Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

The Real-Time Simulation Protocol

Source Code Accompanies This Article. Download It Now.

- rtsp.txt
- rtsp.zip

The Real-Time Simulation Protocol is a package of C++ source code and tools that enables high-performance real-time distributed simulation across a TCP/IP network.

May 01, 2001
URL:http://www.drdobbs.com/cpp/the-real-time-simulation-protocol/184404611

Jim is a consulting electrical engineer and can be contacted at jim@ledin.com or http://www.ledin.com/.

Dynamic system simulation is an important tool in the development of a variety of complex embedded systems such as satellites, robotic systems, and military weapons. The benefits of effective simulation include reduced development costs, faster time to market, and enhanced product quality. As a result, there is a strong desire to expand the application of simulation to all aspects of the product development process and to employ simulation in other areas such as system testing and operator training.

One type of simulation that is seeing increasing use — particularly in defense-related work — is distributed simulation. In a distributed simulation, different elements of a complex system are simulated at separate locations and communicate among themselves over a network. A distributed simulation may operate either as a real-time system or in a nonreal-time manner. In real-time distributed simulation, the speed and reliability of network data transfer are critical. In nonreal-time distributed simulations, these factors are less important because delays do not affect the results and any lost data packets are retransmitted by the communication protocol. In this article, I'll focus on real-time distributed simulation.

In a distributed simulation, not all of the elements actually need to be simulations. It is common for a distributed simulation to include some simulated elements while other elements are actual systems that may be operating in the real world, or they may use input information generated from the simulations. I adopt the convention of referring to the separate systems or simulations in a distributed simulation as "federates." The entire distributed simulation is called a "federation."

A hardware-in-the-loop (HIL) simulation is a real-time simulation that incorporates some elements of the actual system into the overall simulation. A major benefit of distributed simulation is that you can connect a HIL simulation into a larger distributed simulation, even if the HIL simulation is located a large distance from the other members of the federation.

To illustrate, I'll describe a distributed simulation in use by the U.S. Navy for performing high-fidelity testing and operator training on shipboard weapons systems. I developed a communication protocol for this project that meets the specific needs of high-performance, real-time distributed simulation across a TCP/IP network. I will discuss this protocol, called the "Real-Time Simulation Protocol," in detail.

Virtual Missile Range

The Virtual Missile Range (VMR) is a U.S. Navy project underway at the Naval Air Warfare Center (Point Mugu, California). The goal of this project is to integrate simulation with actual shipboard systems to provide a relatively low-cost, high-fidelity environment for weapon systems testing and operator training. The first weapon system implemented in the VMR is the Sparrow missile system, which is deployed on a variety of ships and aircraft in military forces around the world. The specific configuration used for the initial VMR implementation is the Sparrow missile fired by a U.S. Navy Spruance-class destroyer.

The Sparrow missile is effective against many types of air and surface targets. The specific target simulated by the VMR is a sea-skimming cruise missile headed directly toward the ship. A radar signal simulator located on San Nicolas Island in the Pacific Ocean creates this simulated target signal. The target simulator receives radar signals transmitted by the ship and modifies them to simulate the target's apparent size (radar cross section), its distance from the ship, and its

speed. The target simulator then transmits the modified signals back to the ship, where the simulated radar echo creates a realistic representation of an incoming cruise missile.

The weapon systems operators on the ship track the simulated incoming target on radar consoles and launch a simulated Sparrow missile at the target. For the VMR, instead of launching an actual Sparrow missile, an inert missile is installed in the Sparrow launcher. This inert missile behaves like an actual missile during the power-up and launch sequence, which lets the weapons system operators and computers perform the missile selection and launch operations just as they would with an actual missile. Figure 1 shows the communication paths between the federates in the VMR.

The signals that provide initialization data to the missile prior to launch are monitored on the ship and this data is passed in real time to the Sparrow missile HIL simulation laboratory at Point Mugu. The HIL laboratory contains a flight simulation of the Sparrow missile that uses the actual missile target tracking radar, guidance computer, and autopilot. An RF signal simulator generates the signals that the missile radar system receives directly from the ship radar and from echoes off the target. The HIL simulation allows the missile to fly from prelaunch initialization through target intercept in a laboratory environment. The embedded software within the missile thinks that it is in flight and tracking an actual target during each simulation run. Figure 2 shows the major components of the Sparrow HIL simulation laboratory.

During each VMR run, the HIL missile simulation performs a launch and flight under command of the operators on the ship. The data from the target simulator is used in the HIL simulation in real time to determine the apparent position of the target in space. The HIL simulation flies the simulated target along a trajectory matching the one produced by the signal simulator on San Nicolas Island. After launch, the missile radar receiver acquires and tracks the simulated target. Within this simulated environment, the Sparrow flies out, homes in on the target, and detonates its warhead to destroy the target.

During VMR operation, a TCP/IP network connection exists between the ship, the target simulator, and the Sparrow missile HIL simulation laboratory. The ship communicates with a transceiver on San Nicolas Island over a wireless link that operates in the 2.4-GHz band. Communication between the island and Point Mugu takes place over an undersea fiber optic line. All communications across the network are encrypted for security.

IP Multicast

The network connecting the elements of the VMR uses the TCP/IP protocol. TCP/IP provides features that permit tradeoffs among attributes such as communication overhead, reliability of data transfer, and the network bandwidth required. I'll focus on the transport layer of the protocol stack, where there is a choice between the TCP and UDP protocols. This relationship is shown in Figure 3, where the transport layer lies below the application software level and above the internetwork level.

The Transmission Control Protocol (TCP) transport layer protocol provides a reliable communication mechanism between two applications across a network. TCP is reliable in the sense that the protocol detects the loss or corruption of data traveling across the network and it automatically requests retransmission from the sender. The delay caused by lost or corrupted data and subsequent retransmission causes the delivery of any data that arrives after the error occurs to stop until the retransmitted data arrives. After retransmission completes, any buffered incoming data is delivered to the recipient application in the correct order.

TCP requires the establishment of a connection prior to data transfer. The connection provides a context in which operations such as the detection and retransmission of lost packets occur. After data transfer completes, the connection must be closed.

The alternative to the TCP protocol is User Datagram Protocol (UDP). UDP is an unreliable protocol in the sense that it does not request retransmission of lost or corrupted packets. A corrupted data packet is discarded without notification to the sending or receiving application. It is not necessary to establish a connection when using UDP, so overhead related to establishing and closing connections is avoided.

Figure 4 shows the difference in behavior between UDP and TCP when a packet of data is lost. In Figure 4, federate updates are sent across the network at regular time intervals. At time *A*, a packet is lost. At time *B*, the packet following the lost packet arrives at the destination. After a timeout period, TCP recognizes that the first packet is missing and requests retransmission (time *C* in the figure). UDP, on the other hand, immediately delivers the new packet at time *B* to the destination application. At time *D*, the retransmitted data requested by TCP arrives at the destination. Finally, TCP delivers the packets from times *A* and *B* to the destination application.

As this example demonstrates, network data transfer problems can cause additional delays when using TCP that do not occur in UDP. The UDP model is preferable for a real-time distributed simulation because it minimizes the disruption that occurs when a data packet is lost. This assumes the packets contain parameter updates and the loss of any one packet just causes a delay of one update interval in the system's response.

In real-time distributed simulations, the bulk of data transfer normally consists of parameter updates transmitted by the various federates at a fixed rate. Each federate's update rate should be chosen to allow the federation to continue with minimal degradation in the event of an occasional network packet loss.

Standard UDP provides a communication mechanism between two applications across a network. UDP multicast adds the capability to transmit a message from one application to many other applications simultaneously. In distributed simulations, it is common for data transmitted by one federate to be used by more than one of the other federates. Multicast techniques can significantly reduce the network bandwidth requirements in this situation.

It is a tedious, error-prone process to write software to implement IP multicast for each federate in a distributed simulation, particularly if there are system differences such as byte ordering. To address these issues and to ensure maximum real-time performance, I developed the Real-Time Simulation Protocol. Although initially used for the VMR project, this tool is useful for a variety of high-performance real-time distributed applications.

RTSP

The Real-Time Simulation Protocol (RTSP) is a package of C++ source code and tools that enables high-performance real-time distributed simulation across a TCP/IP network. The RTSP package is available electronically; see "Resource Center," page 5. The RTSP design has two primary goals: maximize real-time performance and provide ease of use across a variety of computing platforms. The goal of maximizing real-time performance is satisfied through the use of the UDP multicast protocol and by performing as much work as possible during the compilation and program initialization phases, thus minimizing the computation required during real-time operation. Ease of use is provided through the use of the Message Definition File (MDF) format, which specifies the contents and other information about each type of data message. Additional techniques that increase the ease of use include automatic C++ code generation by the MDF translator and automatic handling of cross-platform issues such as byte ordering and the avoidance of holes in data structures.

For best simulation fidelity, it is desirable to minimize the effects of communication latency by extrapolating the value of continuous variables to the current time in the receiving application. This presents a problem because some method must be available for determining the elapsed time between the initiation of message transmission and the current time. If the federates are all running on the same computer, the solution is simple because all federates use the same system clock. However, if the federates are running on separate computers (possibly thousands of miles apart) it may be impossible to synchronize computer system clocks to the required level (typically milliseconds).

One way to achieve this synchronization is to install a Global Positioning System (GPS) receiver interface in each computer in the federation. As a by-product of the GPS position measurement, the receiver determines the time very precisely. RTSP supports the use of GPS time (or another precision time reference) to measure communication latency for use in data extrapolation. However, the use of a precision time reference is optional in RTSP.

The RTSP run-time software also provides the capability to log all messages received by a federate to disk as they arrive. Time of receipt for each message is stored in the log file so that detailed studies of communication latency can be performed during postrun analysis. Of course, this message logging facility detracts from the real-time performance of a simulation, so in many cases it makes more sense to create a separate federate that does nothing but receive network traffic and log it to disk. This data collector federate places no additional burden on the real-time network communication since it does not transmit any messages.

The Message Definition File (MDF) describes the data messages transferred across the network and which of the federates send and receive each type of message. The MDF syntax is similar to the C language elements used to describe data structures. A translator program reads the MDF file and converts it into a set of C++ header files and a single C++ source file containing static data definitions describing the federation. The simulation developer uses these files along with the provided RTSP run-time source code to build each member federate of the federation.

A generic Win32 federation controller program is included with the RTSP distribution. This program reads a federation MDF and provides a set of buttons that allow the user to send command messages to the federates. The controller displays the response received from each federate following each transmission of a command message. The available commands are:

- *Ping.* Checks connectivity and times the round trip latency to each federate.
- *Initialize*. Commands all federates to prepare for a simulation run.
- *Start*. Starts real-time execution immediately.
- *Halt*. Stops real-time execution immediately.

The RTSP software has been tested on several different platforms including PCs (Windows 98/NT 4.0), UNIX (Solaris), and WindRiver's VxWorks real-time operating system on a Power PC processor.

On a low-traffic LAN using PCs and standard Ethernet hardware, one-way communication latencies of 1 millisecond or less are typical with RTSP. RTSP also makes efficient use of system resources: On a 400-MHz PC running three simulations where each simulation sends 100 updates per second, the CPU usage is in the 1-2 percent range.

Example RTSP Federation

This example describes the development of a complete distributed simulation using RTSP. Figure 5 shows the system to be modeled is a continuous plant with a continuous controller and a command generator that drives the controller. The plant is a pure inertia system with the transfer function 1/*s*2. The controller is implemented with the transfer function 500(*s*+2)/*s*+40. The command generator outputs a randomly selected command in the range (0,10) at 10-second intervals.

We will implement this federation using three federates plus a federation controller:

1. The *CommandGenerator* federate creates and outputs a new command to the system every 10 seconds.

2. The *Controller* federate computes the error between the command value and the current system output, and passes the resulting error signal through the *Controller* compensator. The output of the *Controller* compensator drives the plant.

3. The *Plant* federate accepts the *Controller* output signal as its input and uses it to drive the plant transfer function.

4. The generic RTSP controller application commands the operation of the entire federation.

The first step is to develop an MDF file (see Listing One) that describes the federates and the messages that pass between them. It contains a message definition for each of the lines that pass through the dotted borders in Figure 5.

We then run the MDF translator program with example.mdf as its input. Listing Two shows the execution of the translation. Each of the source files generated by the translator is named here. Each ".h" file is a complete C++ class definition that allows a federate to send or receive the messages described for it in the MDF file.

Listing Three is CommandGeneratorSend.h, one of the files generated by the translation. The *command* member of the *CommandGenerator.Update* message defined in Listing One appears here in the *Update* structure, along with several parameters used internally by RTSP. Observe that the RTSP parameter names all begin with underscores. The MDF syntax rules prevent the user from defining message elements that start with underscores, so there is no chance of a name collision.

The other generated header files have similar structures. _MsgData.cpp is used internally by the RTSP software and not generally of interest to simulation developers, except perhaps when troubleshooting.

Next, we must develop the main program for each federate. Listing Four is the source code for the CommandGenerator program. Several callback functions are defined before the main function definition. These callbacks execute in response to commands sent by the RTSP controller application. In this example, the callbacks simply print messages, except that the *Halt* callback also sets a flag to stop the simulation execution. The simulation loop runs at a rate of 100 frames per second.

The remaining federate programs have a similar structure but are more complex due to the additional modeling required. We compile the three federate simulations and build an executable for each of them using the generated _MsgData.cpp file and the provided RTSP.cpp source file. The result is three executable programs that can be run on the same machine or on different machines connected by a TCP/IP network.

Having built these three applications, we can now execute them as a federation along with a federation controller application. To test the federation, we begin by running everything on a single computer. On a Windows NT system, we can run the three program executables plus the generic controller application from a batch file like Listing Five, which assumes that all the executable programs and the DistSim.mdf file are located in the current directory.

Once all four programs are running, the three federate names will appear in the RTSP controller's dialog as in Figure 6. Click the Ping button to verify that all federates respond. Then click Initialize and verify that all federates report successful initialization. Click Start to start real-time execution. The plant and controller transmit Update messages at a 100-Hz rate and display onscreen information at a 1-Hz rate. The *CommandGenerator* transmits a new command every 10 seconds and displays each new command as it is transmitted.

To distribute this simulation across a network, simply terminate one of the federate applications and run the same executable on a different computer system on the network. Once the federate is running on the other computer, click Ping to verify connectivity and then perform a simulation run as described earlier.

You should be able to compile the source code for these federates with minimal (or no) changes for execution on a UNIX system. Execute one or more of the federates on the UNIX system and repeat the aforementioned steps to run the simulation.

This simple federation has not exercised many of the more sophisticated capabilities available with RTSP. Some of the areas you may wish to explore further are:

- Perform logging of message traffic to disk during simulation execution. Develop an application to read the log file and analyze it after the simulation run completes.
- Synchronize the time on multiple computers using GPS or some other precision timing reference.

- Extrapolate continuous variables to estimate their value at the current time. Either all the federates must execute on a single computer or a distributed precision time reference must be used.
- Perform real-time collection and post-run display of information about sequences of lost messages. This indicates if there are problems with network speed or reliability.
- Develop a real-time viewer application to display the state of the entire federation by monitoring messages passed between the federates. This requires that a new federate be added to the MDF file that subscribes to messages from all federates.

DDJ

Listing One

```
// Distributed Simulation Message Definition
// Declare names for all federates:
federate CommandGenerator, Controller, Plant;

// Define messages. The federate named with each message is the sender.
message CommandGenerator.Update
{
    double command;
};
message Controller.Update
{
    double drive;
};
message Plant.Update
{
    double output;
};
// Subscription list. Each federate can subscribe to (i.e., receive) any
// message including ones it produces.

// The Controller gets updates from the CommandGenerator and Plant
subscribe Controller : CommandGenerator.Update, Plant.Update;

// The Plant gets updates from the Controller
subscribe Plant : Controller.Update;
```

Back to Article

Listing Two

```
C:\Projects\DistSim>translator DistSim.mdf
Generating CommandGeneratorSend.h
Generating ControllerSend.h
Generating ControllerRcv.h
Generating PlantSend.h
Generating PlantRcv.h
Generating _MsgData.cpp

C:\Projects\DistSim>
```

Back to Article

Listing Three

```
// CommandGeneratorSend class
// Generated Thu Aug 24 13:33:36 2000 from distsim.mdf
// *** This file was automatically generated. Do not edit it! ***
#include <RTSP.h>
#include <RTSP_Internals.h>
class CommandGeneratorSend
{
public:
    struct Update
    {
        double command;
        uint _TimeTag;
        ushort _SequenceNum;
        uchar _ReceiverIndex;
        uchar _SenderIndex;
        bool Send(RTSP& rtsp, uchar rcv_index = _AllIndexes)
            { return _SendMessage(rtsp, this, 0, rcv_index); }
    };
};
```

Back to Article

Listing Four

```
#include <RTSP.h>
#include <stdio.h>
#include "CommandGeneratorSend.h"

RTSP* rtsp;

CommandGeneratorSend::Update command_update;

bool ping_callback()
{
    printf("Ping callback\n");
    return true;
}
bool init_callback()
{
    printf("Initialize callback\n");
    return true;
}
bool start_callback()
{
    printf("Start callback\n");
    return true;
}
bool end_run = false;
bool halt_callback()
{
    printf("Halt callback\n");
    end_run = true;
    return true;
}
int main()
{
    printf("CommandGenerator Federate\n");
    printf("Jim Ledin    August, 2000\n\n");

    // Determine the number of frames per second
    int frames_per_sec = 100;
    printf("\nUpdate rate: %d frames/sec\n\n", frames_per_sec);

    rtsp = new RTSP("CommandGenerator", 0);
    rtsp->SetPingCallback(ping_callback);
    rtsp->SetInitializeCallback(init_callback);
    rtsp->SetStartCallback(start_callback);
    rtsp->SetHaltCallback(halt_callback);

    for (;;)
    {
        end_run = false;
        printf("Waiting for Initialize\n");
        rtsp->WaitForInit();

        printf("Waiting for Start\n");
        rtsp->WaitForStart();
        printf("Running\n");

        double step_time = 1.0 / frames_per_sec;

        rtsp->StartTimer();
        uint frame = 0;
        while (!end_run)
        {
            // Time since start of run
            double Time = frame * step_time;

            // Output a new random command every 10 seconds
            if (frame % (10*frames_per_sec) == 0)
            {
                double command = 10.0 * double(rand()) / RAND_MAX;
                command_update.command = command;
                command_update.Send(*rtsp);

                printf("Time: %.1lf; New command: %.3lf\n", Time, command);
                fflush(stdout);
            }
            uint end_of_frame = uint(1e6 * ++frame * step_time);
            uint cur_time = rtsp->ReadTimer();
            while (cur_time < end_of_frame)
            {
                rtsp->PollRcv(true, end_of_frame - cur_time);
                cur_time = rtsp->ReadTimer();
```

```
            }
        }
        printf("End of run\n");
    }
    return 0;
}
```

Back to Article

Listing Five

```
start controller DistSim.mdf
start CommandGenerator
start Controller
start Plant
```

Back to Article

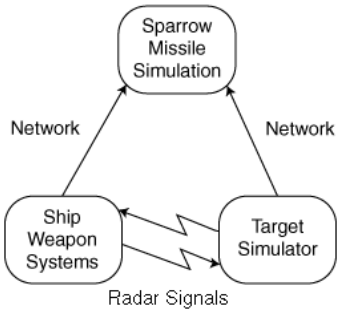

Figure 1: Virtual missile range.

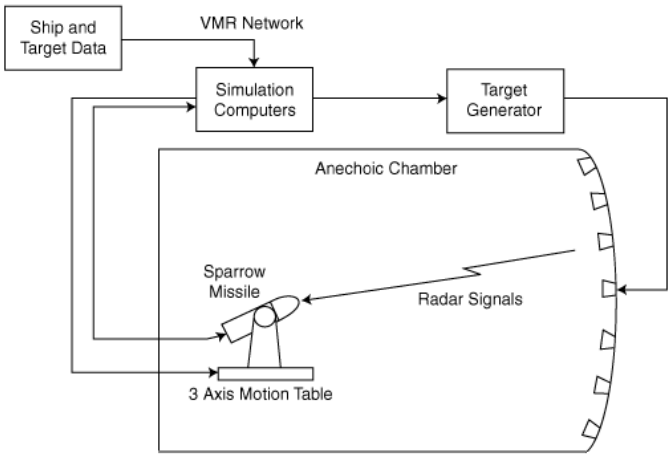

Figure 2: HIL simulation.

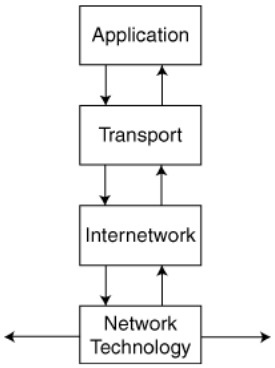

Figure 3: TCP/IP protocol layers.

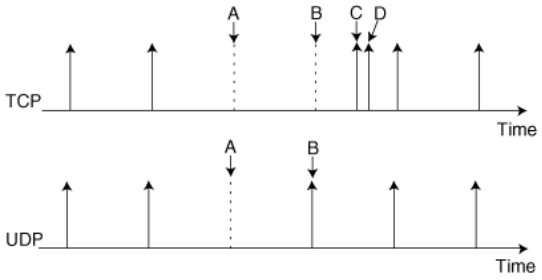

Figure 4: TCP versus UDP lost packet handling.

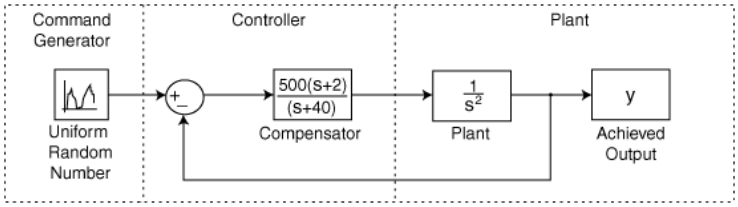

Figure 5: Example dynamic system.

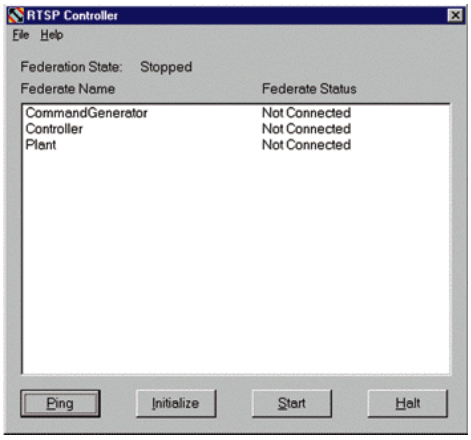

Figure 6: RTSP controller application.