Development of a Distributed, Cross-Platform Simulator

Thomas C. Brooke
Titan Corporation
5350 Tomah Dr.
Colorado Springs, CO 80908
brooket@acm.org

ABSTRACT

In developing real-time mission control software for terminals in a large satellite communications system, my team realized that a script-based stimulus/response tool was inadequate for developmental testing and training. As an initial proof-of-concept, we first designed a monolithic, single-user system simulator for engineering development. During the project, the requirements expanded to include the addition of a multi-user, cross-platform capability, and later distribution in a two-tier client/server system.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: Applications; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*client/server, distributed applications*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms

Design, Performance, Experimentation, Standardization

Keywords

Ada, distributed, linux, portability, satellite, simulation, testing, training, windows

1. BACKGROUND

Our group builds software applications for satellite communications systems. Our products are mainly ground based[1] planning and execution tools. Appendix A contains a more detailed description of the system involved in this project.

[1]As much fun as it would be, we don't have any software flying on the satellites themselves. Yet.

SIGAda'02, December 8–12, 2002, Houston, Texas, USA.

We were developing an application for direct control of a satellite ground terminal, targeted to a laptop running Windows NT[2][3] (see figure 1).

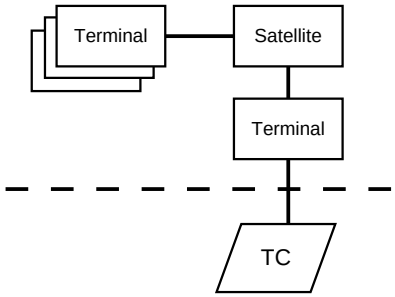


Figure 1: Terminal Control Application Context

The key functional areas of the TC (Terminal Control) software are

- Controlling satellite antennas
- Managing communications networks
- Monitoring networks and terminals

The interface between TC and the terminal is message based. TC sends a command message to the terminal, and the terminal sends one or more response messages back. The terminal may spontaneously send an "unsolicited" message to TC, but it expects no message in response.

The physical connection between TC and the terminal is a basic RS-232 serial line [4]. There is also a provision for running TCP/IP (Transport Control Protocol / Internet Protocol) over this line, via PPP (Point-to-Point Protocol), and exchanging the messages through a socket.[4]

The problem we were experiencing was that real terminals are expensive, and satellite resources are scarce. So, finding terminals, locating baseband equipment, and coordinating satellite time for day-to-day testing is practically impossible. What we needed was something to stand in for the rest of the system—everything above-the-line in figure 1.

Since the messaging interface was so simple, our first approach was to use an off-the-shelf terminal emulator (like

[2]Windows NT is a registered trademark of Microsoft Corporation
[3]Yes, it's dated. But it was the platform that the customer had already fielded.
[4]This would come in very handy later on.

HyperTerminal.[5] We manually constructed binary response messages,[6] and concatenated them into response scripts. During testing, when TC sent a message to the "terminal," we would manually transfer a prepared response file back. While this method worked, after a fashion, developing the individual messages and scripts was extraordinarily labor intensive.

Our second approach was a more sophisticated, locally developed tool, which featured hierarchical scripts, default responses,[7] message templates, and helper functions for constructing response messages dynamically at run-time, based on data in the request itself. Although the new tool reduced test preparation effort by an order of magnitude, it was still a difficult, time-consuming process. A basic test scenario consists of hundreds of messages (excluding default responses) and dozens of scripts. Devising the test suite for a new scenario remained very expensive.

The fundamental difficulty turns out to be managing terminal and system state. First of all, messages and scripts have to accommodate the expected system state: they have to account for prior commands which may have affected networks, antennas, or terminals. For example, if a command moves an antenna, then subsequent requests for antenna information have to reflect the new location. Second, response scripts must be sequenced in the expected order of execution. For example, the response tool should not be waiting for an antenna information request when TC is trying to activate a network. Due to the complexity of the TC software, determining this ordering is often a trial and error process.

A script-based tool can be invaluable for testing error paths and impossible conditions, because it's very easy to produce *exactly* the desired behavior. But we also needed a less labor intensive way to perform nominal path and day-to-day developmental testing.

1.1 The Project

I was fed up with generating the "same old scripts" for the "same old scenarios." So, I decided to initiate a low-level IR&D (Internal Research and Development) project to build a simple system simulator. The TSim (Terminal Simulator) would serve as a proof-of-concept, with the hope that it could be expanded into a complete testing solution.

There was really only one constraint: it had to be inexpensive. As an internal, unfunded project, I had to keep the costs down. This meant minimizing time, effort, and capital.

My goals for the project were:

- Make it work. TSim was to be a tool for our use, not an academic exercise.
- Explore Ada95. Because it was an independent project, there was an opportunity to experiment with new techniques in a forgiving environment.
- Employ third-party tools, packages, and components wherever possible. This was a direct result of the cost constraint.

[5]HyperTerminal is ©1998 by Hilgraeve, Inc.

[6]We built some crude tools to format the messages and attach the framing.

[7]Where the most common response sequence to a message would be sent automatically, unless a specific response were pending

The following sections describe the evolution of TSim and the lessons learned during its development. But it's important to keep the lessons in context.

The first priority was to make things work. Towards that end, we used what we knew—except where we were experimenting. Due to the constraints, we spent little or no effort scouring the literature (or web resources) for elegant solutions: we would apply the first tool/technique that looked workable, or we'd just figure something out on our own.

Furthermore, despite a strong Ada83 background, we had only limited Ada95 experience. As a result, some of the lessons "learned" on this project are probably obvious. And some of them are probably wrong—or at least suboptimal.

One thing to keep in mind that TSim is a low-fidelity simulator. It does not need to comprehensively and accurately model the full system behavior;[8] it only needs to simulate the Terminal/TC interface. We did consider more detailed simulation standards, such as IEEE 1278 [6], but concluded that they were overkill for this particular application.

2. ROUND 1

Initially, the TSim requirements were fairly simple: build a basic stimulus/response engine which would simplify TC testing by maintaining state and automating responses to commands.

It would be a monolithic,[9] console-based application (no GUI (Graphical User Interface)) which would simulate just a single terminal—the one "connected" to TC. Figure 2 depicts the general block architecture and relative source code sizes of the major software components.

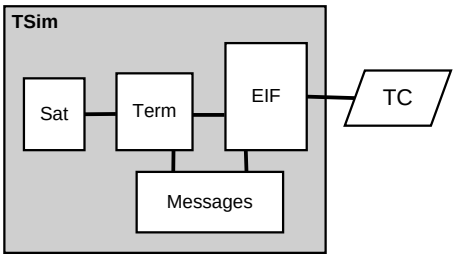


Figure 2: Round-1 Block Diagram

Anticipating future needs, most of the effort went into the EIF (External Interface) and the Messages hierarchy. The EIF provides message-level communication with TC. It sends and receives messages on the serial port (or socket), and implements the alternating bit protocol used for elementary error detection. Correct behavior of the EIF is critical to the realism of TSim. The Messages hierarchy not only defines the actual message formats, but also the message framing and serialization (streaming). A good design here would pay dividends when scaling up TSim, as adding the dozens of new messages needed would be much easier.

At this stage, the result was an application which looked like figure 3.

The EIF implementation used sockets for Terminal/TC communication, mainly because it was more convenient than hanging a loopback null-modem cable off the back of the

[8]Not yet, anyway.

[9]Where "monolithic" here denotes a single executable, not one-big-subprogram.

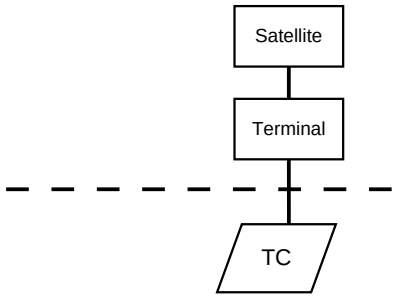


Figure 3: Round-1 Application Context

computer—or stringing (and re-stringing) null-modem cables between computers. For this, we used the AdaSockets package developed by Samuel Tardieu [11], and adapted for Win32[10] by Jerry van Dijk [12].

Even in this relatively simple application, we still used the Booch Components [13] for message buffering. Not that it would have been so difficult to manually code a buffer—after all, who doesn't have a generic list package laying about? But we fully expected to need other capabilities—such as guarded or synchronized forms, more complicated containers, and iterators—later on, and manually coding those would have been much more difficult than reusing available components.

2.1 Controlled Types and Task Termination

A `Terminal` object contains an EIF component to manage the message handling between the simulated terminal and TC. When there's no further use for a terminal (e.g., when ending the simulation), the EIF must be shutdown as well. Ideally, EIF shutdown should be automatic. For example, when the `Terminal` object leaves scope, the EIF should stop.

The EIF consists of several tasks, and unfortunately tasks don't just stop at the end of a block. Instead, finalization of the `Terminal` is held up until the tasks in the EIF are [ready to] terminate. These tasks don't know about the enclosing terminal, and so they're never ready.

We needed a way to signal the EIF tasks to stop, and a `terminate` alternative just wouldn't work. Since we wanted them to terminate at end of scope, we tried to use the finalization characteristics of `Limited_Controlled`:

```
with Ada.Finalization;
use  Ada.Finalization;
package Task_Termination is

  type Agent is new Limited_Controlled
     with private;

  procedure Finalize
     (Object : in out Agent);

private

  task type Processor is
    entry Stop;
  end Processor;

  type Agent is new Limited_Controlled
     with record
    Worker : Processor;
  end record;

end Task_Termination;
```

Where the actual work is done in `task type Processor`, and the `Agent` wrapper exists solely for termination.

```
package body Task_Termination is

  task body Processor is
  begin
    loop
      select
        accept Stop;
        exit;
      else
        -- do something useful,
        -- like read some data.
      end select;
    end loop;
  end Processor;

  procedure Finalize
     (Object : in out Agent) is
  begin
    Object.Worker.Stop;
  end Finalize;

end Task_Termination;
```

Unfortunately, this doesn't work. Objects are not finalized at the end of a block until *after* all tasks terminate [10, 7.6.1(1)]. In practice, everything would come to a screeching halt at the end of the enclosing block, waiting for the tasks to terminate, before calling `Finalize`. The tasks themselves were still running, waiting for `Finalize` to call their termination entries. Catch-22.

Despite some cautions on `comp.lang.ada` and in [1], we settled on ATC (Asynchronous Transfer of Control) to interrupt the EIF. By using a protected object as a broadcast signal, all the EIF tasks could be notified at once. While this approach was not as clean as automatic termination, it was an improvement over individual notification of each task separately.

2.2 Child Packages for Structuring

Initially, the Messages design used child packages to physically collect all of the units related to the message interface into a unified subsystem. Figure 4 depicts this hierarchy of message related packages.

It seemed beneficial to mirror domain-level relationships in the structure of the library units. Thus, all packages related to messages, as defined in the terminal IDD (Interface Design Document), are nested within `package IDD`, even if there is no dependency between them.

For example, although the package `IDD.Message.Types` contains type definitions that are common to all messages, and are formally defined in the terminal IDD, there is nothing in it which requires visibility into `IDD` or `IDD.Messages`.

While this structure makes for a nice diagram—clearly identifying the relationships among various messaging packages—it also tends to clutter the code when referencing basic concepts.

[10] Win32 is a registered trademark of Microsoft Corporation.

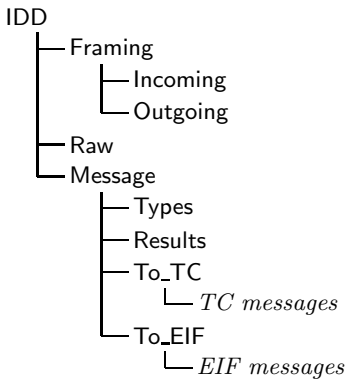


Figure 4: Message Package Hierarchy

```
begin
   ...
   Send(IDD.Message.To_TC.Ack.Formatted(M));
   ...
end;
```

A flatter hierarchy, which only uses child packages for necessary visibility, might have resulted in something like this:

```
begin
   ...
   Send(Message.Ack.Formatted(M));
   ...
end;
```

We prefer the latter style. Using child packages to illustrate relationships is useful for familiarization, but it can cloud the detailed implementation. At the level where these packages are used—such as the code fragments above—the relationships should already be well understood. Any introduction or overview is best provided in summary documentation, outside of the library unit structure.

3. ROUND 2

Almost before Round-1 was complete, the project scope expanded. This was not entirely unexpected; despite the simplicity of the initial application, TSim was designed for growth. The new requirements pushed it in three directions:

First, we needed more than just a single terminal. Many of the more difficult things to test come from interactions between TC and other terminals in the system—especially other terminals with their own TC connections. We hoped that an OO (Object Oriented) design would make it only slightly harder to create two or more terminals, than one.

Second, a console-based interface is easy and effective, but it's not very attractive. At some point, especially to gather broader support for full-scale development, we would have to add a GUI. Fortunately, we had anticipated this, and kept the internal simulation and modeling largely separate from the user interface. All (!) we had to do was find an easy way to build a simple GUI.

Third, in order to demonstrate that TSim could be a complete solution, it needed to fully implement one of TC's core functional areas (see section 1). We chose the control of satellite antennas. This would entail new messages to support additional commands, responses, and events.

As expected, supporting multiple terminals was not very difficult. Creating several terminal objects, rather than just one, was almost trivial. And due to the use of sockets for EIF communication, it was a simple matter for any or all of these terminals to connect to a TC instance anywhere on the network. Had the EIF initially implemented the serial interface, TSim would have been limited to at most two terminals (with connected TC applications) at a time. This resulted in an architecture as shown in figure 5.

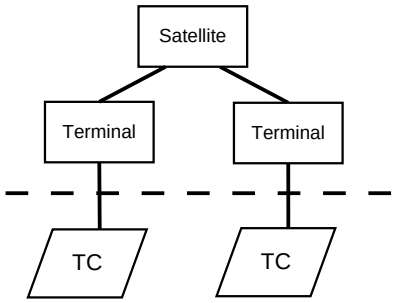


Figure 5: Round-2 Application Context

3.1 Adding a GUI

One of the design goals was to keep TSim as flexible as possible. Part of this was ensuring that it would remain embeddable; that is, that it could act as a "simulation server" with no user interface at all. Meeting this goal would imply there was no technical limitation preventing a switch from a console-based to a graphical interface. And in fact, that turned out to be the case. Figure 6 shows the revised software block architecture at this stage.

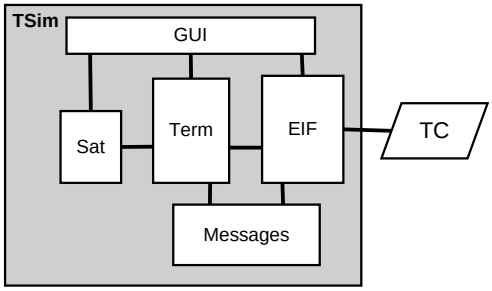


Figure 6: Round-2 Block Diagram

The absence of a user interface on the original Round-1 block diagram (figure 2) is intentional. The console (i.e., `Text_IO`) based interface was incorporated into the main subprogram. There were no application-specific library units developed for the Round-1 user interface.[11] However, we expected the graphical interface to be complex enough that it would require additional packages.

The difficulty was in deciding *which* framework/tool to use to build the GUI. As a general principle, we didn't want anything Win32 specific. Although at this time we were not seriously considering porting TSim to other platforms, we didn't want to impose any arbitrary limitations that would

[11]There were, generic input/output packages to support the text user interface, but they were not application specific.

later make porting more difficult. On the other hand, we did want something easy to use.

I don't remember how we ended up with the combination of TASH (Tcl Ada Shell) [14] and RAPID (Rapid Ada Portable Interface Design tool) [2]. It may just have been the first viable solution we found following a web search. There are a number of alternatives out there, and we didn't come close to investigating all of them.

For our purposes, TASH and RAPID made a productive team. RAPID made "painting" the GUI screens and generating skeleton Ada source very simple. And the high-level abstractions of TASH and Tcl/Tk [8] made implementing the GUI events easy. Ten minutes from painting the screen to running the application[12] is on a par with other well-known visual development environments.

The result consisted of two dialogs,[13] both modeless, and both visible (or "active") at all times. From the Main dialog, the user could create terminal objects, connect them to TC instances, and log them onto the satellite. From the Satellite dialog, the user could manipulate the satellite—mainly its antennas—directly, outside of the terminals and TC. This organization fit nicely into the block diagram of figure 6: the main dialog interacted with the Terminal and EIF subsystems, while the satellite dialog interacted with the Satellite subsystem. A fairly clean division.

The final product was very simplistic—far from production quality in both human factors and robustness. But this was only a proof of concept. It didn't need to have a polished, professional appearance; it just had to show the potential. If we could lop off a console interface and replace it with a basic GUI, we felt confident that we could also replace a naive GUI with a sophisticated one.

3.2 Limits of Inheritance

The single biggest failure of the project was the inability to devise a way to cleanly extend related abstractions independently. It's clear that this is a limitation of traditional single inheritance models. What's not so clear is whether this is also a limitation of Ada95.

Up to this point, TSim had supported only a single kind of terminal: one specific version of a terminal connected to TC. It was possible to create several instances of this particular terminal type, but they all had identical behavior. There is a need, however, to support multiple kinds of terminals. More specifically, to support terminals with distinct behaviors. For example:

- Terminals running different software/firmware versions
- Terminals in different modes (e.g., with and without TC connected)
- Different terminal types altogether, such as from different manufacturers

Additionally, there may be terminal objects with different simulation behavior. For example:

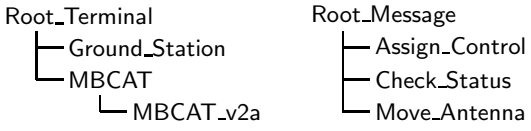
- Different fidelity, such as more accurately modeling real terminal capacity or timing
- Different simulation strategies, such as autonomous vs. unreliable operations (e.g., for fault injection)

We didn't know specifically what kinds of terminals would be needed, nor did we have the resources to build many. But to be generally useful, TSim would have to support arbitrary Terminal types.

Furthermore, as TSim was enhanced, it would need to define and support additional Messages. Both in the course of normal development, as the simulator was completed, and also later on, as the system itself matured and additional messages were added to the IDD.[14]

We wanted to be able to define new messages, and only add support for them to the terminals as needed. In general, not all terminals will support all messages (e.g., when new software versions are fielded to implement new messages, the old software versions remain unchanged). This would give us the freedom to extend the message definitions, and incrementally update the simulated terminals. Similarly, we wanted to be able to define new terminal types without updating the message definitions.

Figure 7 illustrates the `Terminal` and `Message` class hierarchies.



(a) Terminals (b) Messages

Figure 7: Class Hierarchies

The EIF receives a stream of bytes, and constructs the appropriate messages from that:[15]

```
package EIF is
   ...
   procedure Receive
      (M : out Root_Message'Class);
   ...
end EIF;
```

The main purpose of a terminal object is to receive messages from TC, and process them:

```
begin
   ...
   loop
     EIF.Receive(Next_Message);
     Process(Next_Message);
   end loop;
   ...
end;
```

To process a message, we have to know

1. The contents of the message
2. The terminal implementation and state

[12] For a simple "Hello World" application. Building TSim's interface took substantially longer, but it was still a very efficient process.

[13] Or "screens", or "windows", or "forms"; take your pick.

[14] This had already happened once during TSim development. There was no reason to expect it wouldn't happen again.

[15] Note: the code fragments in this section are for illustrative purposes only. Don't try to compile them; it won't work.

So, each concrete terminal class has to define how to process each kind of message that it supports. Certainly, `Process()` can be implemented as a large `case` statement, based on the kind of message. But it would be much more OO—and maintainable—if `Process()` could dispatch on the message type (and implicitly the terminal type).

That is, we'd like to be able to define a `Process()` for each kind of message *within* each terminal; the loop above would then dispatch to the proper `Process()` subprogram. Supporting a new message type for a terminal is simply a matter of defining how to process it. Terminal types which don't support certain kinds of messages just don't define a `Process()` for them.

Obviously, this doesn't work.

Traditionally, each kind of message knows how to process itself:

```
package Message is

  type Root_Message is abstract tagged ...

  procedure Process
     (M : in out Root_Message);
  ...
end Message;

with Message;
use  Message;
package Assign_Control is

  type Messages is
     new Root_Message with private;

  procedure Process
     (M : in out Messages);
  ...
end Assign_Control;
```

However, this does not account for differing implementations among the different terminal types. It also doesn't provide the visibility into terminal state needed to actually process the message.

The Ada95 Rationale [7] suggests redispatching on a second, class-wide parameter:

```
with Terminal;
use  Terminal;
package Message is

  type Root_Message is ...

  procedure Process
     (M : in out Root_Message;
      T : in out Root_Terminal'Class);
  ...
end Message;

package body Message is
  ...
  procedure Process
     (M : in out Root_Message;
      T : in out Root_Terminal'Class) is
  begin
     ...
     Terminal.Process(T, M);
     ...
  end Process;
  ...
end Message;
```

But this only works when the secondary, redispatched class is independent of the first. In this case, the terminal processing still needs the message contents. This leads to visibility problems—a circular dependency between `Message` and `Terminal`. Furthermore, the incoming message started off in the correct terminal class in the first place; dispatching it over to `Message` just to send it back again seems pointless. In the end, we couldn't make redispatching work.

We also tried some other, pretty outlandish approaches; none of which were satisfactory:

- Parallel message hierarchies for each terminal class
- Inverting the class relationships (e.g., so that the EIF receives the messages, and dispatches them to the terminal)

Ultimately, we chose the crude but expedient procedural approach of a `case` statement within each terminal class. Although this requires changes in two places—creating a new `Process()` as well as updating the `case` statement—when adding support for a new message, it was still cleaner than force-fitting an awkward OO solution where it didn't belong.

4. ROUND 2A

At the conclusion of Round-2, TSim, as a proof of concept, was essentially complete. It demonstrated all the functional aspects needed of a system simulator for developmental testing. What happened next was an accident.

The TC application is tied to the Win32 platform. However, because the TC/Terminal interface uses RS-232 serial or sockets, the platform for TSim need not be.[16]

So far, TSim development had also proceeded on a Win32 platform. Although we had tried to avoid any obvious Win32 dependencies (e.g., we chose Tcl/Tk for the GUI, rather than Win32 primitives), we hadn't been overly obsessive about it. We wanted TSim to be portable, but we expected that the porting process—when eventually undertaken—would reveal a number of subtle dependencies to be exorcised.

Then one day, with only a Linux[17] box available, somebody started thinking:

- GNAT has a Linux version
- Tcl/Tk is available for Linux
- TASH supports Linux
- AdaSockets supports Linux
- The Booch Components are portable

The results were astonishing. Total lines of code modified porting to Linux: 1. Furthermore, this single modification was due to a change in the AdaSockets interface (we had been using an earlier version for Win32 development).

[16]There was nothing brilliant about this observation, as the real-world terminal itself is not Win32-based

[17]Linux is a trademark of Linus Torvalds.

As it turns out, the most time consuming part of the Linux port was getting and installing the various tools, and components needed to build TSim. This was made substantially easier through the efforts of the Ada for Linux Team [9], which provides pre-packaged Linux binaries. Total porting time, excluding environment installation: 20 minutes.

The resulting TSim executable performed identically to its Win32 counterpart (excepting cosmetic differences). We could now demonstrate a Win32 application (i.e., TC) interoperating with a simulator running under Linux.[18]

5. ROUND 3

As TSim matured, we began using it more. In some limited cases, we actually used it to test TC. Although it was doing exactly what we had hoped it would, we were still surprised at how much easier it made testing. We realized that a simulator could also make training easier and more realistic.

Not only do we build satellite communications planning and execution tools, but we also train the operators in their use. For TC training, we had been using the same script-based tool that we used internally for testing. Using scripts for training has several additional drawbacks:

- It stresses rote learning. The students must press the right button at the right time, exactly as prescribed.
- It's fragile. Pressing the wrong button often results in behavior so divergent from the script, that the only recourse is to start over.
- It may be misleading. Even if the students do the wrong thing, they may get the right answer, because the script is built to expect it.

Using TSim would improve initial classroom instruction, but it would also be valuable for proficiency training and "dry-running" new communications plans.

Not only are terminal and satellite resources difficult for us to reserve for testing, they're just as difficult for the operators to reserve for training. And even if training resources are available, the operators must be very careful not to impact other, "real" users. For testing new communications plans, or practicing emergency procedures, a simulator is much safer than using the real system.

In order to fully support training, TSim would need some enhancements.

Mainly, TSim would need to support a GUI for each TC. That's because the operator has access to *both* the control and planning system (which hosts TC) *and* the terminal itself. TSim would have to provide a simulated terminal interface to the operator.[19] The Round-2 solution has a terminal interface (the Main dialog screen), but it uses a single screen for *all* simulated terminals—and this screen is only displayed on the TSim platform.

Since TC operates under the Windows NT operating system, a monolithic application using remote GUI's was not feasible. We would not be authorized to run an X Server[20] or other remote display technology on the TC platform. We would need to actually host the terminal interface GUI.

A minor concern was TSim performance. A real-world training scenario consists of approximately 12 terminals. Not all of these terminals would be connected to TC applications, but all of them would have operators present. A large scenario might consist of even more terminals, and be geographically separated. Overall, this may result in a relatively large number of tasks.

If we were going to host the terminal interface GUI on the TC platform, it made sense to host the terminal object as well. This resulted in the software architecture shown in figure 8.

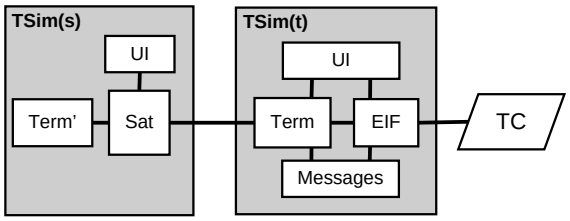


Figure 8: Round-3 Block Diagram

The question was: how to take a monolithic application, and divide it into client and server tiers? As with most things on this project, we tried to do it the easy way: we chose the Distributed Systems Annex [10, Annex E]. The advantage of this to us was that we could still treat the system as one big Ada95 application—just partitioned. From a practical standpoint, using an object in a different partition is no different from using a local object. Thus, the bulk of the TSim code could remain unchanged. All (!) we had to do was split it up.

The Annex E implementation for GNAT is GLADE (GNAT Library for Ada Distributed Execution)[21] [5]. It provides a PCS (Partition Communications Subsystem), a language for defining distributed partitions, and tools for building the partitions (executables).

The first step was to identify the partitions in detail. There were basically two:

Satellite – TSim(s): Contains satellite and system-wide simulation objects. A simple user interface provides direct control to the simulation operator. There would be only one instance of this partition, and it would run on the simulation "server."

Terminal – TSim(t): Contains simulation objects needed by a single terminal. The user interface provides the terminal operator with access to the expected "terminal" functions. There would be one instance (copy) of this partition per simulated terminal; it would run on the TC "client."

Would this work? Annex E states that multiple copies of a partition are legal, so long as all copies "present a consistent state of the partition" [10, E.1(12)]. Here, we wanted the state of each copy to be different—to maintain the state for a single, distinct terminal. Fortunately, we had already

[18] No, not technically impressive; after all, browsers and web servers do it all the time. But it was helpful for internal marketing.

[19] This terminal interface need not be displayed on the TC platform. But for training, it probably would be.

[20] X Window System is a trademark of The Open Group.

[21] Which should not be confused with Glade, the GUI builder for GTK+, of the same name [3].

defined the Terminal types as pointers to the actual implementation; this was so that the satellite object could find and notify terminals when the payload status changed.

That meant that each TSim(t) partition was not declaring a global object for its terminal, but rather was creating its terminal through an allocator. Within the "consistent state" of the combined partition, each allocated terminal remained distinct within the access type collection. The satellite partition could then find each distinct terminal through its access value; dispatching remotely to the appropriate partition instance because they are `Remote_Types`.

```
with Payload;
package Terminal is
  pragma Pure;

  type Root_Terminal is
     abstract tagged limited private;

  procedure Notify
     (T : access Root_Terminal;
      M : in     Payload.Messages'Class);
  ...
end Terminal;

with Terminal;
use  Terminal;
package Satellite_Interfaces is
  pragma Remote_Types;

  type Terminal_Ptr is
     access all Root_Terminal'Class;
...
end Satellite_Interfaces;
```

The most trouble we had when trying to convert TSim into a distributed application was in categorizing the library units. We started with the obvious `Remote_Call_Interface` packages (e.g., the Satellite object). However, that meant that all the library units on which that specification depended *also* had to be categorized. Many of these were packages defining domain-specific types, and so were categorized as `Pure`. But not all of them turned out to be `Pure`. In a number of cases we had to relax the category, defer the dependency to the body, or refactor the affected unit. And the problem was transitive; we also had to categorize all the dependent units of dependent units, and so on.

Ultimately, all the appropriate units were categorized, the partitions were defined, and the executables were built. As we had hoped, once the units had the proper categories, writing "distributed" code was exactly like writing monolithic code—we couldn't tell the difference. Partitioning was transparent to the application.

We did encounter one surprising problem. It had to do with, presumably, how partitions are assembled, and how tagged types are implemented. Essentially, the problem was that passing objects to a partition that is not expecting that specific type results in a failure—in our case, the application appeared to "hang."

As shown in the code fragments above, the Satellite will notify individual Terminals of "interesting" events. For example, when an antenna moves, or when the terminal's access authorization has been revoked, or any of a number of other events occurs, the satellite will send a notification message to the affected terminal(s). The Terminal will receive these notifications through the `Terminal.Notify()` method, and process them accordingly. So, the satellite (payload) messages look something like this:

```
with Payload;
package Antenna is

  type Move_Notify is
     new Payload.Messages with private;
  ...
end Antenna;

with Payload;
package Accesses is

  type Revoke_Notify is
     new Payload.Messages with private;
  ...
end Accesses;
```

And, within a specific Terminal implementation, these payload messages are handled by the class-wide `Notify()`:

```
with Payload;  use Payload;
with Antenna;  use Antenna;
with Accesses; use Accesses;
package body Ground_Station is
  ...
  procedure Notify
     (T : access Terminal;
      M : in     Payload.Messages'Class) is
  begin
    if M in Move_Notify then
      Process_Move(Move_Notify(M), T);
    elsif M in Revoke_Notify then
      Process_Revoke(Revoke_Notify(M), T);
    else
      Process_Unknown(Messages(M), T);
    end if;
  end Notify;
  ...
end Ground_Station;
```

The problem arises if a specific message is sent to a terminal, but the terminal partition does not include its definition (i.e., package). How could this happen? As with the situation described above in section 3.2, when a new payload message is added, the terminals may not be updated to handle it. So, if we create a new message:

```
with Payload;
package Network is

  type Teardown_Notify is
     new Payload.Messages with private;
  ...
end Network;
```

but do *not* update the body of `Ground_Station` to use it, we will have a problem. In a monolithic application this would be handled by the `else` branch within `Notify()`. However, in a distributed application, the partition only includes those library units that are directly or indirectly referenced. The new unit, `Network`, is not referenced, and so is not included in the partition. Instant failure when a `Teardown_Notify` message is sent to a `Ground_Station`.

Obviously, this is the developer's fault. But this situation is very likely to arise under maintenance. How to resolve it? There are at least two solutions. When adding a new message:

1. Ensure that every Terminal `with`'s the new package, even if it doesn't actually use it.
2. For GLADE at least, explicitly include the new package in the partition definition for each Terminal.

At last we had a distributed version of TSim, with two partitions: a Satellite "server" and a Terminal "client," which could be deployed in a variety of configurations.

- Both Satellite and Terminal on the same machine, connecting to a TC application also on the same machine.
- Satellite on one machine, Terminal on a second machine, connecting to TC on a third machine.
- Satellite and Terminals on the same machine, connecting to one TC on the same machine, and other TC(s) on different machine(s).
- Satellite on one machine, with Terminal and TC on another machine.
- etc.

Throughout Round-3, we managed to retain the source code compatibility between Win32 and Linux. So, we could also build distributed TSim under Linux, and operate exactly the same way.[22]

Miraculously, we also found that the Satellite and Terminal partitions were interoperable under Win32 and Linux. That is, we could run the Satellite partition under Linux, and the Terminal partition under Win32, with TC connected on the same or a different machine. Or, we could run the Satellite partition on Win32, and the Terminal partition under Linux, connected to TC on Win32—possibly the same machine running the Satellite server. This was truly unexpected, and gave us something technically satisfying.

6. AFTERWORD

Everyone who has seen TSim has liked it. But no one has been willing to invest the additional resources needed to scale it up.

Ironically, TSim is languishing because TC has been so successful. TC was fielded, and has been operating extremely well for some time. There have been a few defects and some minor enhancements, but mainly, TC is a mature product well into its maintenance phase. Everyone is happy with it, and they see no compelling reason to develop a system simulator for testing a robust product.

Hopefully, the lessons learned on TSim will be applied to any development of a TC follow-on system.

7. REFERENCES

[1] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, second edition, 1998.

[2] M. Carlisle and W. B. Watkinson II. Rapid Ada portable interface design tool (RAPID). `wuarchive.wustl.edu/languages/ada/usafa/rapid`, Dec 2001.

[3] D. Chaplin et al. GLADE GTK+ user interface builder. `glade.gnome.org`, Aug 2002.

[4] EIA/TIA. EIA232E—interface between data terminal equipment and data circuit-terminating equipment employing serial binary data interchange. Standard, Electronic Industries Association, 1991.

[5] A. Europe. ACT europe – GLADE. `www.act-europe.fr`, Sep 2002.

[6] IEEE. IEEE standard for distributed interactive simulation. Standard IEEE-1278, Institute of Electrical and Electronics Engineers, 1998.

[7] Intermetrics. *Ada 95 Rationale, the Language, the Standard Libraries*. U.S. Government, 1995.

[8] J. Ousterhout. Tool command language (Tcl)/Tk. `www.tcl.tk/software/tcltk`, Sep 2002.

[9] J. Pfeifer. Ada for GNU/Linux team. `www.gnuada.org/alt.html`, May 2002.

[10] S. T. Taft and R. A. Duff. *Ada 95 Reference Manual: Language and Standard Libraries*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag Inc, New York, NY, USA, 1997. International Standard ISO/IEC 8652:1995(E).

[11] S. Tardieu. AdaSockets. `www.rfc1149.net/devel/adasockets`, Aug 2002.

[12] J. van Dijk. Jerry's Ada on Win32 page. `users.ncrvnet.nl/gmvdijk`, Mar 2002.

[13] D. Weller and S. Wright. The Ada95 Booch components. `www.pogner.demon.co.uk/components/bc`, Jun 2002.

[14] T. J. Westley. Tcl Ada shell (TASH), an Ada binding to Tcl/Tk. `www.adatcl.com`, Aug 2001.

APPENDIX

A. THE SYSTEM

Satellite communications systems provide wide-area connectivity when terrestrial resources, such as land lines or microwave towers, are either unavailable or impractical.

Ground-based terminals in the system serve as access points to the satellite, and provide communications among other terminals.

Figure 9 depicts these relationships in the system modeled by TSim; we'll call it TSCS (The Satellite Communications System).

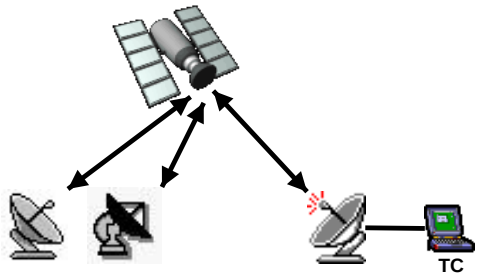


Figure 9: Satellite Communications System

In TSCS, there are several different kinds of terminals, some of which may be directly controlled by TC—more on the implications of this, later.

The main purpose of TSCS is to provide an IP (Internet

[22] Except that TC had to be on a separate machine, of course

Protocol) backbone among the terminals via PTP (Point-To-Point) links. End user devices may connect directly to the backbone through a terminal, but it is more common for a number of users to connect to a switch or router which is in turn connected to the terminal. A terminal may also serve as a gateway from the TSCS WAN (Wide Area Network) to, say, the internet.

Figure 10 shows a sample network topology under TSCS.

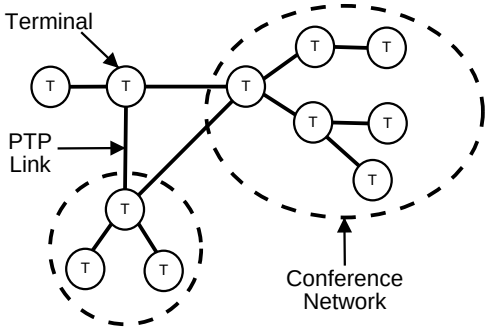


Figure 10: Example Network Topology

In addition to the PTP IP links between terminals, figure 10 also shows conference networks. Conference nets behave very much like a radio: when one person talks, everyone on the network can hear them, but typically only one person can talk at a time. Conference networks are normally used for coordination among terminals.

As any system administrator can tell you, setting up a WAN and managing the routing tables is a non-trivial task. In TSCS, this is made more difficult by the need to plan and coordinate the satellite accesses among terminals. For this, there is a software tool which the operator uses to plan and maintain the network. Some of the functions this tool provides are:

- Satellite accesses for each terminal
- PTP link configuration
- Conference network configuration
- Satellite antenna positioning
- Terminal communication plans

Using the Planning tool, the operator can assign satellite accesses to each terminal, completely define the desired links and networks, select the satellite antenna pointing locations for optimal terminal coverage, and then generate data loads for each participating terminal. With this communications plan in hand, each terminal operator follows the plan to provide the necessary connectivity.

Unfortunately, the plan is not static. The topology must be constantly adjusted in response to changing user needs, load balancing, planned outages, and equipment failures. The Planning tool can help the operator do all this, but there may be a lot of work involved in implementing the changes.

This is where TC comes in. The TC application connects directly to the terminal, and sends the commands neccessary to implement a communications plan—or replan.

Figure 11 illustrates how this all fits together.

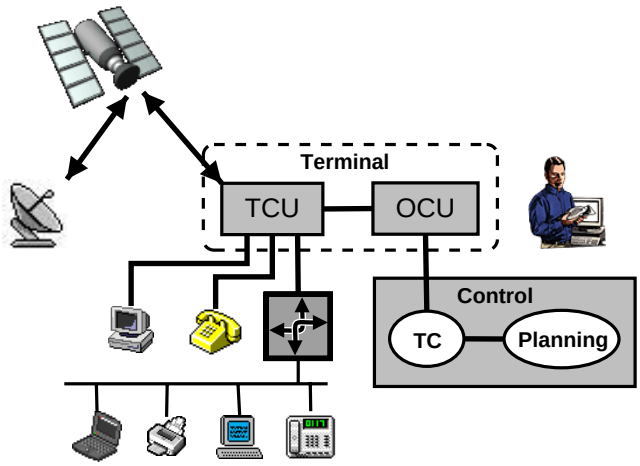


Figure 11: Terminal Operations

The terminal itself consists of two parts. The TCU (Terminal Communications Unit) provides the actual link to the satellite, and supports the connected baseband equipment, such as switches, routers, PC's, etc. The OCU (Operator Control Unit) is the operator's interface to the terminal. Through the OCU, the operator can command the terminal to logon to the satellite, establish networks, manage antennas, etc.

The Control PC is a separate laptop which hosts both the Planning tool and TC, and is connected to the OCU. The Planning tool produces the communications plan, which TC uses as its basis for commanding the terminal

In order to keep the network operating properly, TC uses the terminal to perform the following functions:

Antenna Control: This involves keeping the satellite antennas pointed at the planned locations, to optimize terminal coverage.

Network Control: Which includes activating, deactivating, modifying, and managing the membership of PTP links and conference nets.

Monitoring: Periodically determining the configuration of the network and the status of the participating terminals, to ensure that everything is operating according to plan, and there are no unexpected problems.

Although TC can execute a communications plan completely autonomously, the terminal operator normally oversees the process. The operator can control the degree of autonomy that TC exercises, and can manually execute terminal commands through TC. The combination of an automated communications plan, and direct operator control, makes TC a powerful network management tool.