Australian National University

DISTRIBUTION

Week 10 Laboratory for Systems, Networks and Concurrency

Uwe R. Zimmer

Pre-Laboratory Checklist

- ❑ **You have read this text before you come to your lab session.**
- ❑ **You understand and can utilize message passing locally.**
- ❑ **You have a firm understanding of memory based synchronization.**
- ❑ **You understand and can apply implicit concurrency.**
- ❑ **You can create and control tasks.**

Objectives

This lab will introduce you to the world of distributed computing in practice. You will use BSD socket based communication utilizing TCP/IP connections to combine all of your fellow students' computers into one logical ring.

Interlude: Passing messages everywhere

As you have learnt in the lectures of this course (and in other courses) in a little while, network protocols are usually designed in layers where each layer abstracts a certain aspect of message passing. The lower layers are more concerned about abstracting the physical medium (may it be copper cables in various configurations or glass fibre connections). Then there will be an abstraction to a enable a single, fixed-size message to be routed to the correct node. After that there will be synchronization over longer messages or message sequences from end node to end node in a complex network. This lab will work in this specific abstraction level. In the case of the internet, this will would be called **TCP/IP** or **Transmission Control Protocol** over the **Internet Protocol**. While there are specific interfaces to TCP/IP directly, this is usually to be avoided by anybody but the most brave (unless you write the software for a router) and one mostly utilizes more abstracted interfaces – for instance the universally available **BSD**[1] **socket** interfaces. BSD sockets can be used to send individual datagrams (messages without end-to-end flow control/synchronization, which translate into **UDP/IP** in case of the internet or **User Datagram Protocol**), or to connect end nodes in a way that they can exchange messages in a continuous ("streaming") way and with guarantees for their arrival (synchronization) (which translates into TCP/IP).

Very similar to what you are used to from Ada messages, message passing via BSD sockets assumes a server and a client side, while the server opens a **port** (Ada: `entry`) to the outside which can then be utilized by any client who happens to know about this port.

1 Berkeley Software Distribution

To save you some time, I abstracted over most of the finer details and provided an Ada package for you which contains only the essential steps to communicate with any other node on the internet.

First we need to open a port to the network, such that other computer nodes can call us:

```
      function Open_Server_Port
         (Port        : Port_Type;
          Server_Addr : Inet_Addr_Type := Addresses (Get_Host_By_Name (Host_Name)))
                                                                 return Socket_Type;
```

We only need to nominate a port which should be opened which is simply a number between `0` and `16#FFFF#`. Not all of those port numbers will be freely available to you as some ports have specific meanings. Port `0` is for instance used for internal communications local to your operating system, while for instance port `22` is the default port for your ssh connections[2]. Generally port numbers above `50,000` are not reserved by anybody and you can use them freely for experimentation. The interface also accepts a local IP address which can come in handy if your computer has multiple interfaces. If there is only one IP address associated with your computer, then you can just rely on the default value which uses your computer's first address[3]. The return value is a reference to this port which you just opened so that you can refer to it later within other operations. This first step relates to what you did frequently before when you declared an entry as part of your task definitions.

The second step is to accept a call from the outside, and thus relates to your accept statements from your previous exercises:

```
   function Accept_Connection
      (Server_Socket      :     Socket_Type;
       Connection_Socket  : out Socket_Type;
       Connection_Address : out Sock_Addr_Type) return Stream_Access;
```

You will use the reference which you gained from the previous function as the first parameter here. This call is blocking until a client actually calls in. Once this happened, a channel to the client is been established and you will gain three values in return: The main return value is a reference to the channel on which you can now exchange information (more on this in a moment). The other two parameters are of lesser importance: `Connection_Socket` is for now only needed so that you can close this channel once you had enough from this client and `Connection_Address` tells you from which network node and port this call originated – just in case this is important (in the labs we will use it for debugging so that you know that this call came from your friendly neighbour and not from the other side of the room/planet).

One last missing connection call: A client also needs to have a means to connect to an open server. This is implemented via:

```
   function Connect
      (Server_Addr        :     Inet_Addr_Type;
       Port               :     Port_Type;
       Connection_Socket  : out Socket_Type)    return Stream_Access;
```

The parameters are rather obvious now: We need to provide a network address and port on which to call. In return we will also receive a reference to a communication channel and a `Connection_Socket` which is again needed to close the channel once it won't be needed any more.

By now, both, the server as well as the client possesses a reference to a communication channel which is also bidirectional (i.e. both sides can read and write on this channel). If we assigned this to a variable `Channel`, we can then transfer the value of a variable `Message` of the type `Message_Type` by saying:

```
   Message_Type'Write (Channel, Message);
```

2 You can look up reserved numbers around the internet at: *http://www.iana.org/*

3 Your computer's second address would be: `Addresses (Get_Host_By_Name (Host_Name), 2)` etc.

on one computer node and:

```
Message_Type'Read (Channel, Message);
```

on the connected computer node side.

Keep in mind that the communication systems which we use here has no idea about types. This implies some good and some bad news: The good news is that this technique works without any issues between any self-respecting programming languages. So exchanging messages between say Python and Ada is easy and none of the sides will ever notice that the other side speaks a foreign language. Yet, there is also bad news when things are untyped: Nobody checks the correctness of your types any more. So if one node thinks the current message format is one thing, while the other node thinks it is something different, unbound chaos will prevail. There are systems to amend those issues to a degree (usually called **Middleware**), but they will also reduce your freedom in programming languages and types. For now you don't need to worry about this too much as all your nodes in this lab are programmed in the same programming language which makes consistent typing easier. If you stick with one language and one distribution system then you can guarantee type safety rather easily even if the processor architectures on your computing nodes are different. In almost all other cases, there will be some additional effort required.

So the complete story on the server side could like something like that:

```
declare
   Server_Socket : constant Socket_Type := Open_Server_Port (Port => 60043);

   Connection_Socket  : Socket_Type;
   Connection_Address : Sock_Addr_Type;

   Channel : constant Stream_Access :=
        Accept_Connection (Server_Socket, Connection_Socket, Connection_Address);

   Message : Message_Type;
begin
      ...
   Message_Type'Read (Channel, Message);
      ...
   Message_Type'Write (Channel, Message);
      ...
   Close_Connection (Connection_Socket);
   Close_Server     (Server_Socket);
end;
```

While a client could for instance look like this:

```
declare
   Connection_Socket : Socket_Type;

   Channel : constant Stream_Access :=
      Connect (Server_Addr       => Inet_Addr ("192.168.115.11"),
               Port              => 60043,
               Connection_Socket => Connection_Socket);

   Message : Message_Type := 42;
begin
      ...
   Message_Type'Write (Channel, Message);
      ...
   Message_Type'Read (Channel, Message);
      ...
   Close_Socket (Connection_Socket);
end;
```

But what happens if the same node is client with one channel and server with another? This brings me to your lab exercise for this week …

Exercise 1: Find the ring leader

You arranged tasks in a ring before and passed messages along the ring. This time will arrange all your lab computers into a ring and run a distributed election algorithm on it – in case there would be a need for an arbiter or other form of central management. A classical algorithm for this is has been suggested by Ernest Chang, Rosemary Roberts in 1979[4]. We use it here in a slightly simplified form:

a. Find a unique id in each node. To make this interesting, we use a random value followed by our local address.

b. Each node sends its own id around the ring (as an election bid).

c. If a node receives a bid which is larger than its own bid then it forwards this bid to the next node (and can forget about being elected).

d. If a node receives a bid which is smaller than its own bid then it drops this message without a trace (and can still be hopeful of being elected).

e. If a node receives its own bid then it knows that it is the new leader (and the node can send a confirmation message around the ring to indicate who is boss).

You can download a framework which has this algorithm already implemented and also provides you with a command line interface to set the address and port of the next node in your ring. Once you solved this exercise, you should team up with other students in your lab which also have a solution and form a ring of connected computer nodes with them (and then run the above distributed election algorithm).

Your job is to solve the following problem: In order to set up such a ring you need to complete two blocking calls, namely `Connect` and `Accept_Connection`. Obviously, if all nodes block on `Connect` first and `Accept_Connection` second, then the ring will never be established. The other way round does not work either which you will agree with after a few seconds of reflection. So it sounds like a classical concurrency/synchronization job which should find its perfect match with you.

These two code blocks need to be embedded into your provided node code somehow:

```
Put_Line (“Waiting for previous node to connect”);

Incoming_Channel :=
   Accept_Connection
      (Server_Socket, Incoming_Connection_Socket, Incoming_Connection_Address);

Put_Line (“--> Previous node connected from: “
          & Image (Incoming_Connection_Address));
```

and:

```
Put_Line (“Waiting for next node to become available”);

Outgoing_Channel :=
   Connect (Server_Addr       => CLP.Next_Addr,
            Port              => CLP.Next_Port,
            Connection_Socket => Outgoing_Connection_Socket);

Put_Line (“--> Connected to next node at: “
          & Image (CLP.Next_Addr) & “:” & Port_Type’Image (CLP.Next_Port));
```

Make sure that both, `Incoming_Channel` as well as `Outgoing_Channel` have a valid value before they are being used in subsequent read and write operations.

4 in their short paper: *An improved algorithm for decentralized extrema-finding in circular configurations of processes*; Communications of the ACM, Volume 22 Issue 5, Pages 281-283, May 1979

Once your code works, you can use the command line options:

```
      [-a {IP address of this node : String    }]
      [-n {IP address of next node : String    }]
      [-p {This node's port        : Port_Type }]
      [-q {Next node's port        : Port_Type }]
```

to test your ring by starting just two nodes on your own computer first. For this you open two terminals, navigate in both into the `Executable` directory of your current lab and type in one terminal:

```
  ./ring_node -p 50041 -q 50042
```

and in the other terminal:

```
  ./ring_node -p 50042 -q 50041
```

If this works you will also have seen your own IP address[5] (as part of the output of your own program) which might have been for instance `192.168.115.11` and can hand this IP address to your friendly neighbour, who can then type on his computer:

```
  ./ring_node -n 192.168.115.11
```

to connect to your computer. Accordingly you use your neighbour's IP address on your computer in the same way to form a two-node ring.

Now you get the hang of the idea and you can collect more IP addresses around the lab and arrange for a larger and larger ring of lab computers.

Don't forget to submit your competed, zipped project code to the *SubmissionApp* under "Lab 11 Ring node" for code review by your peers and us.

What form of message passing did you exercise in this lab? Was this synchronous or asynchronous message passing? Is reading and/or writing to a channel potentially blocking? – and if so: until when? You can conclude some of the answers from your working code. Research and discuss the options for the remaining answers with your tutor and class mates.

Exercise 2: Foreign languages

Program a ring node with follows the exact some protocol as the code which has been provided to you in your second favourite programming language. BSD Socket interfaces will be available in all self-respecting languages, yet you need to make sure that you will conform to a common message format. If your second favourite language is not flexible enough to produce the current format, you can as well change the format on the Ada side. For that you might look into the `Interfaces.C` packages.

Submit a zip archive of your code in any language to the *SubmissionApp* under "Lab 11 Ring node foreign" for code review by us.

Make Sure You Logout
to Terminate Your Session!

Outlook

During the last few days before the second assignment deadline spend time to polish things and make sure that your code and diagram is spot on.

5 You can also find out your local IP address in general by typing: `ip addr`