A Case Study in Ada2012: Implementing a simple Http Server

Simon J. Symeonidis

September 17, 2013

Contents

List of Figures

List of Tables

Listings

::Section 1::

Abstract

This is a small case study in the 2012 release of the Ada programming language. The motivation of this case study is to expose one to the broadest topics of the Ada programming language via a detailed example. The topic I have chosen is to write up a very simple web server. A web server typically, has to deal with the following aspects:

- Concurrency
- Sockets, and messaging
- The Protocol (in this case **http**)
- Different Filesystems support

This document is hoped to help the curious reader to learn the aspects of the language with a little more ease, introduce the http protocol slightly, and provide references where possible to direct to sources for more exploration. I also record personal notes. There is a core aim, to present the aforementioned array of topics to the reader, in a project that is not too big, and intimidate the beginner, and not too small to not be of any learning use. We try and achieve this balance by reducing the lines of code and sacrificing aspects of software architecture[1] in order achieve this.

1.1 Disclaimer

As previously stated, this is a case study that is worked on by a complete beginner in the Ada2012 programming language. You might find that this paper is referencing a lot of sources, in a collage fashion. That may be a true fact, but the purpose of this paper is not to shed new light to a problem. This paper is aimed as a tutorial and reference notes to the programming language, for the beginners and intermediate programmers. Please also consider using the excellent references, as they have provided a lot of help in this case study, and can be used more to broaden ones' knowledge.

1.2 The name 'Axios'

Axios stands for "Ada eXtra ineffectual Object Oriented Server". And it is just that!

[1] In the code base, you might notice some aspects that could be factored out, or areas where coupling could be reduced. This will be done on purpose in order to reduce the amount of indirection, and ease the quick navigation, reading and understanding for the reader.

::Section 2::

Topics of interest

Here we discuss in a little more detail the previously outlined topics. These topics offer us a concise and detailed tests to investigate the the Ada 2012 progrqamming language, along with other tools that can be used to supplement and extend future projects. We will be looking at the following aspects:

- The Ada2012 programming language (Language structure, strengths, weaknesses, etc)
- The Ada2012 Toolchain The **gnat** compiler, the project manager files (gpr)
- AUnit for a unit testing platform

2.1 Concurrency

Ada is considered to have a good concurrency support.

2.2 Sockets

Sockets are a good topic in this case study, since sockets are OS dependent therefore will provide some insight on how the language and application will behave on different platforms.

2.3 Cross Platform Filesystem Support

To programmers that have used languages that do not require virtual machines, concerns for cross platform availability of an application, more often arise. This case study will use some simple filesystem operations to indicate what we may and may not do with the language. We will investigate how:

- Location to resources are handled
- Permissions for inodes are handled

2.4 Protocol Implementation

A topic that I have came accross in programming languages that is somewhat interesting to see in implementation, are protocols. Notably **Erlang** makes this a breeze. Investigating this in Ada will be interesting.

For this small project, we'll be implementing 5 parts of the Http protocol. Namely the HEAD, GET, POST, PUT and DELETE methods. We refer to the RFC2616 manual [7] for the definition of behaviours of these methods. Once they are implemented in this case study using Ada, the project shall be tested by using any mainstream browser such as *Firefox* or *Chromium*.

For the reader we extract the definitions and behaviour of these four methods.

2.4.1 Overview of Http Protocol

The Http protocol is a text, request-response protocol. The RFC2616 specification is quite detailed, but we will be implementing a very small subset of the specification for this case study.

The Http version we are concerned with is **1.1**. There exists indempodent and non indempodent methods in the Http specification.

GET is described as an indempodent method. That is, if a request is being made with the same specific set of parameters, then the response should be the same each time that request is repeated. For example, if we had a function such as $f(x) = x + 1$, then each time we provided $x = 1$ we should always retrieve '2'. We present the following table, which describes the methods we will be implementing:

Method	Type
HEAD	Indempondent
GET	Indempondent
POST	Non-Indempondent
PUT	Indempondent
DELETE	Indempondent

Table 1: HTTP Method Indempodency

2.4.2 HEAD

This Http method requests the header fields of an html document. The message body is not to be retrieved. Here is an example of what http headers look like. The definition is taken straight out of the manual:

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                    and consisting of either *TEXT or combinations
                    of token, separators, and quoted-string>
```

Listing 1: Header Definition

Here is a more practical example. These headers are retrieved from *www.google.ca*:

```
HTTP/1.1 302 Found
Location: http://www.google.ca/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Date: Mon, 22 Apr 2013 20:26:36 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
```

Listing 2: Html Header

And this is the expected html response definition from a successful request.

```
Response  = Status-Line
            *(( general-header
              | response-header
              | entity-header ) CRLF)
            CRLF
            [ message-body ]
```

Listing 3: Http Response

2.4.3 GET

A **GET** method is defined in the specification, to be used for operations that require **reading** information. On the server side, this should not tamper with the resources. A get request requires **Request-Line** field to be sent. It is up to the server to look for the resource requested, and send the response afterwards.

Taken directly from the manual, here is a demonstration of what the GET request looks like, when "accessing www.w3.org":

```
1 GET /pub/WWW/TheProject.html HTTP/1.1
2 Host: www.w3.org
```

Listing 4: Http Get Request

2.4.4 POST

POST is the non-indempodent method we mentioned before. This means each time a post request is sent, a resource must be created, always guaranteeing different response, for a same request. An example for this would be creating a book with common data fields such as *author, year, title, etc...*

This is what a typical POST request looks like:

In this case study, we will be using this to write a text file on the server, with the request body as file contents. It would be possible to retrieve these contents later on.

2.4.5 PUT

PUT is a indempodent method. It is typically used in order to update resources on the sever side. For example, if we wish to update the book re

In this case study we will be using this to change a variable on the server. That way it will be possible to start the server, change this value, and get a response showing that the update was a success. It is quite possible to use this along with AUnit in order unit test these features automatically.

2.4.6 DELETE

DELETE is an indempodent method. It is typically used in order to delete a resource on a website. We will be using this to delete resources that are created by the POST definition of Axios (refer to 2.4.4).

2.5 First Steps

The first steps in an Ada application is to define the *project manager file* for the given project as outlined on the manual [2].

The GNAT project file abstracts a lot of the building details into this file. It will take care even about other language builds, meaning that if you wish to create, for example C extensions to the language, you could achieve this easily, as opposed to tedious makefiles.

Another added advantage is that GNAT Project Files also take into consideration the file naming conventions. Another automated feature in this file is automatic building of external libraries. There are various other ways to achieve this, either on a makefile level, or alternatively using features in certain SCM tools to clone and build the libraries automatically.

A notable feature of the GPR files, is that hierarchical builds are allowed, meaning that if different components require different builds, and different dependencies, that build can be isolated to that specific build alone [2]. The hierarchical layering of the building process, also exploits a similar aspect to that of a layered software architecture: different layers can be developed at parallel, as could subsystems.

2.5.1 Deployment View

The overall view of the system components can be seen in Figure 1.

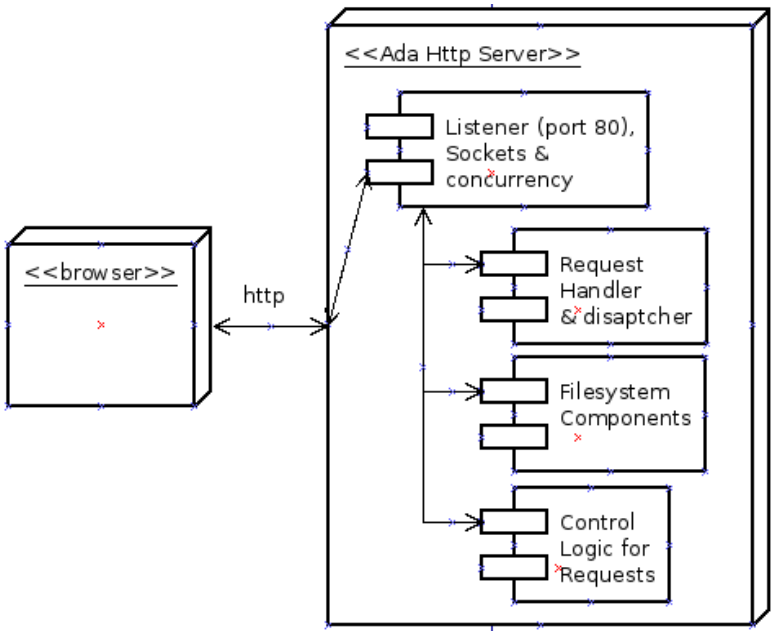

Figure 1: Deployment Diagram

src	The directory to contain the sources
obj	The directory to contain the compiler output
bin	The directory to contain the binaries
obj/debug	The non-optimized compiled binaries with debug info are stored here. You can notice this in the GPR file, where the '-g' flag is given to the compiler.
obj/release	The optimized compiled binaries without debug information. You can notice this in the GPR file, where the 'O2' flag is given to the compiler.
tests	The directory to contain the tests

Table 2: Directory Structure Specification for GPR File

::Section 3::

Implementation

This section will deal with the more technical aspects of this case study. From now on we will list topics that have to do with the implementation of the simple Http server.

3.1 Defining the required GNAT Project File Specification

The specifications list the following requirements for the directory structures:

3.1.1 GNAT Project File

Here is the project file we will be using.

```
1  -- Simon Symeonidis, 2013
2  -- GNAT Project file for the Http server case study.
3  --
4  -- Used Manuals:
5  --   http://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html
6  --   http://docs.adacore.com/gnat-unw-docs/html/gnat_ugn_12.html
7  --
8  -- Example use:
9  --   gnatmake -P axios.gpr -Xmode=debug -p
10 project Axios is
11
12   -- Standard configurations
13   for Main        use ("main.adb");
14   for Source_Dirs use ("src/**");
15   for Exec_Dir    use "bin/";
16
17   -- Ignore git scm stuff
18   for Ignore_Source_Sub_Dirs use (".git/");
19
20   -- Objects are contained in their own directories (this is also
21   -- known as an isolated build).
22   for Object_Dir use "obj/" & external ("mode", "debug");
23   for Object_Dir use "obj/" & external ("mode", "release");
24
25   package Builder is
26     for Executable ("main.adb")  use "axios";
27   end Builder;
28
29   -- To invoke either case, you need to set the -X flag at gnatmake in
         command
30   -- line. You will also notice the Mode_Type type. This constrains the
         values
```

```
31    -- of possible valid flags.
32    type Mode_Type is ("debug", "release");
33    Mode : Mode_Type := external ("mode", "debug");
34    package Compiler is
35      -- Either debug or release mode
36      case Mode is
37      when "debug" =>
38        for Switches ("Ada") use ("-g");
39      when "release" =>
40        for Switches ("Ada") use ("-O2");
41      end case;
42    end Compiler;
43
44    -- Not needed for this study, but listed
45    package Binder is
46    end Binder;
47
48    -- Not needed for this study, but listed
49    package Linker is
50    end Linker;
51
52  end Axios;
```

Listing 5: GNAT Project File

Notice that some of the flags for the compiler in both release, and debug modes are the same flags used for GNU C++ compiler. We explain each section with a little more detail.

Directory Structure The directory structure is as outlined on the previously demonstrated table. We see these defined in the GPR file by the names of **Source_Dirs**, **Exec_Dir**. Because of the definition of the Object file directories for a debug and release version, we also get the "obj/release" and "obj/debug" directories, as previously required. A Simple hello world application would yield the following structure, for example:

```
1  .
2  bin
3  bin/hello_world
4  hello.gpr
5  obj
6  obj/debug
7  obj/debug/main.ali
8  obj/debug/main.o
9  obj/release
10 obj/release/main.ali
11 obj/release/main.o
12 src
13 src/main.adb
14
15 5 directories, 7 files
```

Listing 6: Directory Structure

Flags There are many flags that can be set for the compilation and linking process. The manual covers a lot of them, but the ones used are explained.

Builder Notice the "package Builder is ..." line. We define the flags that should be used by the builder (gnatmake), when invoked.

Executable This is to specify the output name of the binary once the whole project is compiled.

Compiler Notice the "package Compiler is ..." line. This contains the flags for the actual compiler. We have two flags '-g', and '-O2' that the seasoned GNU C++ compiler user would recognize instantly. The '-g' flag would be used to produce non-optimized binaries in order to contain extra information when debugging. The '-O2' is for optimization, and that is the reason why it is in the release case. We define a Mode_Type which can either be one of the two literals 'debug', and 'release' in order to achieve this.

3.2 Components

We now describe the components of the project. Figure 2

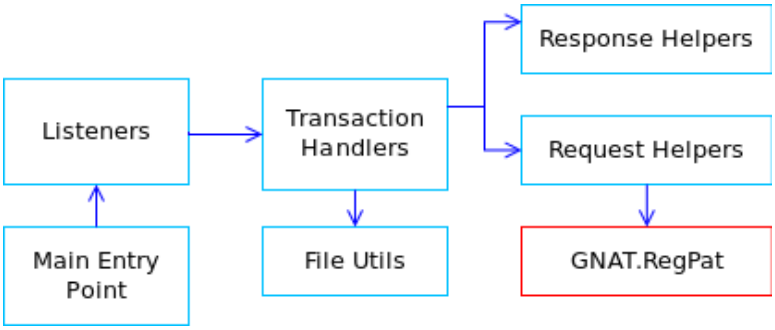

Figure 2: Overall organization of software components of Axios

3.2.1 Listener

Specification Here is the code listing for the listener specification

```
1 -- @author Simon Symeonidis
2 -- @date   Mon May 6 2013
3 -- Implementation for the listener
4 -- @note Also good to know why using IO Streams will not work with this
      sort of thing.
5 -- http://stackoverflow.com/questions/7540064/simple-http-server-in-ruby-
      using-tcpserver
6 -- @note Using Ada get_line for unknown input size
7 -- http://www.radford.edu/~nokie/classes/320/stringio.html
8
9 package body Listeners is
10
11   -- Print to the command line information such
12   -- as host and port number
13   procedure Print_Info(This : Listener) is
14   begin
15     Ada.Text_IO.Put_Line("Listener Info: ");
16     Ada.Text_IO.Put_Line("Port Number : "
17       & Integer'Image(This.Port_Number));
18     Ada.Text_IO.Put_Line("Hostname    : "
19       & Ada.Strings.Unbounded.To_String(This.Host_Name));
20     Ada.Text_IO.Put_Line("Root Dir.   : "
21       & Ada.Strings.Unbounded.To_String(This.WS_Root_Path));
22     Ada.Text_IO.New_Line;
23   end Print_Info;
24
25   function Tiny_Name(This : Listener)
26   return String is
27     Name : String :=
28       Ada.Strings.Unbounded.To_String(This.Host_Name) & "@" &
29       Integer'Image(This.Port_Number);
30   begin
```

```
31      return Name;
32    end Tiny_Name;
33
34    -- Return the date as string, in the format specified by the RFC2616
          manual.
35    -- TODO need to fix this.
36    function Response_Date
37    return String is
38    begin
39      return "Date: Tue, 20 Jan 2012 10:48:45 GMT";
40    end Response_Date;
41
42    -- Listen forever. Graceful shutdown if it receives some signal.
43    procedure Listen(This : Listener) is
44      Socket   : Socket_Type;
45      Server   : Socket_Type;
46      Address  : Sock_Addr_Type;
47      Channel  : Stream_Access;
48      The_Host : String := Ada.Strings.Unbounded.To_String(This.Host_Name);
49    begin
50
51      Address.Addr := Addresses (Get_Host_By_Name (The_Host), 1);
52      Address.Port := Port_Type(This.Port_Number);
53      Create_Socket(Server);
54
55      Set_Socket_Option(Server, Socket_Level, (Reuse_Address, True));
56      Bind_Socket(Server, Address);
57      Listen_Socket(Server);
58      -- pass to other socket.
59      Ada.Text_IO.Put_Line
60       ("Successfully listening on: " &
61        Port_Type'Image(Address.Port));
62
63      Listening_Loop :
64      while not This.Shutdown loop
65        Accept_Socket(Server, Socket, Address);
66        Channel := Stream(Socket);
67        declare
68          CRLF         : String    := ASCII.CR & ASCII.LF;
69          LF           : Character := ASCII.LF;
70          Counter      : Integer   := 0;
71
72          Request      : Ada.Strings.Unbounded.Unbounded_String;
73          Chara        : Character;
74
75          RTime_Start  : Ada.Real_Time.Time := Ada.Real_Time.Clock;
76          RTime_Stop   : Ada.Real_Time.Time;
77          RTime_Total  : Ada.Real_Time.Time_Span;
78        begin
79          -- Read the request body
80          Read_Request :
81          loop
82          Character'Read(Channel, Chara);
83          Ada.Strings.Unbounded.Append(
84            Source   => Request,
85            New_Item => Chara);
86
87            if Chara = ASCII.LF or Chara = ASCII.CR then
88              Counter := Counter + 1;
89            else -- reset
90              Counter := 0;
91            end if;
92
93            exit when Counter = 4;
94          end loop Read_Request;
95
96          RTime_Stop  := Ada.Real_Time.Clock;
97          RTime_Total := RTime_Stop - RTime_Start;
```

```
98
99          Ada.Text_IO.Put_Line(
100           Ada.Strings.Unbounded.To_String(
101             Request));
102
103         String'Write(Channel,
104         Transaction_Handlers.Handle_Request(
105           Ada.Strings.Unbounded.To_String(Request),
106             Ada.Strings.Unbounded.To_String(This.WS_Root_Path)));
107
108       exception when E : others =>
109         Ada.Text_IO.Put_Line
110           ("Listeners, at main listen loop: " &
111           Exception_Name(E) & Exception_Message(E));
112
113       end;
114       Free(Channel);
115       Close_Socket(Socket);
116     end loop Listening_Loop;
117
118   end Listen;
119
120 end Listeners;
```

Listing 7: Simple Listener Specification

Implementation Here is the code listing for the listener implementation

```
1  -- @author Simon Symeonidis
2  -- @date   Mon May 6 2013
3  -- Implementation for the listener
4  -- @note Also good to know why using IO Streams will not work with this
       sort of thing.
5  -- http://stackoverflow.com/questions/7540064/simple-http-server-in-ruby-
       using-tcpserver
6  -- @note Using Ada get_line for unknown input size
7  -- http://www.radford.edu/~nokie/classes/320/stringio.html
8
9  package body Listeners is
10
11   -- Print to the command line information such
12   -- as host and port number
13   procedure Print_Info(This : Listener) is
14   begin
15     Ada.Text_IO.Put_Line("Listener Info: ");
16     Ada.Text_IO.Put_Line("Port Number : "
17       & Integer'Image(This.Port_Number));
18     Ada.Text_IO.Put_Line("Hostname    : "
19       & Ada.Strings.Unbounded.To_String(This.Host_Name));
20     Ada.Text_IO.Put_Line("Root Dir.   : "
21       & Ada.Strings.Unbounded.To_String(This.WS_Root_Path));
22     Ada.Text_IO.New_Line;
23   end Print_Info;
24
25   function Tiny_Name(This : Listener)
26   return String is
27     Name : String :=
28       Ada.Strings.Unbounded.To_String(This.Host_Name) & "@" &
29       Integer'Image(This.Port_Number);
30   begin
31     return Name;
32   end Tiny_Name;
33
34   -- Return the date as string, in the format specified by the RFC2616
       manual.
```

```
35   -- TODO need to fix this.
36   function Response_Date
37   return String is
38   begin
39     return "Date: Tue, 20 Jan 2012 10:48:45 GMT";
40   end Response_Date;
41
42   -- Listen forever. Graceful shutdown if it receives some signal.
43   procedure Listen(This : Listener) is
44     Socket   : Socket_Type;
45     Server   : Socket_Type;
46     Address  : Sock_Addr_Type;
47     Channel  : Stream_Access;
48     The_Host : String := Ada.Strings.Unbounded.To_String(This.Host_Name);
49   begin
50
51     Address.Addr := Addresses (Get_Host_By_Name (The_Host), 1);
52     Address.Port := Port_Type(This.Port_Number);
53     Create_Socket(Server);
54
55     Set_Socket_Option(Server, Socket_Level, (Reuse_Address, True));
56     Bind_Socket(Server, Address);
57     Listen_Socket(Server);
58     -- pass to other socket.
59     Ada.Text_IO.Put_Line
60      ("Successfully listening on: " &
61       Port_Type'Image(Address.Port));
62
63     Listening_Loop :
64     while not This.Shutdown loop
65       Accept_Socket(Server, Socket, Address);
66       Channel := Stream(Socket);
67       declare
68         CRLF         : String    := ASCII.CR & ASCII.LF;
69         LF           : Character := ASCII.LF;
70         Counter      : Integer   := 0;
71
72         Request      : Ada.Strings.Unbounded.Unbounded_String;
73         Chara        : Character;
74
75         RTime_Start  : Ada.Real_Time.Time := Ada.Real_Time.Clock;
76         RTime_Stop   : Ada.Real_Time.Time;
77         RTime_Total  : Ada.Real_Time.Time_Span;
78       begin
79         -- Read the request body
80         Read_Request :
81         loop
82         Character'Read(Channel, Chara);
83         Ada.Strings.Unbounded.Append(
84           Source   => Request,
85           New_Item => Chara);
86
87           if Chara = ASCII.LF or Chara = ASCII.CR then
88             Counter := Counter + 1;
89           else -- reset
90             Counter := 0;
91           end if;
92
93           exit when Counter = 4;
94         end loop Read_Request;
95
96         RTime_Stop  := Ada.Real_Time.Clock;
97         RTime_Total := RTime_Stop - RTime_Start;
98
99         Ada.Text_IO.Put_Line(
100          Ada.Strings.Unbounded.To_String(
101            Request));
102
```

```
103           String'Write(Channel,
104           Transaction_Handlers.Handle_Request(
105             Ada.Strings.Unbounded.To_String(Request),
106               Ada.Strings.Unbounded.To_String(This.WS_Root_Path)));
107
108         exception when E : others =>
109           Ada.Text_IO.Put_Line
110             ("Listeners, at main listen loop: " &
111             Exception_Name(E) & Exception_Message(E));
112
113         end;
114         Free(Channel);
115         Close_Socket(Socket);
116       end loop Listening_Loop;
117
118     end Listen;
119
120 end Listeners;
```

Listing 8: Simple Listener Body

::Section 4::

Testing

Testing is achieved by using **AUnit** in order to do unit testing.

::Section 5::

Demonstration

Figure 3 is a small demo on what it looks like when accessing the *AXIOS* server. Figure 4 shows the output of the server (which is the text request sent by the browser, unless there is an error rased).

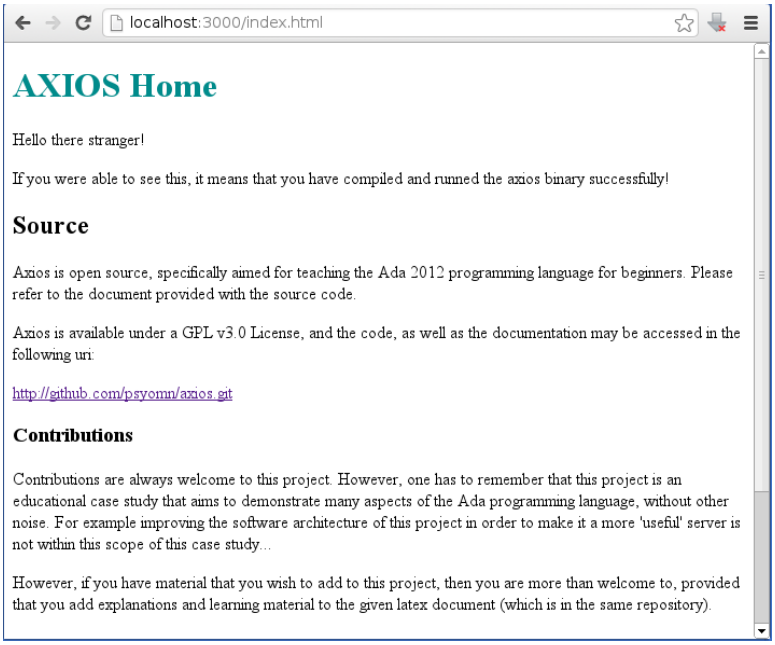

Figure 3: Using a browser to access the server

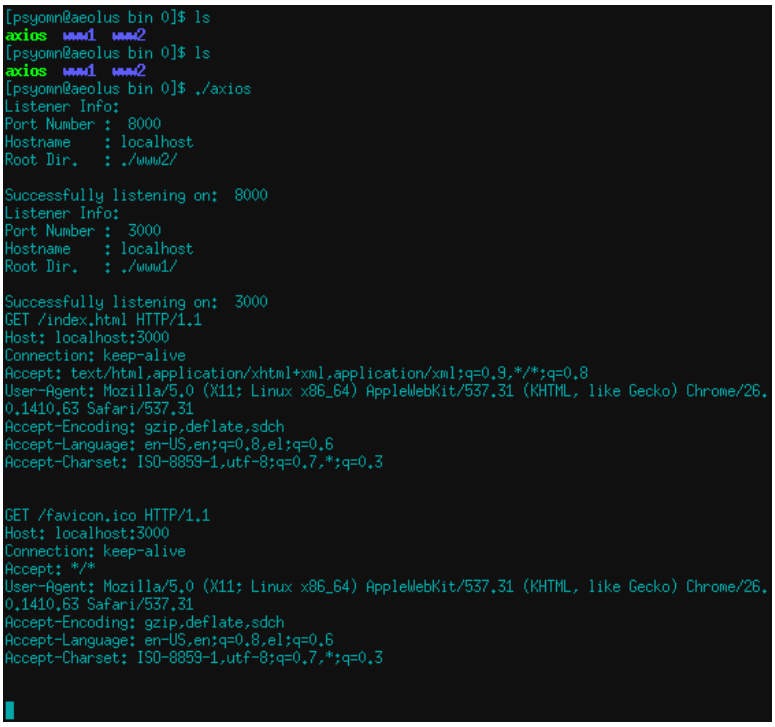

Figure 4: Server output when receiving requests

::Section 6::

Appendix

The appendix will give some extra simplified information which might help understand the smaller parts of the given case study. Explanations and code listings are provided as best as possible.

6.1 Networking

A lot of the networking will seem very familiar to you if you have used BSD C Sockets. Provided are two Ada procedures. One listener, and one sender. The Listener listens for incoming connections. If a message is intercepted, the message is uppercased, and sent back to the client. The client simply sends a string message to the listener. The listener listens at port 3000 (it is assumed these procedures run on your local machine, together).

```
1 -- @author Simon Symeonidis
2 -- A concise example of a listener procedure, adapted from the
3 -- ping pong example in the GNAT.Sockets specification file.
4 with Ada.Text_IO;
5 with Ada.Strings;
6 with Ada.Strings.Fixed;
7 with Ada.Strings.Maps;
8 with Ada.Strings.Maps.Constants;
9 with GNAT.Sockets; use GNAT.Sockets;
10 with Ada.Exceptions; use Ada.Exceptions;
11
12 procedure Simple_Listener is
13   Address  : Sock_Addr_Type;
14   Server   : Socket_Type;
15   Socket   : Socket_Type;
16   Channel  : Stream_Access;
17 begin
18   -- Init socket stuff.
19   Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);
20   Address.Port := 3000;
21   Create_Socket (Server);
22
23   Set_Socket_Option(Server, Socket_Level, (Reuse_Address, True));
24   Bind_Socket (Server, Address);
25   Listen_Socket(Server);
26   Accept_Socket(Server,Socket,Address);
27   Channel := Stream (Socket);
28   declare
29     Message : String := String'Input (Channel);
30   begin
31     Ada.Text_IO.Put_Line("Got: " & Message);
32     String'Output (Channel,
33       Ada.Strings.Fixed.Translate(Message,
34         Ada.Strings.Maps.Constants.Upper_Case_Map));
35   end;
36
37   Close_Socket (Server);
38   Close_Socket (Socket);
39
40   exception when E : others =>
41     Ada.Text_IO.Put_Line
42       (Exception_Name (E) & Exception_Message(E));
43 end Simple_Listener;
```

Listing 9: Simple Listener

```
1 -- @author Simon Symeonidis
```

```
 2 -- A concise example of a client procedure, adapted from the
 3 -- ping pong example in the GNAT.Sockets specification file.
 4
 5 with Ada.Text_IO;
 6 with GNAT.Sockets; use GNAT.Sockets;
 7
 8 procedure Simple_Sender is
 9   Address : Sock_Addr_Type;
10   Socket  : Socket_Type;
11   Channel : Stream_Access;
12 begin
13   -- Init socket stuff
14   Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);
15   Address.Port := 3000;
16   Create_Socket(Socket);
17   Set_Socket_Option(Socket, Socket_Level, (Reuse_Address, True));
18   Connect_Socket (Socket, Address);
19   Channel := Stream (Socket);
20   String'Output (Channel, "hack the planet.");
21
22   declare
23     Message : String := String'Input(Channel);
24   begin
25     Address := Get_Address(Channel);
26     Ada.Text_IO.Put_Line ("Server @ " & Image(Address)
27                           & ": " & Message);
28   end;
29
30   -- Cleanup
31   Close_Socket(Socket);
32 end Simple_Sender;
```

Listing 10: Simple Sender

Also, here is an example http get program that helped a lot in this case study. It is one of the GNAT Sockets examples that can be found in the following location [1]. I have modified the source a little, and provided it in listing 11.

```
 1 with Ada.Text_IO;              use Ada.Text_IO;
 2 with GNAT.Sockets;             use GNAT.Sockets;
 3 with Ada.Strings.Unbounded;    use Ada.Strings.Unbounded;
 4
 5 with Ada.Streams;
 6 use type Ada.Streams.Stream_Element_Count;
 7
 8 -- Thanks to
 9 -- http://en.wikibooks.org/wiki/Ada_Programming/Libraries/Ada.Streams#
       Read_attribute
10 procedure Vcv is
11    Client  : Socket_Type;
12    Address : Sock_Addr_Type;
13    Channel : Stream_Access;
14
15    CRLF    : String := (1 => ASCII.CR, 2 => ASCII.LF);
16    Send    : String := CRLF & CRLF;
17    Offset : Ada.Streams.Stream_Element_Count;
18    Data    : Ada.Streams.Stream_Element_Array (1 .. 256);
19 begin
20    GNAT.Sockets.Initialize;  -- initialize a specific package
21    Create_Socket (Client);
22    Address.Addr := Addresses (Get_Host_By_Name("www.google.com"));
23    Address.Port := 80;
24
25    Connect_Socket (Client, Address);
26    Channel := Stream (Client);
27
```

```
28     String'Write (Channel, "GET / HTTP/1.1" & CRLF &
29                             "Connection: close " & Send);
30     loop
31        Ada.Streams.Read (Channel.All, Data, Offset);
32        exit when Offset = 0;
33        for I in 1 .. Offset loop
34           Ada.Text_IO.Put (Character'Val (Data (I)));
35        end loop;
36     end loop;
37 end Vcv;
```

Listing 11: VCV Get Http Client

6.2 Task Oriented Concurrency Model

Concurrency in Ada uses tasks. Here is a small example to demonstrate this feature of the language that exploits a very interesting way of achieving concurrent tasks. If more detail is required, refer to [9]

6.2.1 Simple Concurrent Program

This is the simplest example in using tasks in Ada. The example was adapted from notes found in [5], and was also the first concurrent program I wrote for testing and understanding tasks, the first time around.

```
1  with Ada.Text_IO;
2
3  procedure Task_Types is
4    task type A (Label : Character) is
5      entry Start;
6    end A;
7
8    task body A is
9    begin
10     accept Start;
11
12     for I in Integer range 1..10 loop
13       Ada.Text_IO.Put_Line(Character'Image(Label) & " is working...");
14       delay 0.4;
15     end loop;
16   end A;
17
18   My_Task_01 : A(Label => 'A');
19   My_Task_02 : A(Label => 'B');
20   My_Task_03 : A(Label => 'C');
21
22 begin
23   My_Task_01.Start;
24   My_Task_02.Start;
25   My_Task_03.Start;
26 end Task_Types;
```

Listing 12: Task Types in Ada

This procedure would have the following output:

```
1 [psyomn@aeolus concurrency 0]$ ./task_types
2 'A' is working...
3 'B' is working...
4 'C' is working...
5 'A' is working...
6 'B' is working...
7 'C' is working...
8 'A' is working...
```

```
 9 'B' is working...
10 'C' is working...
11 'A' is working...
12 'C' is working...
13 'B' is working...
14 'A' is working...
15 'B' is working...
16 'C' is working...
17 'B' is working...
18 'C' is working...
19 'A' is working...
20 'A' is working...
21 'B' is working...
22 'C' is working...
23 'A' is working...
24 'B' is working...
25 'C' is working...
26 'A' is working...
27 'B' is working...
28 'C' is working...
29 'A' is working...
30 'B' is working...
31 'C' is working...
```

Listing 13: Output for the Task Types in Ada

6.3 Text Manipulation

We list some text manipulation example code in this section, if better understanding is required. First it would be better to familiarize oneself with the *String* type; one can refer to [8], and another excellent tutorial of Java vs Ada String type comparison [6].

6.3.1 Splitting

A common task one wants to do with strings is split them. Refer to this [4], to read upon string splitting. Notice that this was added on the GNAT platform.

6.3.2 Regular Expressions

Regular expressions don't quite exist in the Ada programming language. However the GNAT platform provides some libraries with similar effects. A simple example can be found at [3].

6.4 Random Notes

Some other miscellanious random notes, that might help in the future.

6.4.1 ADS / ADB file location

If the *gcc-ada* compiler is being used, the specification and body files can typically be found in the following directory

$$/usr/lib/gcc/\$MACHTYPE/\$GCC-VERSION/adainclude/$$

On another platform, namely *Lubuntu*, the absolute path included another directory right before the ada include one.

$$/usr/lib/gcc/\$MACHTYPE/\$GCC-VERSION/rts-native/adainclude/$$

6.4.2 Sanity Checking

I wanted to test out if things were working properly every now and then. Here are a few commands that I used in order to do this.

6.4.3 Using Telnet to conenct to a Http Server

You can use telnet to connect to a http server, and write the headers manually, and receive the actual response. This helps at understanding a little more how the protocol works, and was used in order to debug the server.

```
[psyomn@aeolus ~ 0]$ telnet www.concordia.ca 80
Trying 132.205.244.34...
Connected to www.concordia.ca.
Escape character is '^]'.
HEAD / Http/1.1
Host: www.concordia.ca
Connection: close

HTTP/1.1 200 OK
Date: Sun, 28 Apr 2013 23:08:49 GMT
Server: Apache
X-Powered-By: PHP/5.1.6
Vary: Accept-Encoding
Connection: close
Content-Type: text/html; charset=UTF-8

Connection closed by foreign host.
```

Listing 14: Telnet to an Http Server

6.4.4 Using ncat to dump packets to a binary file

On the various stages of debugging, *ncat* was used in order to dump the packets to a file, in order for inspection. This helped when I was facing trouble when the socket would send random garbage due to mistakes to the http servers, and have undesired consequences.

```
# First we start ncat on localhost, and make it listen at port 3000
[psyomn@aeolus ~ 0]$ ncat -l localhost 3000 > log.txt

# Then we telnet to ncat, at port 3000
[psyomn@aeolus ~ 0]$ telnet localhost 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello there.
These are messages that are sent to you via telnet
I hope you like them.
Connection closed by foreign host.

# The log has the messages sent. The log file can also store byte values, not
# just text messages, hence a means to see what the application is writing at
# said sockets.
[psyomn@aeolus ~ 0]$ cat log.txt
Hello there.
These are messages that are sent to you via telnet
I hope you like them.
```

Listing 15: Using ncat to dump packets

Here is an investigation example of data sent through sockets from a test script. This is a GET request.

```
[psyomn@aeolus 423 0]$ xxd /tmp/loggy
```

```
2 0000000: 4745 5420 2f20 4874 7470 2f31 2e31 0d0a  GET / Http/1.1..
3 0000010: 486f 7374 3a20 6c6f 6361 6c68 6f73 740d  Host: localhost.
4 0000020: 0a43 6f6e 6e65 6374 696f 6e3a 2063 6c6f  .Connection: clo
5 0000030: 7365 0d0a 0d0a                           se....
```

Listing 16: Using ncat a test script and xxd to hex dump

You can notice the CRLFs at the end of each request (that is the two bytes on the left, 0x0A, and 0x0D respectively).

And here is a small, priceless one-liner that has helped me debug some of my code during this case study. It uses ncat, and pipes the output to xxd, which in turn makes a hex dump.

```
1 [psyomn@aeolus 423 0]$ for (( ;; )) do ncat -l localhost 8080 | xxd; done
2 0000000: 0100 0000 0500 0000 4865 6c6c 6f          ........Hello
3 0000000: 0100 0000 0100 0000 48                    ........H
4 0000000: 0100 0000 0100 0000 48                    ........H
5 0000000: 4745 5420 2f20 4874 7470 2f31 2e31 0d0a  GET / Http/1.1..
6 0000010: 486f 7374 3a20 6c6f 6361 6c68 6f73 740d  Host: localhost.
7 0000020: 0a43 6f6e 6e65 6374 696f 6e3a 2063 6c6f  .Connection: clo
8 0000030: 7365 0d0a 0d0a                           se....
9 0000000: 2a00 0000 0100 0000 0100 0000 42          *...........B
10 0000000: 2a00 0000 0100 0000 0100 0000 42          *...........B
11 0000000: 2a00 0000                                 *...
12 0000000: 2a00 0000 2a00 0000 2a00 0000             *...*...*...
```

Listing 17: One liner to hex dump dynamically

References

[1] GNAT Sockets examples, 2013. [Online Fri May 3 2013].

[2] adacore. 2013. http://docs.adacore.com/gnat-unw-docs/html/gnat_ugn_12.html.

[3] Rosetta Code. Regular Expressions for Ada, 2013. http://rosettacode.org/wiki/Regular_expressions#Ada.

[4] commons.ada.cx. Substrings using Gnat.String_Split, 2013. `http://bit.ly/107SVuN`.

[5] M. J. Feldman. Chapter 15: Introduction to Concurrent Programming, 1996. http://www.seas.gwu.edu/ mfeldman/cs2book/chap15.html.

[6] Edward G. (Ned) Okie. Differences Between Strings in Ada and Java, 2013. http://www.radford.edu/ nokie/classes/320/strings.html.

[7] The Internet Society. 1999. http://tools.ietf.org/pdf/rfc2616.pdf.

[8] Wikibooks. Ada Programming: String Type, 2013. http://en.wikibooks.org/wiki/Ada_Programming/Strings.

[9] Wikibooks. Ada Programming: Tasking, 2013. http://en.wikibooks.org/wiki/Ada_Programming/Tasking.