TP006: An Introduction to RealiNet

Introduction

The aim of this document is to introduce the RealiNet networking utilities provided by the RealiMation libraries. Two examples of networking shall be presented: a socket based reader-writer scenario, and a RealiMation client-server system. These shall present the two levels of networking support that the RealiMation libraries provide. These examples shall also show that networking between machines of differing architectures (such as Intel to SGI) can be achieved with little extra effort. This document assumes a basic familiarity with socket based communication and the ability to compile and run the example programs on the target machines.

RealiNet Socket Based Communication

The `rnu` library provides a number of convenient utilities to make socket based communication simpler. The API Programmer Reference Guide provides a complete list of the `RN*` commands and `rn*` structures that are made available. These utilities are independent of the other RealiMation libraries and may therefore be used for non-graphics based application if desired.

Some of the RealiNet socket functions are utilised in the simple reader-writer example provided below. This example uses a “reader” program to listen to a specified port for socket connections. A second program, the “writer”, sends data over a socket to a named machine using a specified port number. The data is sent once a successful socket connection between the writer and reader has been achieved.

The basic steps of setting up socket based communication are:

1. Initialise the socket library using `RNInitialise()`.
2. Create an empty network by calling `RNOpenNetwork()`.
3. Create a socket on the network to listen to a specific port, `RNOpenSocket()`.
4. Send and receive data over the socket using calls to `RNSendSocket()` and `RNReceiveSocket()` respectively.
5. Close the socket using `RNCloseSocket()`.
6. Close the network using `RNCloseNetwork()`.
7. Exit the socket library using `RNExit()`.

These basic steps, outlined in the source code example below, are common for both the reader and writer applications. As such, similar source can be used for both variants of the example. The differences between the compiled code are determined by the `READER` and `WRITER` definitions and determined using `#ifdef` statements. It is advised that you add these to the project setting definitions to produce two separate executables.

```
/*——————Start of example code——————————*/

/************************************************************************/
/*** This example code is desgined to produce two separate executables.  ***/
/*** The first listens to the socket and reads a message. This is termed ***/
/*** the ‘reader’. The second executable, the ‘writer’, sends a message  ***/
/*** over the socket.                                                    ***/
/*** It is suggested that port 4999 is used.                             ***/
/*** Each application uses command line aguments to specify the          ***/
/*** address of the reader and the port number to use.                   ***/
/*** Typical usage is as follows:                                        ***/
/***    reader 4999                                                      ***/
/***    writer 4999 remote.server.com                                    ***/
/************************************************************************/

#include <stdio.h>

#include “rnu.h”
```

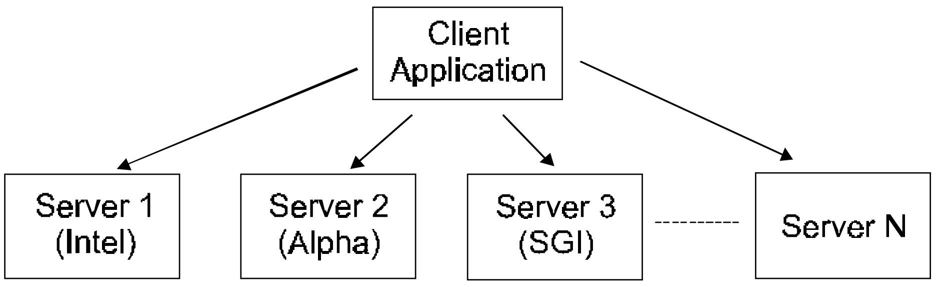
```
#if defined (WIN32)
#if defined(_DEBUG)
#pragma comment(lib, “rnu4d.lib”)
#else
#pragma comment(lib, “rnu4.lib”)
#endif
#endif

#define MESSAGE          “Welcome to the wonderful world of RealiMation”
#define MESSAGE_LENGTH   46

void main(int argc, char *argv[])
{
   rnError    error;
   rnNetwork *network;
   rnSocket  *socket;
   char       *this_machine;
   UShort     port = (UShort)atoi(argv[1]);
#ifdef READER
   rnHeader  header;
   char       buffer[MESSAGE_LENGTH];
#endif

   /*** Step 1 ***/
   RNInitialise(&this_machine);
   printf(“%s: This machine = %s\n”, argv[0], this_machine);

   /*** Step 2 ***/
   printf(“Opening network”);
   error = RNOpenNetwork(0, &network);
   printf(“, error=%d\n”, error);

   if ( error == RNNoError )
   {
      /*** Step 3 ***/
#ifdef READER
      printf(“Opening socket to port %d”, port);
      error = RNOpenSocket(network, NULL, port, &socket);
#elif WRITER
      printf(“Opening socket to %s on port %d”, argv[2], port);
      error = RNOpenSocket(network, argv[2], port, &socket);
#endif
      printf(“, error=%d\n”, error);

      if ( error == RNNoError )
      {
         /*** Step 4 ***/
#ifdef READER
         printf(“Receiving message (expect %d bytes)”, MESSAGE_LENGTH);
         error = RNReceiveSocket(socket, &header, MESSAGE_LENGTH, &buffer);
         printf(“, error=%d\n”, error);
         printf(“Message = \”%s\”\n”, buffer);
#elif WRITER
         printf(“Sending message (contains %d bytes)”, MESSAGE_LENGTH);
         error = RNSendSocket(socket, 0, MESSAGE_LENGTH, MESSAGE);
         printf(“, error=%d\n”, error);
         printf(“Message = \”%s\”\n”, MESSAGE);
#endif

         /*** Step 5 ***/
         printf(“Closing socket\n”);
         RNCloseSocket(network, socket);
      }

      /*** Step 6 ***/
      printf(“Closing network\n”);
      RNCloseNetwork(network);
   }

   /*** Step 7 ***/
   RNExit();
}
/*————————End of example code————————*/
```

The above code assumes that the reader is executing on the target machine before the writer is run, otherwise the writer will be unable to make a connection and would execute prematurely. The writer code could be simply modified to wait for a connection with the inclusion of a `while` loop over the call to `RNOpenSocket()`.

The above example has presented the most fundamental level of socket communication provided by the `rnu` library. Other functions provided by the library also allow for buffered communication, packet based message passing, and the transmission of typed data (such as `Dword`, `Ord`, etc) between machines of differing architecture. This latter facility means that values can be passed between machines with big and little endian byte orderings transparently. An example of where this may be used is the passing of `Ord` values between an Intel/Alpha (MS Windows) based machine and a MIPS (SGI Irix) based machine.

The RealiMation Client-Server Model

The RealiMation libraries provide a further level of abstraction, above that of the `rnu` library, for setting up a network of machines to act as graphics servers responding to the requests of a remote client application. At this level the programmer need not be concerned with low level socket initialisation, protocol definitions or data packet transmission.



Each of the servers indicated above may provide a number of channels to which views may be associated. The RealiMation package provides a server application for each of the supported architectures. This is a command line application which services the requests of a client application. These requests take the form of `RT*` commands, as listed in the API Programmer Reference Guide. For example, once a connection has been established between a client and a server, a call to `RTLoadRealiBase()` will be executed by both the client application and the server. The client application may reside on a host machine with no graphics capability where channels and views are realised on a server machine.

Since the RealiMation client-server model is built upon the endian independent socket layer utilities, a RealiMation server may reside on a different architecture machine to that of the client application. For example, an Intel (Windows) based application may generate graphical data which is visualised, in real time, on an SGI Onyx2. Or, conversely, an SGI client application may utilise graphics devices, attached to PC servers, to provide economical multiple screen solutions.

As with most client-server models, a particular port is reserved for RealiNet applications. At present all RealiMation client-server communication takes place over sockets connected to port 4999. As stated above, the lower level `rnu` socket utilities may use any user defined port.

Modifying Existing Applications for RealiNet

The example provided below shows a simple client application which connects to a server. The client application does not realise any channels locally. As such all graphical output is produced by the server. This example is a modified version of the code given in Technical Application Paper 005 - A Simple Command Line Application. Only a few simple changes have been made which are indicated by the `#ifdef REALINET` regions of the code (see highlighted regions). These occur at the following stages:

Step 1: Initialise the RealiMation Libraries

Each server connection should be made as early as possible in the client application, ideally directly after a call to `RTInitialise()`. This ensures that subsequent operations are mirrored correctly on the client and server applications with objects being assigned the same ID on each side of the connection. This will ensure consistent object referencing, particularly when a new RealiBase is loaded.

A server object is created using a call to `RTCreateID()` and is initialised with the server's Internet domain address. This address must match exactly the server name displayed on executing the server. Finally a connection is made with the server on calling `RTRealise()`:

```
RTCreateID(RTServerID);
RTSetServer(ServerID, &server);
RTRealise(ServerID);
```

Step 3: Create a Graphics Channel

So that a channel is created at the server, and all subsequent commands for that channel are sent to the server, a new channel should be associated with the required server:

```
RTAddObject(ServerID, ChannelID);
```

Step 5: Enter a Basic Display Loop

The TCP/IP network layer, and the RealiMation packet implementation, may buffer data being sent between the client and server. This buffering ensures optimal use of network bandwidth. However, within the display loop it is advisable to ensure that all packets have been physically sent to the server before the next buffer is drawn. It is therefore advised that the network buffer is flushed and the server synchronised with the client:

```
RTFlush();
RTSync();
```

Failure to ensure this will be observed with apparently jerky motion as the server receives a number of buffered page swaps occurring at the wrong time intervals.

Step 6: Close Down Gracefully

To close the socket connection with the server the server should be unrealised:

```
RTUnrealise(ServerID);
```

It should also be noted that instead of linking to `rmation4.dll` the RealiNet version of the application instead uses `rnet4.dll` (or the debug equivalent where applicable).

Running the Server

The RealiMation package includes a server for each of the supported architectures, with debug and release variants. Each server is a command line application which can optionally take a number of arguments. Typically the server is set running before any client applications are executed. To run a server type `rserver4`. The name of the current machine is now displayed. It is this name that a client application must use to achieve a correct connection. Once a connection with a client has been made, various informational messages are displayed. When a client terminates the connection the server will wait for further clients to connect. Note that only one server is permitted on a single machine.

The RealiNet Client Example

```
/*———————Start of example code———————————*/
/*— Initial definitions —*/

/* Load standard include files */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

/* Load RealiMation specific include files */
#include “rmation.h”

#if defined(_RMDEBUG)
#include “rmdebug.h”
#endif

/* Set directives for external library loading */
#if defined (WIN32)
#if defined(_DEBUG)
#ifdef REALINET
#pragma comment(lib, “rnet4d.lib”)
#else
#pragma comment(lib, “rmation4d.lib”)
#endif
#pragma comment(lib, “rfu4d.lib”)
#pragma comment(lib, “rmm4d.lib”)
#pragma comment(lib, “rmu4d.lib”)
#pragma comment(lib, “rmdebug.lib”)
#else
#ifdef REALINET
#pragma comment(lib, “rnet4.lib”)
#else
#pragma comment(lib, “rmation4.lib”)
#endif
#pragma comment(lib, “rfu4.lib”)
#pragma comment(lib, “rmm4.lib”)
#pragma comment(lib, “rmu4.lib”)
#endif
#endif

/* Specify the display driver for the server to use    */
/* In this case the OpenGL driver. Note that the       */
/* driver depends upon whether this is the Debug or    */
/* Release version of the application.                 */
#if defined(_RMDEBUG)
#define DRIVER “rgdgl4d.dll”
#else
#define DRIVER “rgdgl4.dll”
#endif

/* Specify the RealiBase to load */
#define REALIBASE “helisim.rbs”

/* Specify the server address to connect to */
#ifdef REALINET
#define SERVER “remote.server.com”
#endif

/* Specify the characteristics of the display device */
#define XRES 640
#define YRES 480
#define PIXELDEPTH 16
#define REFRESH 60

/* Specify the position of the view relative to top left */
#define XPOS 0
#define YPOS 0

/* Function prototypes */
void InqError(char *msg);
rtError imageCB (rtImageCBType type, rtID Id,
                 const char *fname, const char *path,
                 rtImage *info);
```

```
/* The main function definition */
void main(void)
{
    rtChannel cInfo;
    DWord key;
    DWord terminate = 0;
    Ord t, t0;

    /*————————————————————————————*/
    /*— Step 1 : Initialise RealiMation libraries —*/
    /*————————————————————————————*/
    RTInitialise();
    InqError("RTInitialise()");
```

```
#ifdef REALINET
    rtServer server;
    rtID ServerID = RTCreateID(RTServerID);
    InqError("RTCreateID(ServerID)");

    server.name = SERVER;

    RTSetServer(ServerID, &server);
    InqError("RTSetServer(ServerID, &server)");

    RTRealise(ServerID);
    InqError("RTRealise(ServerID)");
#endif
```

```
    /*————————————————————————————*/
    /*— Step 2 : Load the RealiBase                  —*/
    /*————————————————————————————*/
    RTSetImageCallback(imageCB);
    InqError("RTSetImageCallback(imageCB)");
    RTLoadRealiBase(REALIBASE, 0);
    InqError("RTLoadRealiBase(REALIBASE, 0)");

    /*————————————————————————————*/
    /*— Step 3 : Create a graphics channel         ——*/
    /*————————————————————————————*/
    rtID ChannelID = RTCreateID(RTChannelID);
    InqError("RTCreateID(RTChannelID)");
```

```
#ifdef REALINET
    RTAddObject(ServerID, ChannelID);
    InqError("RTAddObject(ServerID, ChannelID)");
#endif
```

```
    cInfo.name = DRIVER;
    cInfo.res.width = XRES;
    cInfo.res.height = YRES;
    cInfo.res.depth = PIXELDEPTH;
    cInfo.res.refresh = REFRESH;
    cInfo.auto_load = 0;
    cInfo.n = 0;
    cInfo.data = NULL;

    RTSetChannel(ChannelID, &cInfo);
    InqError("RTSetChannel(ChannelID, &cInfo)");
    RTRealise(ChannelID);
    InqError("RTRealise(ChannelID)");

    /*————————————————————————————*/
    /*— Step 4 : Add a predefined view to channel —*/
    /*————————————————————————————*/
    rtID ViewID = RTGetNextID(RTNullID, RTViewID);
    InqError("RTGetNextID(RTNullID, RTViewID)");

    RTSetViewPortSize(ViewID, XRES,YRES);
    InqError("RTSetViewPortSize(ViewID, XRES, YRES)");
    RTSetViewPortPos(ViewID, XPOS, YPOS);
    InqError("RTSetViewPortPos(ViewID, XPOS, YPOS)");
    RTAddObject(ChannelID, ViewID);
    InqError("RTAddObject(ChannelID, ViewID)");
    RTRealise(ViewID);
    InqError("RTRealise(ViewID)");

    /*————————————————————————————*/
    /*— Step 5 : Display Loop                          —*/
    /*————————————————————————————*/
    t0 = RTInqClock();
    while ( !terminate )
    {
#ifdef REALINET
        RTFlush();
#endif
        t = RTInqClock();
        RTSetViewTime(ViewID, t-t0);
        RTDisplayView(ViewID);
        RTSwapPage(ChannelID);
#ifdef REALINET
        RTSync();
#endif

        while ( kbhit() )
            key = getch();
        if ( key == 0x1B )       // Check for Escape key hit
            terminate = 1;
    }

    /*————————————————————————————*/
    /*— Step 6 : Close down                            —*/
    /*————————————————————————————*/
    RTUnrealise(ViewID);
    InqError("RTUnrealise(ViewID)");
    RTUnrealise(ChannelID);
    InqError("RTUnrealise(ChannelID)");
#ifdef REALINET
    RTUnrealise(ServerID);
    InqError("RTUnrealise(ServerID)");
#endif

    RTShutDown();
    InqError("RTShutDown()");
}

/*————————————————————————————————*/

void InqError(char *msg)
{
    /* Function to enquire the RealiMation error status */
    /* and report an error string based upon the        */
    /* supplied message. This is only enabled for the   */
    /* Debug version of the application.                */

    rtError err = RTInqLastError(0);
#if defined(_RMDEBUG)
    char errmsg[32];
    RTInqErrorString(err, errmsg);
    fprintf(stderr, "%s : %s\n", msg, errmsg);
#endif
    if ( err != RTNoError )
        exit(1);
}

/*————————————————————————————————*/

rtError imageCB (rtImageCBType type, rtID Id,
                 const char *fname, const char *path,
                 rtImage *info)
{
    return RTNoError;
}

/*—————————End of example code—————————*/
```

Further RealiNet Examples

The two examples above have presented simple applications which make use of the RealiNet functions provided by the RealiMation libraries. Many more advanced features are provided by the libraries, some of which may be seen in the code samples provided in the RealiMation package. Of particular interest may be the RealiGame example which realises up to nine views on servers using RealiNet. The source of the server application is also available which shows how a buffered, packet based protocol, message system may be used for communication between clients and servers.

Summary

This document has shown how to utilise the socket based communication utilities provided by RealiMation. The detailed examples have also shown how to write simple, platform independent, application that communicate over a network. The later of these examples introduced a client-server application for the remote rendering of channel information.