Lesson 10
Introduction to PS3 Networking

Playstation 3 Development

Sam Serrels and Benjamin Kenwright[1]

Abstract
A beginners guide to the theory behind digital networking with reference to writing network code for the Sony Playstation 3. An example of using the Sony Http library to request and process a web page from the internet is also covered.

Keywords
Sony, PS3, PlayStation, Setup, Windows, Target Manager, ELF, PPU, SPU, Programming, ProDG, Visual Studio

[1] *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

Contents

1 Introduction to networking **1**
1.1 The Networking protocol layers 1
1.2 Transport protocols 2
1.3 Sockets 2
2 Playstation 3 specific Networking **3**
2.1 Playstation 3 Network Libraries 3
2.2 Network setup 3
2.3 Network Test Code 3
2.4 HTTP fetch Code 4
3 Conclusion **5**
References **5**

Preface

About the Edinburgh Napier University Game Technology Playstation 3 Development Lessons Edinburgh Napier University Game Technology Lab is one of the leading game teaching and research groups in the UK - offering students cutting edge facilities that include Sony's commercial development kits. Furthermore, within the Edinburgh Napier Game Technology group are experienced developers to assist those students aspiring to releasing their own games for PlayStation. Students have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [2].

Previous Tutorials This tutorial assumes you have read the tutorial on compiling and deploying applications to the PS3.

1. Introduction to networking

Networking, the process of communicating between different systems, is the heart of multilayer gaming. Technicality, networking is intertwined into nearly every layer of a your game. At the lowest level, queues, callback, events and driver calls operate the most technical aspects of the networking protocals. At a middle level you game must know how to prioritise, manage and predict network packets to smooth out any irregularities in the transmission.
At the highest level, the aspects of your game that the player will see and use, you will need to expose some aspects of networking here also. Tasks such as finding a game to join is a complex task behind the scenes, any must be simplified to be usable by the games player.

The packet Once you have data that you wish to send over the network, it goes through a process of being *"bundled up"* with various bits of header data required for transmissions. this *bundle* of data and headers is called a **packet**.

1.1 The Networking protocol layers

The fundamental concept of data transmission is that the protocols are divided up into **layers**. Your data starts at the topmost layer, your application, from then on it goes down the structure, getting padded with different protocol headers as the packed is built. It's important to note which of these layers you have control over and need to manage.

Link layer
E.g. Ethernet
How the data is sent down cables/over wifi. Handled exclusively by the networking hardware.

Internet layer
E.g. IP addresses
Where the data packet is going, and other routing info. Handled mostly by low level subsystems in the operating system, and by network routers.

Transport layer
E.g. TCP, UDP, RUDP
The overall layout of the entire packet. Standards can have different lengths for headers and different encoding of data. In addition to the layout of data packets it also

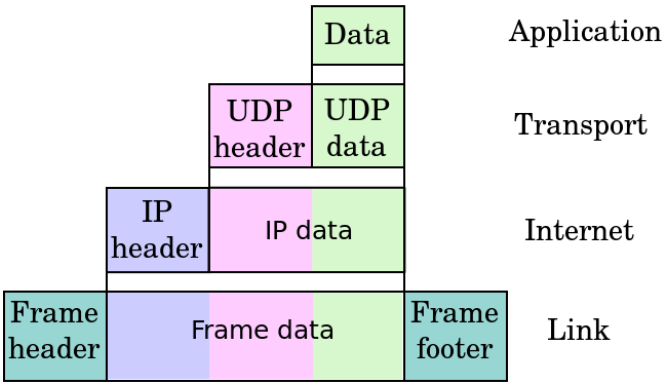

Figure 1. Network Layers - Application data descending through the networking protocol layers. Via Wikipedia[1]

defines functional packets which convey info like ”packet received”, ”packet lost” and ”connection established”

Application layer

E.g. HTTP, FTP, IRC, SSH, TELNET, IMAP

This is the format of the actual data within the packet once all the other headers have been removed. This is handled by applications, like a web browser or an email client. You can define your own protocol if you are only communicating between your own applications. There is no largely accepted standard for multilayer game data, as network overhead bottlenecks force most games to ”roll their own”.

1.2 Transport protocols

The Transport layers bring the first major factors that we have to interact with when sending data. They dictate most of the important data that we need to care about in the packet, and how the ”conversation” between two machines is structured.

TCP The most common protocol is The Transmission Control Protocol. TCP insures that your data will get to where it needs to go, in the right order and without errors. This reliability comes at a cost of a large overhead as the TCP packet headers need to be quite large (20 bytes). In the case a packet doesn't reach it's destination (referred to as ”dropped”), then it will be retransmitted which can cause slowdowns.
TCP is **Slow and Accurate**

UDP The User Datagram Protocol is the speedy but unreliable alternate to TCP. Data is not guaranteed to arrive in the right order, or even arrive at all. There is minimal error checking built into the header, the receiving machine can detect errors in the data, if one is found, the packet is discarded rather than retransmitted. The advantage of forgoing the guarantees of TCP, is a lower overhead(8 bytes), which means a faster transmission. This model is perfect for fast updating games, as data will be sent very frequently, so if one packet is dropped, the next one is already on its way. Don't be mistaken into thinking UDP is faulty, under good network conditions, it can be almost as reliable as tcp. Packets can get delayed or dropped when navigating large and busy networks (e.g. *The Internet*), using TCP would require every dropped packet to be resent, which would not be good for most game data.
UDP is **Fast and unreliable, perfect for non-critical data.**

RUDP Reliable User Datagram Protocol is UDP with extra data bits squashed in to add more reliability, without having the full overhead of TCP. The actual packets are just plain UDP packets, with the data section being shared by application data and the RUDP data bits. This means that you can send the RUDP packets easily, the issues are with the decoding of the data.
RDP is a protocol that has been around since 1999 but is not currently a formal standard. This means that there a few different RUDP libraries, that may all do the same thing, but are not guaranteed to be compatible with each other. The PS3 has a Sony RUDP library, and if you are just communicating between PS3s there will be no problem. If you need to send RUDP packets to a windows machine, then you will have to make sure your windows RUDP library is compatible with the PS3 implementation. This is the pain of non standardised protocols. Standardisation aside, RUDP is really useful. The Sony library provides data received acknowledgements, packet prioritisation, data statistics and congestion control. RUDP seems almost purpose built for multilayer gaming (in fact, it probably was).
RUDP is **UDP with extra features, still much faster than TCP but slightly slower UDP**

1.3 Sockets

To send data to another machine, you need a socket to ”send” the data to (imagine a mailbox). You don't care how the packet gets to the socket, that's the job of networks and routers (the mail service). Due to the speed of electronic transmission (compared to our post analogy), we can get useful data back like confirmation that the data arrived, or when the socket ”closes”.

Socket implementations The exact concept of a socket isn't part of any of the communication standards, it's a concept that network libraries use to implement the transport standards. The Windows socket implementation (from the Microsoft winsock library) has many differences from Linux sockets (using the socket.h library) in terms of programming functionality, however they all make use of the same transport protocols(udp/tcp) so they can talk to each other just fine.

The BSD network stack (BSD sockets) Most operating systems and network libraries base their methodology of handling packets and sockets on the ”BSD standard” sometimes called ”BSD sockets”. BSD, short for Berkeley Software Distribution, was a distribution of Unix in the 70s, BSD was used as a starting point/inspiration for many unix/linux distributions. The BSD project was overtaken by open source versions which are still in active development today. As so many systems were designed following the methodologies of BSD, the code from the open source projects was re-integrated into many newer systems, including Windows. This resulted in all the major operating systems having the same or similar foundation networking code. The really good news is that the Sony LibNet library also follows this trend and has an almost exact clone of the NetBSD(an active open-source version of BSD) networking library. This

means that 90% of the networking code that works on most systems will also work on the PS3.

2. Playstation 3 specific Networking

2.1 Playstation 3 Network Libraries

The libraries that deal with Networking on the Playstation 3 are:

Lib-NetCtl
: Functions for getting the state of the network link. E.g. Is the system connected to a network? It can initiate a system dialogue to hep the user diagnose network issues.

Lib-Net
: The main networking workhorse of the PS3; Sockets, DNS, packets and connections. Based on the standard BSD style networking procedures.

Lib-Rudp
: An implementation of *Reliable User Datagram Protocol*. This uses libnet to create a UDP connection, which it uses to send the normal data packets and the extra data checking bits. Faster but less robust than TCP, slower but more reliable than UDP. Very useful for Games.

Lib-htttp
: Used for decoding and encoding http data. Can create and send http requests, and deal with the returned data in a usable fashion. There are also features to support cookies, and other basic http features. All this is possible with raw packets and libnet, but why do the extra work?

Lib-htttp_util
: Used extensively behind the scenes by Lib-htttp, this lib defines the data-types and decoding/encoding functions for things like urls and http status codes.

Lib-ssl
: Like libhtttp_util, this is used by libhttp for HTTPS communication. This library cannot be used as a standalone library for secure communication as SSL certificates rely on other http features (e.g. Hostnames) to function properly.

2.2 Network setup

Tasks such as obtaining an IP address and connecting to a network are all handled by the system and the user settings in the main menu. The available libraries give no access to these functions, the NetCTL library is used only to detect if the system is connected to a network. If there is no available connection, the only thing you can do ask the system to display a prompt to the user to fix the network settings. If you try to call network functions in other libraries with no active network connection you will get all kinds of errors, and possibly even crash the system.

Connecting the devkit to a network The PS3 Reference tool (DECR-1000) has two Ethernet ports, one for development and debugging(the dev port), the other is for application use(the system port), essentially this is like the Ethernet port on a regular PS3. Once you have the system port plugged into a network, it can be configured from the system menu.

2.3 Network Test Code

This code will only initialise NetCTL and the continuously test the network connection status. This is a useful piece of code to have as you will need to reuse it every time you need to do any networking operation. As with most PS3 code you will write, there is a substantial amount of safety wrapping around the few lines one line of code doing the work.

Listing 1. Testing the Network status

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cstring>
4 #include <netex/libnetctl.h>
5 #include <cell/sysmodule.h>
6 #include <sys/timer.h>
7 #include <sysutil/sysutil_sysparam.h>
8 //Linked with: libnetctl_stub.a, libsysmodule_stub.a, libsysutil_stub.a
9
10 //#define DBG_HALT { __asm__ volatile( "trap" ); }
11 #define HALT { std::abort(); }
12 // Calls the suplied function on assert fail, then call DBG_HALT
13 #define ASSERT_F( exp, func) { if ( !(exp) ) {func; HALT;} }
14
15 void sysutil_callback(uint64_t status, uint64_t param, void *userdata)
16 {
17  printf("System Event! %#08x\n",status);
18   if (status == CELL_SYSUTIL_REQUEST_EXITGAME) {
19      printf("System has requested EXITGAME\n");
20   std::abort();
21  }
22 }
23
24 int main(void)
25 {
26  //Register the exit handler
27  int err = cellSysutilRegisterCallback(0, sysutil_callback, NULL);
28
29  //Load the network system module
30  err = cellSysmoduleLoadModule(CELL_SYSMODULE_NETCTL);
31  ASSERT_F((err == CELL_OK),
32     printf("cellSysmoduleLoadModule() : %x\n", err));
33
34  // Init netCTL library
35  err = cellNetCtlInit();
36  ASSERT_F((err >= 0), printf("cellNetCtlInit() error : %x\n", err));
37
38  int connection_status;
39  int prev_status = -1;
40  while (true)
41  {
42   // Get connection status
43   err = cellNetCtlGetState( &connection_status );
44   ASSERT_F((err >= 0),printf("cellNetCtlGetState() : %x\n" err));
45   if(prev_status != connection_status)
46   {
47    switch ( connection_status ) {
48     case CELL_NET_CTL_STATE_Disconnected:
49      printf("Network: Dissconected\n");
50      break;
51     case CELL_NET_CTL_STATE_Connecting:
52      printf("Network: Connecting\n");
53      break;
54     case CELL_NET_CTL_STATE_IPObtaining:
55      printf("Network: Obtaining IP address\n");
56      break;
57     case CELL_NET_CTL_STATE_IPObtained:
58      printf("Network: Connected\n");
59      union CellNetCtlInfo info;
60      err = cellNetCtlGetInfo(
61       CELL_NET_CTL_INFO_IP_ADDRESS, &info);
62      ASSERT_F((err >= 0),printf("cellNetCtlGetInfo():%x\n",err))
63      printf("IP: %s\n",info.ip_address);
64      break;
65    }
66   }
67   prev_status = connection_status;
```

```
68     sys_timer_sleep(1);
69   }
70 }
```

2.4 HTTP fetch Code

With the background info out of the way, we can create something useful to end the tutorial with. The following code listing makes use of the http module to request and load a webpage and display the source and http headers in the console window. An understanding of the http protocol will help with understanding certain parts of this code. The simple breakdown is:

- Client sends a request
- Server sends back a ”Header” which contains a status code and metadata about the requested resource.
- The client can then request the ”Body” to be sent, or this can be pre-requested in the original request. This is what we do in this code.
- Server sends the ”Body”, which is normally the web page in html format.

Listing 2. Testing the Network status

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/timer.h>
5 #include <netex/net.h>
6 #include <netex/errno.h>
7 #include <netex/libnetctl.h>
8 #include <cell/http.h>
9 #include "ps3_network_utils.h"
10 #include <cell/sysmodule.h>
11
12 /* Linked Libs
13   libnetctl_stub.a, libsysmodule_stub.a, libnet_stub.a,
14   libhttp_util_stub.a, libhttp_stub.a
15 */
16 #define REQUEST_URI  "www.example.com"
17 #define BUFFER_SIZE  256
18 #define HTTP_POOL_SIZE (8 * 1024)
19
20 void getheaders();
21
22 bool is_up()
23 {
24   int timeout_count = 10;
25   int state;
26   int err;
27
28   while(timeout_count > 0) {
29     err = cellNetCtlGetState(&state);
30     ASSERT_F((err >= 0),printf("NetCtl GetState error: %x\n" err));
31     if (state == CELL_NET_CTL_STATE_IPObtained) {
32       return true;
33     }
34     sys_timer_usleep(500000);
35     timeout_count--;
36   }
37   return false;
38 }
39
40 int main(void)
41 {
42   printf("\n\n------- Program Started -------\n\n");
43   int err = -1;
44
45   //====== SETUP ====
46   printf("intialising HTTP module\n");
47   err = cellSysmoduleLoadModule(CELL_SYSMODULE_HTTP);
48   ASSERT_F((err >= 0), printf("Module Load error: %x\n", err));
49
50   //Load the netCTL module
51   err = cellSysmoduleLoadModule(CELL_SYSMODULE_NETCTL);
52   ASSERT_F((err == 0), printf("Module Load error: %x\n", err));
53
54   // start network
55   printf("\nstarting LibNet network\n");
56   err = sys_net_initialize_network();
57   ASSERT_F((err >= 0), printf("network init error: %x\n", err));
58
59   // Init netCTL library
60   err = cellNetCtlInit();
61   ASSERT_F((err >= 0), printf("cellNetCtlInit() error : %x\n", err));
62
63   printf("\nifconfig\n\n");
64   err = sys_net_show_ifconfig();
65   ASSERT_F((err == 0), printf("ifconfig error: %x\n", err));
66
67   printf("nameservers:\n");
68   err = sys_net_show_nameserver();
69   ASSERT_F((err == 0), printf("nameserver error: %x\n", err));
70
71   if(!is_up()){
72     printf("\nNetwork is down, can't contiune\n");
73     return 0;
74   }
75   printf("\nNetwork is live, starting program\n");
76
77   //====== Main Program ==========
78   getheaders();
79
80   //======= Finish =======
81   sys_net_finalize_network();
82   printf("done\n");
83   return 0;
84 }
85
86 void getheaders()
87 {
88   CellHttpClientId client = 0;
89   CellHttpTransId trans = 0;
90   CellHttpUri uri;
91   int err;
92   bool has_cl = true;
93   uint64_t length = 0;
94   uint64_t recvd;
95   size_t poolSize = 0;
96   const char *serverName;
97   size_t localRecv = 0;
98
99   //Array to store headders in
100  CellHttpHeader *headers;
101  //buffer to store http data as it's parsed.
102  char buffer[BUFFER_SIZE];
103
104  //Memory area to store the http status
105  void *statusPool = NULL;
106  //Memory area to store the http headers
107  void *headersPool = NULL;
108  //Memory area to store the parsed url
109  void *uriPool = NULL;
110  //Memory area to store http library buffers
111  void *httpPool = NULL;
112
113  memset(buffer, 0x00, sizeof(buffer));
114
115  serverName = REQUEST_URI;
116
117  //===== startup procedures ====
118  httpPool = malloc(HTTP_POOL_SIZE);
119  ASSERT_F((httpPool != NULL), printf("libhttp malloc failed\n"));
120
121  err = cellHttpInit(httpPool, HTTP_POOL_SIZE);
122  ASSERT_F((err == 0), printf("libhttp error (0x%x)\n", err));
123
124  err = cellHttpCreateClient(&client);
125  ASSERT_F((err == 0), printf("http client error(0x%x)\n", err));
126
127  //===== parse uri =====
128  err = cellHttpUtilParseUri(NULL,serverName,NULL,0,&poolSize);
129  ASSERT_F((err == 0), printf("error parsing URI (0x%x)\n", err));
```

```
130
131 uriPool = malloc(poolSize);
132 ASSERT_F((uriPool != NULL), printf("uriPool malloc failed)\n"));
133
134 err=cellHttpUtilParseUri(&uri,serverName,uriPool,poolSize,NULL)
135 ASSERT_F((err == 0), printf("error parsing URI... (0x%x)\n", err));
136
137 printf("\nUri parsed as:\n");
138 printf("  scheme:   %s\n", uri.scheme);
139 printf("  hostname: %s\n", uri.hostname);
140 printf("  port:     %d\n", uri.port);
141 printf("  path:     %s\n", uri.path);
142 printf("  username: %s\n", uri.username);
143 printf("  password: %s\n\n", uri.password);
144
145 //====== Send Request ======
146
147 err = cellHttpCreateTransaction(
148       &trans, client, CELL_HTTP_METHOD_GET, &uri);
149 ASSERT_F((err==0),printf("Create Transaction failed: %x\n",err));
150
151 free(uriPool);
152 uriPool = NULL;
153
154 err = cellHttpSendRequest(trans, NULL, 0, NULL);
155 ASSERT_F((err == 0), printf("Send Request failed: %x\n", err));
156
157 //======= Get status code =======
158
159 CellHttpStatusLine status;
160 err = cellHttpResponseGetStatusLine(
161         trans, NULL, NULL, 0, &poolSize);
162 ASSERT_F((err == 0), printf("GetStatus failed: %x\n", err));
163
164 statusPool = malloc(poolSize);
165 ASSERT_F((statusPool!=NULL),printf("statusPool malloc fail\n"))
166
167 err = cellHttpResponseGetStatusLine(
168       trans, &status, statusPool, poolSize, NULL);
169 ASSERT_F((err == 0), printf("GetStatus failed: %x\n", err));
170
171 printf("HTTP Status Code is %d\n", status.statusCode);
172
173 //======= Get headers ========
174
175 size_t items = 0;
176 err = cellHttpResponseGetAllHeaders(
177         trans, NULL, &items, NULL, 0, &poolSize);
178 ASSERT_F((err == 0), printf("GetAllHeaders failed: %x\n", err));
179
180 if (items) {
181   headersPool = malloc(poolSize);
182   ASSERT_F((NULL != headersPool), printf("malloc fail\n"));
183
184   err = cellHttpResponseGetAllHeaders(
185         trans, &headers, &items, headersPool, poolSize, NULL);
186   ASSERT_F((err == 0), printf("GetAllHeaders failed: %x\n", err));
187 }
188
189 //======= print headers ===========
190
191 printf("\nHeaders: \n");
192 printf("  %s %d.%d %d %s\n",
193     status.protocol, status.majorVersion, status.minorVersion,
194     status.statusCode, (status.reasonPhrase)?status.reasonPhrase:""
195   );
196 for (int i = 0; i < items; ++i) {
197   printf("  %s: %s\n", headers[i].name, headers[i].value);
198 }
199
200 free(headersPool);
201 headersPool = NULL;
202 free(statusPool);
203 statusPool = NULL;
204
205 //======= get content ========
206
207 err = cellHttpResponseGetContentLength(trans, &length);
208 if (err != 0 ) {
209   if (err == CELL_HTTP_ERROR_NO_CONTENT_LENGTH) {
210     has_cl = false;
211   } else {
212     printf("error in receiving content length... (0x%x)\n\n", err);
213     HALT;
214   }
215 }
216 //printf("Received content, with length %llu\n", length);
217
218 //====== print content ==========
219 recvd = 0;
220 printf("\n\n=====BEGINNING OF TRANSMISSION=====\n");
221 while ((!has_cl) || (recvd < length)) {
222   err = cellHttpRecvResponse(
223       trans, buffer, BUFFER_SIZE-1, &localRecv);
224   ASSERT_F((err == 0), printf("error parsing response %x\n", err));
225   if (localRecv == 0){
226     break;
227   }
228   recvd += localRecv;
229   buffer[localRecv] = '\0';
230   printf("%s", buffer);
231 }
232 printf("=====END OF TRANSMISSION=====\n\n");
233
234 //======= shutdown procedures ===========
235 if (trans) {
236   cellHttpDestroyTransaction(trans);
237   trans = 0;
238 }
239 if (client) {
240   cellHttpDestroyClient(client);
241   client = 0;
242 }
243 cellHttpEnd();
244 free(httpPool);
245 httpPool = 0;
246 return;
247 }
```

3. Conclusion

The http example only uses the libNet library behind the scenes, so this example doesn't deal with the *real* internals of networking(e.g sockets), but it does serve a useful function. You could host new content and updates on a web page or use a web server as a game server browser. HTTP operates on the higher levels of networking and can be just as essential as actual game data packets.

Future tutorials will cover operating the lower levels, whereas this tutorial is an introduction to the theory and to demonstrate the level of code needed to start writing network code.

Due to the implementation of commonly used methodologies and libraries, networking on the Plantation 3 is no more difficult than on any other platform (but as always, you have to watch your memory usage much more closely).

References

[1] Wikipedia The Internet protocol suite. www.wikipedia.org. *Accessed: July 2014*. 2

[2] Edinburgh Napier Game Technology Website. www.napier.ac.uk/games/. *Accessed: Feb 2014*. 1