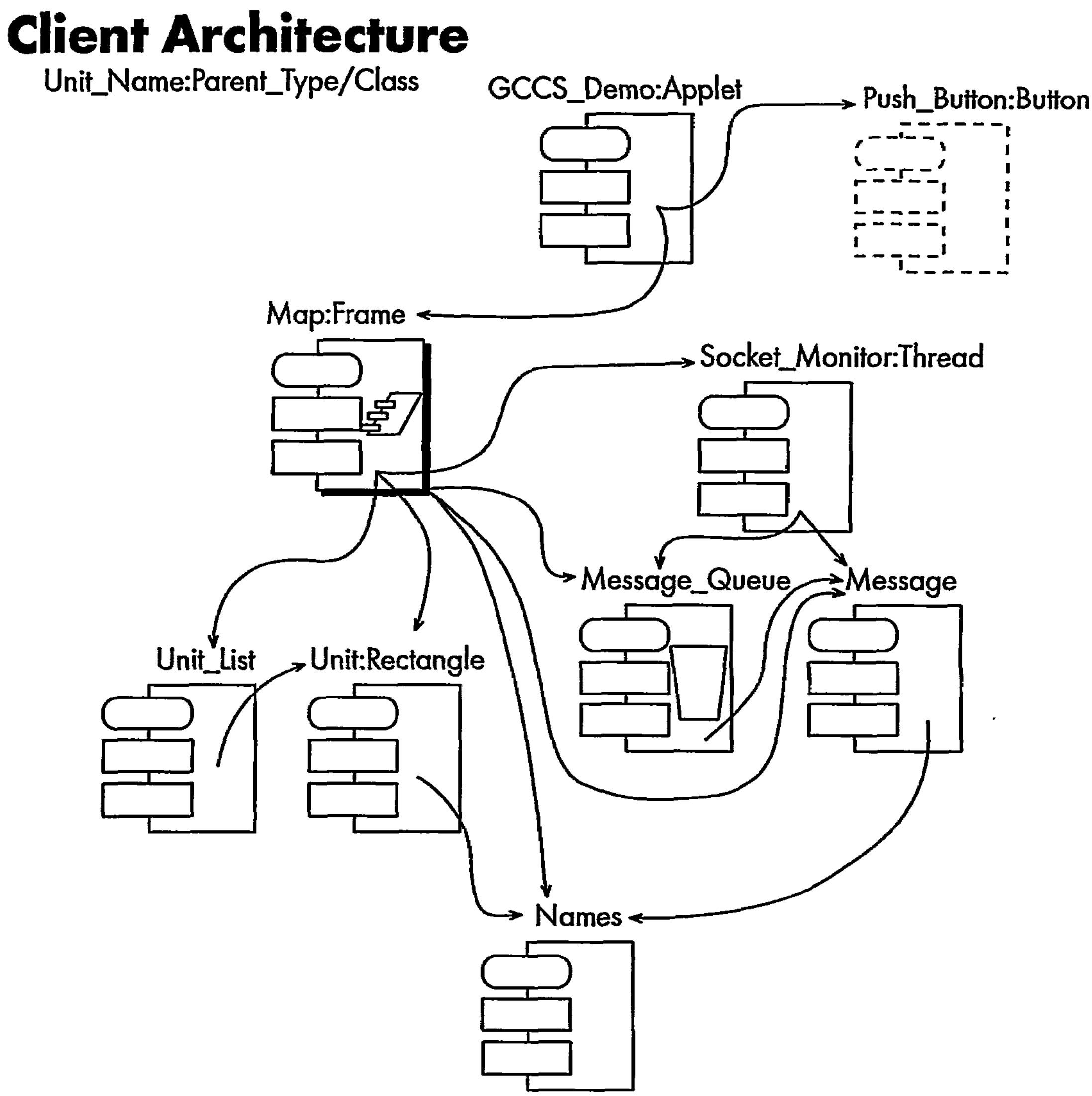

Figure 4: Client Dependency Architecture

Inheritance Relationships

Five of the new types defined in the client program inherit from standard Java types:

- ◆ The GCCS_Demo class (the "Client main program") inherits from Applet (which enables it to be embedded in a web page).
- ◆ The Map class inherits from Frame (which provides a separate window and menu bar).
- ◆ The Push_Button class inherits from Button (which provides a call-back on a GUI button).
- ◆ The Unit class inherits from Rectangle because a unit adds Armed Forces information to a GUI concept of a rectangle class that has a location and a size. This design allows easy determination of when the user has clicked within a unit or when units overlap on the screen as this functionality is provided by the Java APIs for Rectangle.
- ◆ The Socket_Monitor class inherits from Thread (which provides a separate thread that runs in parallel to the main class) and reads data from the TCP/IP socket. AppletMagic v1.38 did *not* implement Ada tasking semantics. If it had, these could have been used. Instead, the design falls back on Java's thread semantics which are provided. This is less intuitive for the experienced Ada programmer, but it helps the program fit more directly with the use of JavaByteCode and the Java APIs.

The other packages do not inherit from pre-existing Java structures, but represent other design abstractions. In this small example none of the newly defined abstractions inherit from each other.

Aggregation Relationships

The GCCS_Demo class is composed of instances of the Map and Push_Button classes as well as instances of several predefined Java AWT GUI classes. The Map class is composed of instances of the Socket_Monitor, Message_Queue and Unit_List classes. Message_Queue and Unit_List are aggregates of the Message and Unit classes respectively.

Table 1 contains a brief description of each of the Ada packages.

Working with the Java APIs: The Learning Curve

When writing Java applets in Ada 95 using AppletMagic, the effort of learning Java's syntax is reduced but not eliminated. Unfortunately, in order to follow any of the examples in many books or articles, one must know enough of Java's syntax and semantics to read and understand them. Since Java's semantics are very similar to Ada 95's, this is not difficult. Rather, when learning Java, the biggest hurdle is that of mastering the vast APIs that are part of Java. In this sample application, the design draws heavily upon the network APIs (java.net) and the GUI APIs (java.awt). Due to the complexity of these APIs, there was a substantial learning curve involved. Several sets of Java books and on-line tutorials were consulted. In fact, approximately seven incremental iterations of the design and implementation were performed in order to master the complexity of the APIs and building this type of applet for the first time. The final applet, though, is very much a Java applet — done in Ada 95 syntax and semantics. It makes full use of the Java APIs and, except for the Algol/Pascal syntax, resembles most other applets.

Working with the Java APIs: Advantages and Disadvantages

In general, the advantages of using the Java APIs far outweighed the single biggest disadvantage: the complex learning curve. The advantages include: a rich set of functionality to choose from; plentiful examples in books and magazines; and a uniform GUI look and feel on all platforms from a single, standard, set of source code. An additional minor disadvantage was the fact that the implementa-

GCCS_Demo	Contains the applet and housekeeping code. It starts the applet and presents the user with a login screen. It makes use of Push_Button to invoke the right action when the user pushes the "Login" button.
Push_Button	A generic package that provides a subclass of the Java Button class and an associated action routine that is called when the button is pushed. In the Sensor Demo, the action is to create a new map frame and display it. Map and its associated thread take over from there.
Map	Brings up a separate window (a Java Frame) along with a menu bar and menu. Map contains an instance of the Socket_Monitor class which is responsible for getting data from the server. Map contains an additional thread that depends on the Message_Queue class and reads each new message from the queue. Based on the message, the thread then creates new units and adds them to the Unit_List. The Paint() method then makes use of the Unit_List and Unit Paint() methods to display these new units.
Socket_Monitor	Responsible for getting data from the server. Is a subclass of Thread so it operates in parallel with other threads in the applet. Each new line of data read from the socket is converted to a message via the Message class's constructor. The new message is then added to the Message_Queue.
Message	Parses the raw text string from the server into its component information.
Message_Queue	A FIFO list of Messages that have been retrieved from the server. It is "synchronized" because it is a shared data structure which is access by both the Socket_Monitor and Map "threads".
Unit	A single observation from a single sensor. It has a location on the map and a unit kind.
Unit_List	An array of three Singly-Linked Lists of Units. Each list corresponds to a different sensor and its unit observations.
Name	A pair of enumerated types listing the Sensor names and the kinds of Enemy units.

Table 1: Description of each Client application package

tions are not yet implemented in a uniform and bug free way. Although the code does not need to change in order to produce a portable GUI, the resulting application is unlikely to be 100% identical among platforms. Many JVM GUI bugs are well documented on the internet and are of a minor nature.

The Use of Java APIs and Concepts in the Client Design

In order to produce a JavaByteCode based applet which would meet the goals listed at the beginning of this paper, it was necessary to make use of several of the features of Java. Foremost among these is Java's platform independent GUI toolkit: AWT (Abstract Windowing Toolkit). AppletMagic provides a full set of Ada interfaces to these predefined routines. It is typical, as seen in Figure 4, to design ones application by inheriting from these predefined classes. In addition, it was a critical requirement that the Client Applet be able to simultaneously display sensor results, accept user input and receive new results from the server. This necessitated the use of multi-threading in the Applet. Ada 95 tasks and protected types could have been used to implement this requirement. Unfortunately, version 1.38 of AppletMagic did not yet provide support for these features. Java does provide a very similar multithreading capability, and it was available through the use of standard Java APIs and AppletMagic supported pragmas. Although the final design is not the same as if Ada tasking were used, it is very close — proof of the similarity between Ada's semantics and those of Java.

Creating Client Applets in Ada 95

Overall, some things about creating a client applet are made simpler by the use of Ada 95. As has been mentioned, the learning curve is simpler because of the use of Ada. However there are also difficulties in using Ada 95 for an applet. For example, a translation must be made when using most common references, examples or books. Certainly this is mitigated by the excellent examples supplied by AppletMagic, but they don't replace a book or article. In the same way, it is difficult to ask questions in forums such as comp.lang.java.programmer without first translating one's question and example code into Java syntax and/or translating a response. This is offset by the willingness of the AppletMagic development team — especially lead developer Tucker Taft — to directly answer questions from users. Certainly, this design effort shows that it is feasible and practical to create thin client applets using Ada 95.

Server Architecture

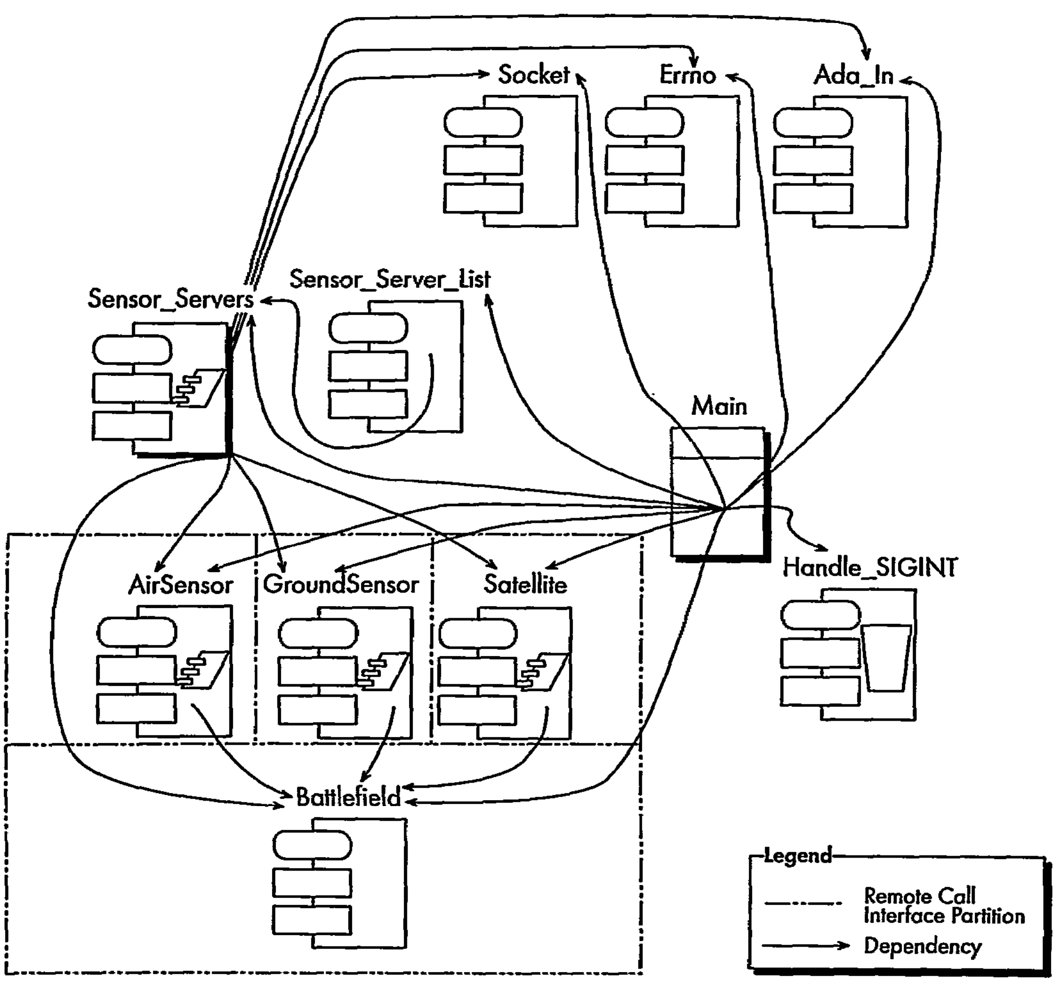

Figure 5: Server Dependency Architecture

Server Design

The server software consists of eleven (11) Ada 95 packages. The basic architecture for the server software is shown in the dependency diagram in Figure 5.

There are five separate distributed components in the server software:

- **The "Main" Program:** Package Main and the Sensor_Server_List and Sensor_Server components are the main pieces in the Data Server partition. This partition initiates communication with the other partitions but is never called on by any others.
- **The Air Sensor:** A Remote Call Interface partition. This partition responds to requests for the Air Sensor's observations. It also contains a separate task whose thread simulates the sensors — it updates readings from the Battlefield Partition on a continuous basis.
- **The Ground Sensor:** A Remote Call Interface partition. This partition responds to requests for the Ground Sensor's observations. It also contains a separate task whose thread simulates the sensors — it updates readings from the Battlefield Partition on a continuous basis.
- **The Satellite Sensor:** A Remote Call Interface partition. This partition responds to requests for the Satellite Sensor's observations. It also contains a separate task whose thread simulates the sensors — it updates readings from the Battlefield Partition on a continuous basis.
- **The Battlefield:** A Remote Call Interface partition. This partition contains the locations of all enemy units. It responds to requests for the list of locations. Currently, units on the battlefield are static.

Table 2 contains a brief description of each of the Ada packages.

Communicating Between Client and Server

In this prototype system, simplicity was the primary driver in choosing the mechanism to connect the client and server pieces. Although both parts are written in Ada 95, the client compiles into JavaByteCode running on any client machine with a JVM or Web Browser. The server runs on one or more Sun workstations. Therefore, some network communication mechanism must be chosen. Several alternatives exist: a socket connection, Remote Procedure Calls (RPCs), Common Object Request Broker Architecture (CORBA), and a heterogeneous implementation of the DSA (see the section on Alternative Architectures for the trade-offs). Of all of these, only a simple socket connection had been proven in a similar context. While simplicity is a chief advantage of a socket connection, it's low level nature brings along some disadvantages. The socket connection is a *narrow* interface — it can only communicate a simple character or binary based data stream. It is up to the developer(s) of the client and server to determine the messaging protocol (semantics) the are to be used and to convert any and all data to be sent to the low level format supported ("marshalling the data"). For this simple prototype system, this kind of communication worked well. However, it does not scale well to larger systems.

Creating Servers Using Ada 95's Distributed Systems Annex (DSA)

It would certainly have been possible to create the Ada 95 server as a single Ada 95 program containing multi-threaded tasks representing each Sensor. In fact, that design is *not* very different from the actual design shown in *Figure 5*. However, that design does not exactly model the real-world simulation that was desired in this prototype. The goal was to implement a system where each Sensor was located on a different computer and all of these computers communicated to exchange readings about the battlefield. Ada 95's DSA enabled the simplicity of a single program, single language, multi-threaded approach to be combined with the scalability and realism of a design that ran on multiple computers.

The actual prototype server was created as a single Ada 95 program and then made to run as a distributed system by adding only the appropriate categorization pragmas defined in the Ada 95 RM. If compilation and linking proceed normally, then the result is a single Ada 95 program. However, by making use of the post compilation GLADE tool "gnatdist"(ACT's implementation of the DSA which works with GNAT), the program can be made to run on multiple

Main	An infinite loop server program. Opens the socket connection on port 1050 and waits at the Socket.Accept() call for clients to connect. Uses the Sensor_Server_List package to keep track of tasks spawned to handle socket connections. Its code contains an Asynchronous Transfer of Control that handles the SIGINT (^C) interrupt if the user presses control-C to stop the server.
Sensor_Servers	A package containing a task type. One new task is allocated for each client socket connection. The task repeatedly gathers the most recent observation from the AirSensor, GroundSensor, and Satellite and passes them to the client via the socket connection.
Sensor_Server_List	A Singly Linked List of Sensor_Servers. Used to allow the Main to signal all Sensor_Server tasks to shutdown when the server program is interrupted/terminated.
Handle_SIGINT	A protected procedure/interrupt handler. Traps the SIGINT signal (control-C) and allows Main to conduct an orderly shutdown.
AirSensor, GroundSensor, Satellite	A Remote Call Interface package. Each provides an identical interface to return to the caller the most recent set of observations. Each also contains a task that works in the background to observe the Battlefield and update the internal list of observed enemies.
Battlefield	A Remote Call Interface package. The Battlefield contains the true list of enemy unit positions. Each sensor receives this true data and adds its own sensor error adjustments to make a sensor observation.
Socket, Ada_In, Errno	Bindings to the Unix Socket, Socket Address, and Error Number facilities.

Table 2: Description of each Server application package

computers via the DSA. No further source code changes are needed. In the design shown in *Figure 5*, there is one active partition (the main partition) and four Remote Interface Partitions (one for each sensor and one for the battlefield). This allows the resulting program to run on anywhere from one (1) to five (5) different computers *without recompilation or relinking*. The distribution of partitions to computing nodes is strictly a post compliation process.

Workarounds: Creating Better Designs Accidentally

Due to a bug in GNAT v3.09, the prototype application would receive an incorrect SIGPIPE Unix signal whenever a client socket connection closed. ACT provided a workaound for this problem in the form of a protected object which associated a protected procedure as the interrupt handler for the SIGPIPE signal.

This code succinctly illustrated how to trap and react to a Unix signal. As a result, a similar protected type was designed to handle SIGINT, the signal generated when the user interrupts the running program with a control-c (^c) keystroke (*see Listing 1*). This protected type is then used by the server main program in conjunction with an Asynchronous Transfer of Control:

```
begin
  -- startup/initialization code
 select
  Handle_SIGINT.SIGINT_Handler.Interrupted;
  -- got ^c, now shutdown
 then abort
  loop
    -- normal server processing
  end loop;
 end select;
end;
```

The main program starts up and then enters an infinite loop within the ATC. While in that loop it handles client connections and serves sensor data. Upon receiving a ^C generated SIGINT signal, the select part of the ATC is activated and causes the abort of the normal sensor loop. This code then shuts down all partitions and terminates the server. The result is a simple, clean mechanism to have a server which runs in the background until interrupted.

```
package Handle_SIGINT is
  pragma Elaborate_Body;
  protected SIGINT_Handler is
    entry Interrupted; -- wait for SIGINT (^C)
    procedure Signal;  -- handle SIGINT & set flag
     pragma Interrupt_Handler (Signal);
  private
    Interrupt_Received : Boolean := False;
  end SIGINT_Handler;
end Handle_SIGINT;

with Ada.Interrupts.Names;
package body Handle_SIGINT is
  protected body SIGINT_Handler is
    entry Interrupted when Interrupt_Received is
    begin
      null;  -- release caller
    end Interrupted;
    procedure Signal is
    begin
      Interrupt_Received := True;
    end Signal;
  end SIGINT_Handler;
begin
  Ada.Interrupts.Attach_Handler
   (SIGINT_Handler.Signal'Access, Ada.Interrupts.Names.SIGINT);
end Handle_SIGINT;
```

Listing 1: Handling SIGINT

Alternative Architectures

Sockets – A Narrow Interface

The design presented in this prototype application uses standard sockets to communicate between the client and server. These are supported in a very similar fashion by the Solaris operating system used on the Server and by the Java.Net package supplied with the JVM.

As previously mentioned, this provides only a narrow pipeline between the client and server. The API consists of little more than two operations — one to read bytes of data and one to write bytes of data. All information in the application must be converted from Ada datatypes to characters/bytes. All actions to be communicated from the client to the server must be changed from procedure call oriented actions into message oriented events and back again. This places much of the responsibility for the infrastructure of a distributed client/server application onto the programmer. Two higher level alternatives exist, however neither of these was employed in this prototype application due to a lack of time, resources, and the fact that neither alternative had yet been tried.

CORBA – A Wider Option

An approach which reflects the high level software design more directly is that of CORBA. Via the use of IDL (Interface Definition Language), an object-oriented API between the client and server can be defined. This API consists of the operations, data arguments and exceptions that represent each interface (or class). The IDL interface is then mapped into the implementation language for the client and server (e.g., Ada 95). Both the client and server developers work as if they are writing code that makes use of local packages. Underneath, these packages define stubs and skeletons that marshal the data, communicate it across the network (similar to RPCs) and unmarshal the data on the server side. This hides all of the communications detail under the simple API.

This interface is wider than that of sockets because it allows the expression of a complex API as the direct communication path between client and server. As far as the programmer is concerned, the server is just another package in the local client program — even though the server actually runs on a remote machine across the network. This raises the level of abstraction between client and server to a much higher level and provides many benefits over direct socket programming.

For this prototype application, the only current drawback to the use of CORBA would be the fact that the client is running on a JVM with code written in Ada 95. CORBA ORBs exist for both Ada 95 code applications and for the Java JVM. However, the code targeted to the JVM, which would be created from the IDL, would likely be Java source code. Therefore, extra steps would be needed to (a) compile the generated code, and (b) make use of Intermetrics auxiliary tool to create an Ada interface to the generated Java code.

RMI to DSA – Another Wide Option

Just as the prototype was nearing completion, Texas A&M University (TAMU) announced ADEPT/JxA, an upgrade to ADEPT that would connect the Java Remote Method Invocation (RMI) and Ada's Distributed Systems Annex. RMI is Java 1.1's technology for creating Java to Java distributed applications. Using this technology, one could hypothetically have created the Applet Client using Ada 95 code which made use of RMI. This code would then be connected to the server code using the DSA via TAMU's JxAgent. So far as the author knows, no one has yet attempted this connection.

Tool Usage and Results

Both AppletMagic and Glade worked well in this prototype application. Although a small number of bugs were present in both tools, these bugs were easily worked around. Both Intermetrics and ACT were highly responsive to questions and bug reports and their assistance enabled the prototype effort to go smoothly. Based on the experiences in this small prototype effort, the author would recommend the use of either or both tools on a full scale development effort.

Results/Conclusions

The Sensor Client/Server prototype has successfully demonstrated that Ada 95 can be used to create distributed Client/Server applications in the same way as other technologies, based on both the Java Virtual Machine and Distributed Applications (e.g., Remote Procedure Calls). It has also demonstrated that it is possible to combine the Ada 95 DSA (Annex E) type of distributed software with Java-based Client/Server distributed software.

The construction on this prototype system proved that Ada 95 could be used to successfully create both a client and a server which combine the best features of both the Java Virtual Machine and the Distributed Systems Annex. It was also demonstrated that it is possible to use these technologies to produce a realistic client/server system which simulates a simplified C4I sensor display application.

The prototype has successfully shown several of the advantages of the Java Virtual Machine for any large client/server environment:

- *Client Neutrality*: The client software runs (without change) on Sun/Solaris, Macintosh/System 7.5.5, PC/Windows NT 4.0, and PC/Windows 95 environments.
- *No physical distribution and/or installation of client software necessary*: In order to run the prototype, the user needs only to have a machine configured with a web browser.
- The goal of illustrating the automatic download of a new version of the software was not met during the time of the prototype. There was not enough time in the project to modify the software after the deployment of version 1.0. However, the goal was partially demonstrated since new client versions were constantly deployed as incremental prototype versions were built. Therefore, the author has confidence that a subsequent system would easily demonstrate this goal.

The prototype has also successfully shown a strong advantage of the Ada 95 Distributed Systems Annex approach:

- *Scalability*: The server software is able to be configured and run on anywhere from 1 to 5 Sun systems *without changes to the Ada source code and without recompilation of the software.* (Repartitioning the software only required a simple edit of the .cfg file and rerunning "gnatdist".)

With regards to the Ada community, the existence of this prototype serves as a proof-of-concept that Ada 95 software can be used in contexts where developers might naturally think of the use of Java. For an experienced Ada 95 developer who is *not* familiar with Java, the lower learning curve might well prove to be an advantage. Additionally, the demonstration of the ability to combine a JBC applet with a DSA server drives the state of Ada based client/server development forward another notch.

Appendix A – Client Software Source Code Listings

```
--------------------------------------------------------------------------------
--  * AJPO GCCS Demo
--  *
--  * Copyright (c) 1997 CACI, Inc.
--  *
--  * Ada structure derived from TextScroller Applet by Bill Pritchett whose
--  * Ada structure was derived from LifeRect.ada by Tucker Taft and also from
--  * BigCalc by Vince Del Vecchio of Intermetrics, Inc.
--  * Copyright (c) 1995 Intermetrics, Inc.
--  *
--  *
--  * This program is free software; you can redistribute it and/or modify
--  * it under the terms of the GNU General Public License as published by
--  * the Free Software Foundation; either version 2 of the License, or
--  * (at your option) any later version
--  *
--  * This program is distributed in the hope that it will be useful,
--  * but WITHOUT ANY WARRANTY; without even the implied warranty of
--  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  * GNU General Public License for more details.
--  *
--------------------------------------------------------------------------------
```

```
with java.applet.Applet;   use java.applet.Applet;
with Interfaces.java;      use Interfaces.java;
with java.lang.String;     use java.lang.String;
with java.awt.Component;   use java.awt.Component;
with java.awt.Container;   use java.awt.Container;
with java.awt.Image;       use java.awt.Image;
with Map;                  use Map;
package GCCS_Demo is

  type GCCS_Demo_Obj is new Applet_Obj with private;
  type GCCS_Demo_Ptr is access all GCCS_Demo_Obj;

  procedure main(Argv : String_Array);
    -- called as entry point to applet

  procedure init(This : access GCCS_Demo_Obj);
    -- called before start for one time initialization
private
  type GCCS_Demo_Obj is new Applet_Obj with record
    Login_Button   : Component_Ptr ; --button to connect to verify user
    Name_Field     : Component_Ptr ; --user name
    Password_Field : Component_Ptr ; --user password
    Img            : Image_ptr ;     --the map
    The_Map_Frame  : Map_Ptr ;       --the window to display the map
    host           : String_Ptr  := +"192.190.177.181";
                                     -- the default address to Sammy
  end record;
end GCCS_Demo;

--------------------------------------------------------------------------------
-- Body of GCCS_Demo
--------------------------------------------------------------------------------
with java.io.PrintStream;    use java.io.PrintStream;
with java.lang.System;       use java.lang.System;
with java.lang.Integer;      use java.lang.Integer;
with java.awt.Button;        use java.awt.Button;
with java.awt.Label;         use java.awt.Label;
with java.awt.TextField:     use java.awt.TextField;
with java.awt.GridLayout;    use java.awt.GridLayout;
with java.awt.LayoutManager; use java.awt.LayoutManager;
with java.net.URL; use java.net.URL;  -- for codebase
with Push_Button;  -- generic push button
with Ada.Characters.Latin_1;
package body GCCS_Demo is
-- instantiate the generic button & set up its call back

    type PressedButtonInfo is record
      Parent : GCCS_Demo_ptr; -- what applet is button inside of?
    end record;

    procedure ConnectButtonPress(info : PressedButtonInfo);
      package ConnectButton is new push_button
                                       (PressedButtonInfo, ConnectButtonPress);
      use ConnectButton;
         procedure ConnectButtonPress(info: PressedButtonInfo) is
           -- make a frame or just show already made frame
         begin
                   println(stdout, +"Button was pressed!");
                   println(stdout, +"Bringing up map window");
        Info.Parent.The_Map_Frame := new_Map(+"Battlefield Map", Info.Parent.Img,
                                             Info.Parent.Host);
          -- a side effect of creating a new map frame is to also
          -- launch two threads in that frame: (1) to read the socket
          -- and (2) to read messages from the queue and modify the unit list
        setResizable(Info.Parent.The_Map_Frame, False);
        resize(Info.Parent.The_Map_Frame, 880, 656);
          -- should really wait for the image to be ready before the show, but...
        show(Info.Parent.The_Map_Frame);
                 println(stdout, +"Map window up.");
         end ConnectButtonPress;
--------------------------------------------------------------------------------
-- Initialize the applet.
--------------------------------------------------------------------------------
  procedure init(This : access GCCS_Demo_Obj) is
    HC_Parameter : String_Ptr;
    LB_Info : PressedButtonInfo := (Parent => GCCS_Demo_ptr(this));
    N_Field : TextField_Ptr := new_TextField(8);
    P_Field : TextField_Ptr := new_TextField(8);
    A_Label : Component_Ptr;
    RIGHT   : Integer renames Java.AWT.Label.Right;
  begin
    setLayout(this, new_GridLayout(rows=>3, cols=>2,
                             hgap=>10,vgap=>3).LayoutManager'access);
      -- add username and password fields to applet
    A_Label := Add(container_ptr(this), Component_Ptr(new_Label(+"Name", RIGHT)));
    This.Name_Field     := Add(container_ptr(this), Component_Ptr(N_Field));
    Show(This.Name_Field);
    A_Label := Add(container_ptr(this),
                   Component_Ptr(new_Label(+"Password", RIGHT)));
    SetEchoCharacter(P_Field, '*');
    This.Password_Field := Add(container_ptr(this), Component_Ptr(P_Field));
         Show(This.Password_Field);
     -- add null label for spacing in grid
    A_Label := Add(container_ptr(this), Component_Ptr(new_Label(+"", RIGHT)));
     -- add button to applet
    This.Login_Button := ConnectButton.New_push_button
                         (container_ptr(this), LB_Info, +"Login");
    Show(This.Login_Button);
    resize(This, preferredSize(container_ptr(this)));
      -- deal with the (hidden) Frame's Imaage
    This.Img := getImage(This, getCodeBase(This), +"map_gifs/map6.gif");
      --map#6 is 880x656
      --grid from sever is 110x82 each cell is 8x8 (8:1 ratio)
      -- read parameter from HTML file
    HC_Parameter := getParameter(This, +"HC");
    if HC_Parameter /= null and then not equalsIgnoreCase(HC_Parameter, +"True") then
         -- figure out which machine we came from and pass that along to the map
      This.Host := getHost(getCodeBase(This));
    else
      null; -- use the default value set in private part
    end if;
  end init;
--------------------------------------------------------------------------------
-- Main Program
--------------------------------------------------------------------------------
    procedure main(Argv : String_Array) is
      This : aliased GCCS_Demo_Obj;
    begin
      GCCS_Demo.init(This'access);
      GCCS_Demo.start(This'access);
    end main;
end GCCS_Demo;

pragma Suppress(Elaboration_Check);
with java.lang.String;   use java.lang.String;
with Interfaces.java;    use Interfaces.java;
with java.awt.Event;     use java.awt.Event;
with java.lang;          use java.lang;
with java.awt.Container; use java.awt.Container;
with java.awt.Component; use java.awt.Component;
with java.awt.Button;    use java.awt.Button;
generic
   type callbackInfo is private;
   with procedure handlepress(info : callbackInfo);
```

```
package push_button is
    type push_button_Obj is new Button_Obj with record
          cb : callbackInfo;
    end record;
    type push_button_Ptr is access all push_button_Obj'class;
    function New_push_button (Parent : Container_Ptr;
          info    : callbackInfo;
          B_Name  : String_Ptr;
                              Obj : push_button_Ptr := null)
                              return component_ptr;
    -- function "+"(S:String) return String_Ptr renames Interfaces.Java."+";
    function action (Obj : access push_button_Obj;
                          Event : Event_Ptr;
                          what_Obj : Object_Ptr)
                          return Boolean;
end push_button;

with java.lang.System;  use java.lang.System;
with java.io.PrintStream; use java.io.PrintStream;
package body push_button is
    -- creates a new push button and returns it to the parent object
    function New_push_button (Parent : Container_Ptr;
          info   : callbackInfo;
          B_Name : String_Ptr;
          Obj  : push_button_Ptr := null)
          return component_ptr is
         new_button : push_button_Ptr := new push_button_obj;
     begin
        setLabel(Button_ptr(new_button), B_Name);
        new_button.cb := info;
        return add (parent, Component_Ptr(new_button));
     end New_push_button;
      -- Handles a button push event for this object and sends it somewhere
     function action(Obj: access push_button_Obj;
                          Event : Event_Ptr;
                          What_Obj : Object_Ptr)
                          return Boolean is
     begin
          handlepress(Obj.cb);
          return true;   -- put code here to call Orbix client?
     end action;
end push_button;



-------------------------------------------------------------------------------
--  * AJPO GCCS Demo
--  * Map Window Frame
-------------------------------------------------------------------------------
with java.lang.String;     use java.lang.String;
with Interfaces.java;      use Interfaces.java;
with java.awt.Frame;       use java.awt.Frame;
with java.awt.Component;   use java.awt.Component;
with java.awt.Container;   use java.awt.Container;
with java.awt.Event;       use java.awt.Event;        -- needed for handleEvent
with java.net.URL;         use java.net.URL;
with java.awt.Graphics;    use java.awt.Graphics;     -- needed for overriding paint
with java.awt.Image;       use java.awt.Image;        -- needed for storing Map
with java.awt.MenuBar;     use java.awt.MenuBar;
with java.lang.Runnable;   use java.lang.Runnable;    -- needed to provide second
thread
with java.lang.Thread;     use java.lang.Thread;      -- needed to provide second
thread
with NYI;                  use NYI;
with Unit;                 use Unit;
with Unit_List;            use Unit_List;
with Socket_Monitor;       use Socket_Monitor;
package Map is
  type Map_Obj is new Frame_Obj with private;
    -- eventually add runable component for "implements runnable"
  type Map_Ptr is access all Map_Obj;

  function handleEvent(This : access Map_Obj;  evt : Event_Ptr) return Boolean;
    -- handle window close event

  procedure paint(This : access Map_Obj; G : Graphics_ptr);
    -- use to draw map in frame and animate icons

  procedure update(This : access Map_Obj; G : Graphics_ptr);
   -- override to avoid total redraw, use clipping regions.

  procedure run(This : access Map_Obj);
  pragma Convention(Java, run);  -- so matches Runable.Run
    -- called when thread is started
    -- Implements Runnable

  function new_Map(title              : String_Ptr;
                   The_Map            : Image_Ptr;
                   Host_Addr          : String_Ptr;
                   Obj : Map_Ptr := null)
                        return Map_Ptr;
  pragma Convention(Java_Constructor, new_Map);
private
  type Map_Obj is new Frame_Obj with record
    Our_Map    : Image_Ptr;
    Menu_Bar   : MenuBar_Ptr;
    NYI_Dialog : NYI_Ptr;
    Positions  : Unit_List_Ptr;
    SM         : Socket_Monitor_Ptr;
    -- thread related data
    Runnable   : aliased Runnable_Obj;  -- means "implements Runnable"
    The_Thread : Thread_Ptr := null;
      -- points to the thread we kicked off. If it is null, then make a new thread.
  end record;
end Map;

with java.io.PrintStream;     use java.io.PrintStream; -- needed for println
with java.lang.System;        use java.lang.System;    -- needed for stdout
with java.awt.Menu;           use java.awt.Menu;
with java.awt.MenuItem;       use java.awt.MenuItem;
with Names;                   use Names;
with Message;                 use Message;
with Message_Queue;           use Message_Queue;
pragma Elaborate_All(Message_Queue);
package body Map is
  procedure Create_Menus(The_Menu_Bar : MenuBar_Ptr) is
    File_Menu : Menu_Ptr := new_Menu(+"File");
  begin
    File_Menu := Add(The_Menu_Bar, File_Menu);
    Add(File_Menu, +"Observe Sensors");
    Add(File_Menu, +"Quit");
  end Create_Menus;
  function new_Map(title     : String_Ptr;
                   The_Map   : Image_Ptr;
                   Host_Addr : String_Ptr;
                   Obj       : Map_Ptr := null)
          return Map_Ptr is
          -- constructor operation
    New_Map : Map_Ptr := Map_Ptr(new_Frame(title, Frame_Ptr(Obj)));
  begin
    New_Map.Our_Map := The_Map;
    New_Map.Menu_Bar := new_MenuBar;
```

```
    SetMenuBar(New_Map, New_Map.Menu_Bar);
    Create_Menus(New_Map.Menu_Bar);
    New_Map.NYI_Dialog := new_NYI(parent => Frame_Ptr(New_Map),
                                  title  => +"Not Yet Implemented", modal  => True);
    println(stdout, +"after nyi constructor in New_Map");
    println(stdout, +"before unit_list constructor in New_Map");
    New_Map.Positions := new_Unit_List;
    println(stdout, +"after unit_list constructor in New_Map");
    New_Map.SM := new_Socket_Monitor(+"Map Socket Monitor", Host_Addr);
    setPriority(New_Map.SM, java.lang.thread.Min_Priority);  -- to avoid deadlock
    Socket_Monitor.Start(New_Map.SM);
    println(stdout, +"after Socket Monitor start New_Map");
    New_Map.The_Thread := new_Thread(New_Map.Runnable'Access, +"Map Frame thread");
    setPriority(New_Map.The_Thread, java.lang.thread.Min_Priority+2);
        -- to avoid deadlock
    start(New_Map.The_Thread);
    println(stdout, +"after Map Frame Thread start in New_Map");
    return New_Map;
  end new_Map;

  function handleEvent(This : access Map_Obj;  evt : Event_Ptr) return Boolean is
    -- handle window close event
    Super : Frame_Obj renames Frame_Obj(This.all);  -- non dispatching view of "parent"
  begin
     if evt.id = java.awt.Event.Window_Destroy then
       done(This.SM);
       Stop(This.The_Thread);
       hide(This);
       dispose(This);
       return true;
     elsif (evt.target.all in MenuItem_Obj'Class) then
         -- selected some menu item
       declare
         Item  : MenuItem_Ptr := MenuItem_Ptr(evt.Target);
         Label : String_Ptr := GetLabel(Item);
       begin
          if Label.all = Ada_To_Java_String("Quit").all then -- kill the frame
              -- same logic as Event.Window_Destroy
            socket_monitor.done(This.SM);
            --stop our thread by setting This.The_Thread to null
            --the loop in run finishes so Run exits and the thread dies
            stop(This.The_Thread);
            hide(This);
            dispose(This);  -- close down the frame and return to the login screen
            return true;
          elsif Label.all = Ada_To_Java_String("Observe Sensors").all then
            -- toggle this & call socket_monitor.suspend() or .resume()
            -- ?? or should we suspend our thread that reads from the queue?
            show(this.NYI_Dialog);
            return True;
          else -- some other menu? this is an error
            print  (stdout, +"ERROR: Other menu selected. Label: ");
            println(stdout, Label);
            return java.awt.Frame.handleEvent(Super'access, evt);
          end if;
       end;
     elsif evt.id = java.awt.Event.Mouse_Up then
       declare
         Handled : Boolean;
       begin
         Handled := Unit_List.MouseUp(This.Positions, evt, evt.x, evt.y);
                -- delegate click
         if Handled then repaint(This); end if; -- click did something so update screen
         return Handled;
       end;
     else -- not window destroy and not menu item. Pass on to super & container
       return java.awt.Frame.handleEvent(Super'access, evt);
     end if;
  end handleEvent;
  procedure paint(This : access Map_Obj; G : Graphics_ptr) is
    Result : Boolean ; -- stores drawImage result. True if all bits avail. else false
  begin
       -- temporary. replace with double buffering
    Result := drawImage(G, This.Our_Map, 0, 0, This.ImageObserver'access);
    print(stdout, +"Redrawing Map Image. All bits avilable: ");
    println(stdout, Result);
    Unit_List.Paint(This.Positions, G);
  end paint;

  procedure update(This : access Map_Obj; G : Graphics_ptr) is
   --!! in the future avoid total redraw.
   --!! in the future use clipping regions.
  begin
    paint(This, G); --don't clear background first
  end update;

  procedure run(This : access Map_Obj) is
    -- called when thread is started
    -- Implements Runnable
    A_Msg : Message_Ptr;
  begin
      -- new_Frame() set the thread to /= null
      -- Suspend() will set thread to null when we should pause
      -- then we'll just reallocate in Resume() which will call Run again
    while This.The_Thread /= null loop
      -- get next message from the queue (may block)
      -- add message to the Unit_List.
      -- this will cause it to be displayed next time the frame is repainted
      -- yield() so that other threads get the CPU

      yield;  -- so that other threads get the CPU
      A_Msg := Remove;  -- synchronized call may block

      --!! if message kind is start, then clear out unit_list for that sensor
      --!! since a new set of positions is arriving

      if A_Msg.Kind = Message.Start then
        Clear(This_List => This.Positions, For_This_Sensor => A_Msg.Sensor);
      elsif A_Msg.Kind = Message.Observation then
        Add(To_List    => This.Positions,
            For_Sensor => A_Msg.Sensor,
            Item       => new_Unit(A_Msg.Enemy, x=> A_Msg.X, y=> A_Msg.Y));
      else -- A_Msg.Kind = Message.Stop
        repaint(This);  -- processed a new set of observations so make sure they
                        -- show up on the screen (minimal refresh)
      end if;
      yield;  -- so that other threads get the CPU
    end loop;

  end run;

  procedure stop(This : access Map_Obj) is
  begin
    This.The_Thread := null;  -- will cause run to exit its loop & stop
  end stop;

  --add in suspend() and resume() to Map
  --applet calls suspend() and resume() when it gets called
  --suspend sets the_thread = null
  --resume allocates it again
end Map;
-------------------------------------------------------------------------------
--  * AJPO GCCS Demo
--  * Socket Monitor  -- reads messages from socket and adds them to queue
```

```
---------------------------------------------------------------------------------
with java.lang.String;        use java.lang.String;
with Interfaces.java;         use Interfaces.java;
with java.lang.Thread;        use java.lang.Thread;
with java.net.Socket;         use java.net.Socket;
with java.io.InputStream;     use java.io.InputStream;
with java.io.DataInputStream; use java.io.DataInputStream;
with java.net.URL;            use java.net.URL;
with Message;                 use Message;
with Message_Queue;           use Message_Queue;
pragma Elaborate_All(Message_Queue);
package Socket_Monitor is
  type Socket_Monitor_Obj is new Thread_Obj with private;
  type Socket_Monitor_Ptr is access all Socket_Monitor_Obj;
  function new_Socket_Monitor(title              : String_Ptr;
                              Host_Addr          : String_Ptr;
                              Obj                : Socket_Monitor_Ptr := null)
              return Socket_Monitor_Ptr;
  pragma Convention(Java_Constructor, new_Socket_Monitor);
  procedure run(Obj : access Socket_Monitor_Obj);
  procedure done(This : access Socket_Monitor_Obj);
private
  type Socket_Monitor_Obj is new Thread_Obj with record
    Sock           : Socket_Ptr;
    in_stream      : DataInputStream_Ptr;
    Base_Host_Addr : String_Ptr;
  end record;
end Socket_Monitor;

with java.lang.System;       use java.lang.System;  -- for stdout
with java.io.PrintStream;    use java.io.PrintStream;
with java.net.URL;           use java.net.URL;
with java.net.InetAddress;   use java.net.InetAddress;
with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
with Ada.Text_IO;            use Ada.Text_IO;
with Ada.Integer_Text_IO;    use Ada.Integer_Text_IO;
package body Socket_Monitor is
  function Open_Socket(This : access Socket_Monitor_Obj) return DataInputStream_Ptr;
  function new_Socket_Monitor(title              : String_Ptr;
                              Host_Addr          : String_Ptr;
                              Obj                : Socket_Monitor_Ptr := null)
              return Socket_Monitor_Ptr is
  New_SM : Socket_Monitor_Ptr :=
                           Socket_Monitor_Ptr(new_Thread(title, Thread_Ptr(Obj)));
  begin
     New_SM.Base_Host_Addr := Host_Addr;
     return New_SM;
  end new_Socket_Monitor;

  function Open_Socket(This : access Socket_Monitor_Obj) return DataInputStream_Ptr is
        host_inet : InetAddress_ptr;
        port : integer := 1050;
  begin
        println(stdout, +"before new socket call");
        host_inet := getbyName(This.Base_Host_Addr); -- translate string inet #
        This.Sock := new_Socket(host_inet, port); -- make socket
        println(stdout, +"after new socket call.");
        print(Stdout, +"Now Connected to: ");
        println(stdout, getHostName(getInetAddress(This.Sock)));

        println(stdout, +"about to return input stream");
            -- initiate the input stream
        return new_DataInputStream(getInputStream(This.Sock));
   end Open_Socket;
   procedure done(This : access Socket_Monitor_Obj) is
   begin
        println(stdout, +"Closing InputStream");
        close(This.in_stream);
        close(This.Sock);
        stop(This);
   end done;
  procedure run(Obj : access Socket_Monitor_Obj) is
    -- never terminates by itself. the done() method is called to shut it down.
    str_ptr : String_Ptr := new String_Obj;
    A_Msg : Message_Ptr;
    use type String_Ptr;
  begin
    Obj.in_stream := Open_Socket(Obj); --open socket and set input stream to socket
    While_More_Data: loop
      Str_ptr := readline(Obj.in_stream);
      Yield;  -- let other threads proceed
      exit when str_ptr = null;
      print(stdout, +("Socket Message: ")); println(stdout, str_ptr);
      print(stdout, +"This socket message is "); print(stdout, length(str_ptr));
      println(stdout, " characters long");
      A_Msg := new_Message(str_ptr);
      Yield;  -- let other threads proceed
      Message_Queue.Add(Item => A_Msg);
      Yield;  -- let other threads proceed
    end loop While_More_Data;
    done(Obj);
  exception
    when java.io.IOException =>
        println(stdout, +"Got a Java.io.IOException inside of Socket_Monitor.Run");
        done(Obj);
  end run;
end Socket_Monitor;
---------------------------------------------------------------------------------
--  * AJPO GCCS Demo
--  * Message Queue -- a Queue (FIFO style) of Messages
--  *  The queue will block on Remove calls until new messages are added
--  *  Calls to Add never block
---------------------------------------------------------------------------------
with java.lang;          use java.lang; -- for InterruptedException and type Object
with Message;            use Message;
package Message_Queue is --only one message_queue. This is an ASM
   pragma Elaborate_Body;
   -- This class is intended to run in a Multi-Threaded environment
   procedure Add(Item : access Message_Obj);
   function Remove return Message_Ptr;
     -- will block if queue is empty
   procedure Clear;
private
   -- these must be declared in the spec's private part to be
   -- primitive operations on the tagged type
   type Node;
   type Node_Ptr is access Node;
   type Node is record Msg : Message_Ptr; Next : Node_Ptr; Prev : Node_Ptr; end record;
   type Message_Queue_Obj is new Object with record
      Head : Node_Ptr;
      Last : Node_Ptr;
   end record;
   type Message_Queue_Ptr is access all Message_Queue_Obj'class;
   procedure QAdd(To : access Message_Queue_Obj; Item : access Message_Obj);
   function QRemove(From : access Message_Queue_Obj) return Message_Ptr;
     -- will block if queue is empty
   procedure QClear(This : access Message_Queue_Obj);
   -- This class is intended to run in a Multi-Threaded environment
   pragma Convention(Ada_Synchronized, QAdd);
   pragma Convention(Ada_Synchronized, QRemove);
   pragma Convention(Ada_Synchronized, QClear);
end Message_Queue;

---------------------------------------------------------------------------------
--  * AJPO GCCS Demo
```

```
--  * Message -- Sensor observations from the server
---------------------------------------------------------------------------------
with java.lang;         use java.lang;
with java.lang.String;  use java.lang.String;
with Names;             use Names;
package Message is
   type Kinds_Of_Messages is (Start, Stop, Observation);
   type Message_Obj is tagged limited record
      Kind       : Kinds_Of_Messages;  -- Which kind of message did the sensor send
      Sensor     : Names.Sensors;      -- Which sensor recorded this enemy
         -- these three are only valid if Kind = Observation
         -- should be variant record, but these aren't yet supported
      X          : Integer;            -- X coord of enemy
      Y          : Integer;            -- Y coord of enemy
      Enemy      : Names.Enemy_Kinds;  -- Which type of enemy was seen
      --workaround for broken exceptions:
      Valid   : Boolean := True;  -- set to true if a valid message was built
                                  -- if false, all fields are invalid
   end record;
   type Message_Ptr is access all Message_Obj;
   function new_Message(Str : String_Ptr) return Message_Ptr;
   Incomplete_String : exception; -- raised if New_Message is given an incomplete string
end Message;

--------------------------------------
--  * AJPO GCCS Demo
--  * Names :Sensors and Targes -- common types across Client & Server
--------------------------------------
with java.lang.String; use java.lang.String;
package Names is
   Illegal_Value : exception ;
   type Sensors is (Air, Gnd, Sat) ;
   function To_String (S : Sensors) return String_Ptr;
   function To_Sensor (Str : String_Ptr) return Sensors;
   type Enemy_Kinds is (Tank, Infantry, Artillery);
   function To_String (EK : Enemy_Kinds) return String_Ptr;
   function To_Enemy  (Str : String_Ptr) return Enemy_Kinds;
end Names;

--------------------------------------
--  * AJPO GCCS Demo
--  * Unit List -- an  SLL (LIFO style) of Units
--------------------------------------
with java.lang;         use java.lang;
with java.awt.Graphics; use java.awt.Graphics;
with java.awt.Event;    use java.awt.Event;
with Unit;              use Unit;
with Names;             use Names;
package Unit_List is
   type Unit_List_Obj is tagged limited private;
   type Unit_List_Ptr is access all Unit_List_Obj;
   function new_Unit_List return Unit_List_Ptr;
   type Iterator is private;
   procedure Initialize(This_Iterator   : in out Iterator;
                        To_This_List    : access Unit_List_Obj;
                        For_This_Sensor : Names.Sensors) ;
   function Current(In_This_Iterator : in Iterator) return Unit_Ptr;
   procedure Next(In_This_Iterator : in out Iterator);
   function Is_Done(This_Iterator : in Iterator) return Boolean;
   procedure Add(To_List    : access Unit_List_Obj;
                 For_Sensor : Names.Sensors;
                 Item       : access Unit_Obj);
   procedure Clear(This_List : access Unit_List_Obj; For_This_Sensor : Names.Sensors);
   procedure paint(This : access Unit_List_Obj; G : Graphics_ptr);
    -- use to draw Unit_List in frame
   function mouseUp(Obj : access Unit_List_Obj; evt : Event_Ptr;
                x   : Integer;            y   : Integer) return Boolean;
    -- convenience function called when mouse is released inside a unit_list
    -- called from map's handleEvent()
private
...
end Unit_List;

--------------------------------------
--  * AJPO GCCS Demo
--  * Abstract Unit Icon (to be overlayed on Map frame)
--  *  all units are 32x18 icons set in the top left of a 32x32 cell
--  *  currently the grid is 8x8 pixels to server grid
--  *  so all unit icons take up 4x4 cells and can overlap
--------------------------------------
with java.awt.Event;      use java.awt.Event;      -- needed for handleEvent
with java.awt.Image;      use java.awt.Image;      -- needed for storing Map
with java.lang.String;    use java.lang.String;
with Interfaces.java;     use Interfaces.java;
with java.awt.Graphics;   use java.awt.Graphics;   -- needed for overriding paint
with java.awt.Rectangle;  use java.awt.Rectangle;
with java.awt.Color;      use java.awt.Color;
with NYI;                 use NYI;
with Names;               use Names;
package Unit is
  type Unit_Obj is new Rectangle_Obj with record
    Selected : Boolean := False;
    Kind     : Names.Enemy_Kinds;
  end record;
  type Unit_Ptr is access all Unit_Obj;
  function new_Unit(Kind : Names.Enemy_Kinds; x : integer;
                    y : integer; Obj : Unit_Ptr := null) return Unit_Ptr;
  pragma Convention(Java_Constructor, new_Unit);
  procedure paint(This : access Unit_Obj; This_Color : Color_Ptr; G : Graphics_ptr);
    -- use to draw Unit in frame
  function mouseUp(Obj : access Unit_Obj; evt : Event_Ptr;
                x   : Integer;        y   : Integer) return Boolean;
   -- convenience function called when mouse is released inside a unit
    -- called from map's handleEvent()
  procedure highlight(This : access Unit_Obj);
     -- toggles the selection state
     -- causes paint to draw an outset rectangle in black 1 pixel
     -- called when mouseUp happens
end Unit;
```

Appendix B – Server Software Source Code Listings

```
with Battlefield;--Stores current battlefield data
with AirSensor;--remote observation sensor
with Satellite;--remote observation sensor
with GroundSensor;--remote observation sensor
with Ada.Text_IO;use Ada.Text_IO;
with Socket;--Socket constants and functions
with Ada_In;--Internet Socket constants and functions
with Interfaces.C;
with Interfaces.C.Strings;
with Errno;--provides socket error messages
with Ada.Unchecked_Conversion;
with Ada.Characters.Latin_1;  --to obtain the NL/LF character 16#10#
with Sensor_Servers;
with Sensor_Server_List;
with Block_SIGPIPE;
with Handle_SIGINT; -- to allow graceful shutdown on ^C
procedure Main is  use type Interfaces.C.Int;  use type Battlefield.Target;
  package C renames Interfaces.C;
  --Converts Internet style address to 'generic' socket address
  function To_Sockaddr is new Ada.Unchecked_Conversion
                                    (Ada_In.Sockaddr_In, Socket.Sockaddr);
  --Socket will be bound to the local port
  Local_Port       : C.Unsigned_Short := 1050;
  The_Socket       : C.Int;--Socket created locally
  Client_Socket    : C.Int;--Socket connection from client
  --Variables for Internet to 'generic' address conversion
  Temp_Address     : Ada_In.Sockaddr_In;
  The_Address      : Socket.Sockaddr_ptr;
  MaxClients       : C.Int :=10;--Maximum number of Clients that will
  --be accepted for socket connection in one execution of program
  One : Socket.const_char_ptr := new C.Signed_Char'(1);
  SSL : Sensor_Server_List.Sensor_Server_List_Obj;
begin
  Put_Line("Main is Running!!!");
  Put_Line("Creating Socket!!!");
  The_Socket := Socket.Socket(Socket.AF_Inet, Socket.Sock_Stream, 0);
  Put_Line("Socket Number: " & C.Int'Image(The_Socket));
   if Socket.setsockopt(s       => The_Socket,
                      level    => Socket.SOL_SOCKET,
                      optname => Socket.SO_REUSEADDR,
                      optval       => One,
                      optlen       => 4
                      ) = -1 then
     Put_Line("setsockopt Failed!!! with Error No: "&
              C.unsigned'image(C.Unsigned((Errno.Get_Errno))));
     Errno.perror(C.Strings.New_String("setsockopt"));
     raise Program_Error;
   end if;
  --Create Adress to bind to Socket
  Temp_Address.Sin_Family := C.Short(Socket.Af_Inet);
  Temp_Address.Sin_Addr.S_Addr := C.Unsigned_Long(Ada_In.Inaddr_Any);
  Temp_Address.Sin_Port := Ada_In.htons(Local_Port);
  The_Address := new Socket.Sockaddr'(To_Sockaddr(Temp_Address));
  Put_Line("Binding to Address");
  if Socket.Bind(The_Socket, The_Address, 16) /= -1 then
    Put_Line("Bind Successful!!");
  else
    Put_Line("Bind Failed!!!");
    Errno.perror(C.Strings.New_String("Bind"));
    raise Program_Error;
  end if;
  --begin listening for socket connections
  if Socket.Listen(The_Socket, MaxClients) /=-1 then
    Put_Line("Server Listening!!");
  else
    Put_Line("Listen Failed!!!");
    Errno.perror(C.Strings.New_String("Listen"));
  end if;
  --Initialize Battlefield
  Battlefield.Init;
  --Start Sensors
  Airsensor.Init;
  GroundSensor.Init;
  Satellite.Init;

  select
    Handle_SIGINT.SIGINT_Handler.Interrupted;
    New_Line;
    Put_Line("Received a ^C. Beginning Shutdown of Servers");
    Ada.Text_IO.Flush;
      -- if received ^c, then shutdown all servers & tasks gracefully
    Put_Line("Shutting Down Air Sensor");
    Ada.Text_IO.Flush;
    Airsensor.Finish;
    Put_Line("Shutting Down Ground Sensor");
    Ada.Text_IO.Flush;
    Groundsensor.Finish;
    Put_Line("Shutting Down Satellite Sensor");
    Ada.Text_IO.Flush;
    Satellite.Finish;
    Put_Line("All Sensors Shut Down. Now Killing Server Tasks");
    Ada.Text_IO.Flush;
    Sensor_Server_List.Close_All(SSL);
    Put_Line("Program Terminating Normally");
    Ada.Text_IO.Flush;

  then abort
      --put into loop and spawn new task and add to list
    loop
      Ada.Text_IO.Flush;
      Client_Socket := Socket.Ada_Accept(The_Socket, null, 0);
      Ada.Text_IO.Flush;
      if Client_Socket /= -1 then
        Ada.Text_IO.Flush;
        Put_Line("Connection Accepted!!");
        Put_Line("Spawning new task for socket " & C.Int'Image(Client_Socket));
        Sensor_Server_List.Add
          (To_List => SSL, Item => new Sensor_Servers.Sensor_Server(Client_Socket));
      else
        Ada.Text_IO.Flush;
        Put_Line("Connection Error on Accept!!");
        Errno.perror(C.Strings.New_String("Accept"));
      end if;
    end loop;
  end select;  --exit if ^c trapped

exception
  when Others =>
    Put_Line("Shutting Down Air Sensor");
    Ada.Text_IO.Flush;
    Airsensor.Finish;
    Put_Line("Shutting Down Ground Sensor");
    Ada.Text_IO.Flush;
    Groundsensor.Finish;
    Put_Line("Shutting Down Satellite Sensor");
    Ada.Text_IO.Flush;
    Satellite.Finish;
    Put_Line("All Sensors Shut Down. Now Killing Server Tasks");
    Ada.Text_IO.Flush;
    Sensor_Server_List.Close_All(SSL);
    Put_Line("Program Terminating Normally");
    Ada.Text_IO.Flush;
end Main;
```