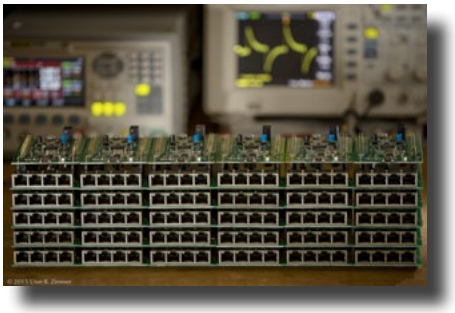
Australian National University

PASSING THE MESSAGE

Assignment for all students of
Systems, Networks and Concurrency

This is the second, carefully marked assignment of the course. Extra care is expected on all levels. A good solution will have a surprisingly small amount of code. Make sure that this code is excellently written – in whichever language you prefer.



Overview

Routers are basic ingredients of most distributed systems. They commonly have no a-priori knowledge about the wider network topology in which they are embedded. Nevertheless they are expected to send messages via a network port which will move those messages closer to their destinations. Many constraints can be taken into consideration for a practical router, yet this assignment focuses on sending messages on the shortest paths to their destinations.

What an individual router needs to provide to clients:

- **Accept a message from a client** (in an predefined format). The message also contains a destination `id`. The expectation is that this message will eventually end at the router of this `id`.
- **Store and provide received messages** for a client to pick up (in an predefined format).

What is accessible to an individual router:

- The **local network ports** which are connected to other routers.
- The **Id's** of all neighbouring routers. In the physical world this would need to be established by means of initial communications first, yet we assume here that this already happened.
- The **total number** of routers as well as the fact that they are numbered consecutively from 1 to the total number of routers. In a real router you would use hash tables for the router id's.

What the routers need to agree upon before they are deployed:

- One or multiple message formats which are exchanged between routers (in addition to the already given formats used to send to and received from clients).
- The semantics for those messages.

What a router can do:

- **Send** messages in any of the globally defined inter-router formats over any of its ports to any of its directly connected neighbours.
- **Receive** messages on any local port and process their contents.

What a router cannot do is to sneak behind the scenes and extract the global topology of the network from a central place in a direct way. Routers can still organize themselves in whichever form they see fit. Thus a central place could for instance be one router which by election, or other means nominated itself as a central place. Communication with this router would need to go through the normal network channels though – no secret direct wires to a central coordinator. Yet there does not need to be any central instance at all – distributed solutions are usually also more robust.

How to implement a router

You can implement this router in any language or environment, as long as it roughly follows the frame described above. One essential feature for your implementation needs to be kept in mind: the connections between routers are “wires” – not software buffers. A close software equivalent to a wire is synchronous message passing. Any buffer which might be needed, will need to be implemented as part of the router. You can emulate synchronous (or otherwise “immediate”) communication by means of multiple asynchronous messages or by means of asynchronous messages combined with signals/interrupts.

There is a framework available to you in Ada which generates a number of classical network topologies and places router tasks inside a chosen topology. It also runs a simple evaluation which sends messages from each router to every other one and takes notes about the number of hops the message passed through.

If you decide to write it in another language than Ada, then you will need to program this simple infrastructure as well (details below). Simply contact us which your specific plans and we will see what is achievable with reasonable effort in your chosen environment.

The Ada framework

The program `test_routers` can be fully controlled via command line parameters:

```
accepted options:
   [-t {Topology            : String   }] -> CUBE_CONNECTED_CYCLES
      by Size            : Line, Ring, Star, Fully_Connected
      by Degree, Depths  : Tree
      by Dimension, Size : Mesh, Torus
      by Dimension       : Hypercube, Cube_Connected_Cycles,
                           Butterfly, Wrap_Around_Butterfly
   [-s {Size               : Positive }] ->  20
   [-g {Degree             : Positive }] ->   3
   [-p {Depths             : Positive }] ->   4
   [-d {Dimension          : Positive }] ->   3
   [-c {Print connections  : Boolean  }] -> TRUE
   [-i {Print distances    : Boolean  }] -> TRUE
   [-w {Routers settle time : Seconds  }] ->  0.10
   [-o {Comms timeout       : Seconds  }] ->  0.10
   [-m {Test mode           : String   }] -> ONE_TO_ALL
      Available modes: One_to_All, All_to_One
   [-x {Dropouts           : Natural  }] ->   0
   [-r {Repeats            : Positive }] -> 100
```

Figure 1: Command line options

Available network topologies are denoted by name (out of the list above) together with the parameters which are required for a given topology. Thus you can test your routers in a 4-d hypercube by: `./test_routers -t Hypercube -d 4`

You can allow your routers some time to get themselves organized and specify for instance:

```
./test_routers -t Hypercube -d 4 -w 0.2
```

for a 200 ms preparation phase during which no client messages will be sent through the network. After this settling time for the routers, the evaluation phase starts and a messages will be sent via every router to every other router. Those messages will be sent in groups, i.e. all messages originating from a specific router will be sent in bulk (“One to all”). This will be changed via the command line parameter `-m All_To_One` to a schema, where all messages which are destined to a specific router will be send in bulk. The default time-out for any network activity is 100 ms (but can also be adjusted via the command line interface). If a router does not respond after this time-out then the communication test is noted as unsuccessful. From the testing environment it cannot be distinguished whether the message was not delivered or whether the destination router is not responsive, as

```
---------------------- Instantiating router tasks -------------------------------
  => Routers up and running
---------------------------- Waiting ------------------------------------------
  Time for routers to establish their strategies :  0.10 second(s)
------------------------- Measurements ----------------------------------------
Number of runs : 100
Minimal hops   :   1
Maximal hops   :   6
Average hops   :   3.22

       1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
    +-------------------------------------------------------------------------+
  1 |  .                 2  2  3  2  3  4  3  4  3  3  2  4  4  3  5  4  4  6  5  5|
  2 |     .        2  3  3  2     2  3  4  4  3  3  2  4  5  4  4  3  5  6  5|
  3 |        .  2  3  3  3  2  3  4  4  5  2  2     3  4  4  4  3  4  5  5  6|
  4 |     2  2  .           4  3  4  3  2  3  4  4  3  3  3  2  6  5  5  5  4  4|
  5 |  2  3  3     .        3  4  4  2     2  4  5  4  3  3  2  5  6  5  4  4  3|
  6 |  2  3  3        .  4  4  5  3  2  3  3  4  4  2  2        5  5  6  4  3  4|
  7 |  3  2  3  4  3  4  .              2  2  5  4  4  6  5  5  3  3  2  4  4  3|
  8 |  2     2  3  4  4     .     2  3  3  4  4  3  5  6  5  3  3  2  4  5  4|
  9 |  3  2  3  4  4  5        .  2  3  3  4  3  4  5  5  6  2  2     3  4  4|
 10 |  4  3  4  3  2  3     2  2  .           6  5  5  5  4  4  4  4  3  3  3  2|
 11 |  3  4  4  2     2  2  3  3     .        5  6  5  4  4  3  4  5  4  3  3  2|
 12 |  4  4  5  3  2  3  2  3  3        .     5  5  6  4  3  4  3  4  4  2  2     |
 13 |  3  3  2  4  4  3  5  4  4  6  5  5  .              2  2  3  2  3  4  3  4|
 14 |  3  3  2  4  5  4  4  4  3  5  6  5     .     2  3  3  2     2  3  4  4|
 15 |  2  2     3  4  4  4  3  4  5  5  6        .  2  3  3  3  2  3  4  4  5|
 16 |  4  4  3  3  3  2  6  5  5  5  4  4     2  2  .           4  3  4  3  2  3|
 17 |  4  5  4  3  3  2  5  6  5  4  4  3  2  3  3     .        3  4  4  2     2|
 18 |  3  4  4  2  2     5  5  6  4  3  4  2  3  3        .  4  4  5  3  2  3|
 19 |  5  4  4  6  5  5  3  3  2  4  4  3  3  2  3  4  3  4  .              2  2|
 20 |  4  4  3  5  6  5  3  3  2  4  5  4  2     2  3  4  4     .     2  3  3|
 21 |  4  3  4  5  5  6  2  2     3  4  4  3  2  3  4  4  5        .  2  3  3|
 22 |  6  5  5  5  4  4  4  4  3  3  3  2  4  3  4  3  2  3     2  2  .     |
 23 |  5  6  5  4  4  3  4  5  4  3  3  2  3  4  4  2     2  2  3  3     .  |
 24 |  5  5  6  4  3  4  3  4  4  2  2     4  4  5  3  2  3  2  3  3        .|
    +-------------------------------------------------------------------------+
```

Figure 2: Measurements

the internal states of the routers are not accessed. Next, the results of the communication tests are displayed (Figure 2). In the provided framework the routers do not exist yet, thus this section will not appear up on your screen until you provide a router implementation.

The minimum number of hops for a message should be '1', and the largest measured number of hops should represents the diameter[1] of the chosen topology. The average number of hops is characteristic for a specific topology. By using the command line option `-i True` you will also see the full table of individual measurements between all nodes. As we only consider bidirectional connections, the number of hops should be identical in both directions between any two routers. If the measurements are different, warnings will be displayed. 1-hop connections are left blank to make the matrix somewhat more readable.

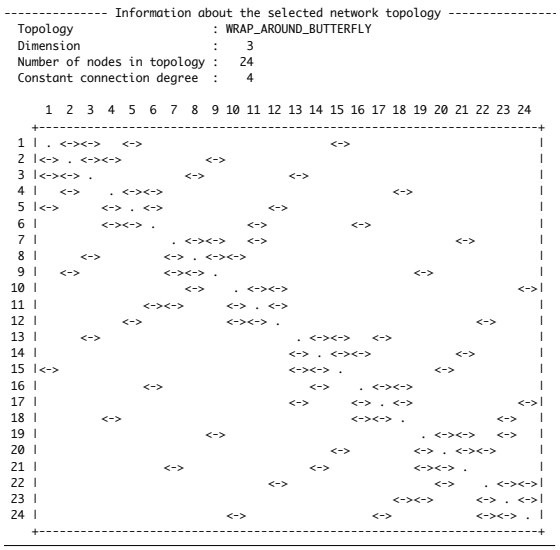


Figure 3: Topology

The last section (Figure 3) which will appear on screen gives general parameters of the current topology, like the minimum and maximal connection degree and the total number of nodes. You can also print the connections inside the topology as a matrix – which should be the mirror image of the empty spots in the measurement matrix. Double arrows denote bidirectional connections.

The individual routers need to respond to the following four pre-defined entries:

```
task type Router_Task (Task_Id  : Router_Range := Draw_Id) is

   entry Configure (Links : Ids_To_Links);

   entry Shutdown;

   entry Send_Message    (Message :     Messages_Client);
   entry Receive_Message (Message : out Messages_Mailbox);
```

`Configure` is only called once right after the task has been created. The existing links to neighbouring routers are delivered here in the form of an array over the router id's in which connected routers will appear while all other entries are `null`.

```
type Ids_To_Links is array (Router_Range) of Router_Task_P;
type Router_Task_P is access all Router_Task;
```

Two message formats are already defined, yet you need to add at least one more message format and entry which you will be using for communications between routers.

```
type Messages_Client is record
   Destination : Router_Range;
   The_Message : The_Core_Message;
end record;

type Messages_Mailbox is record
   Sender      : Router_Range;
   The_Message : The_Core_Message;
   Hop_Counter : Natural := 0;
end record;
```

The purpose of `Shutdown` (which will be called upon once after all tests have been performed) is to allow for a graceful way to stop operations. If your router accepts this call, it is assumed that

[1] The **diameter** of a given topology is the greatest distance between any two nodes.

the router task (and all tasks or resources initiated by it) will terminate eventually. If the call to `Shutdown` is not accepted then an `abort` of this task is attempted by the environment.

All this sounds complicated but it really isn't. The core of the assignment is just to decide in which direction to forward a specific message and to use your knowledge from the course to implement your strategy without locking up, unnecessary busy-waiting loops or losing information. In short: your solution should be implementing a router which is responsive on all ports and directs traffic onto the optimal routes.

Implementation alternatives

The basic requirements for an implementation in another language are:

- Option to select between and parametrize topologies as provided in the framework.
- Make sure that you routers do not access any information other than the initial connection list to neighbouring routers and the information which they gain by exchanging messages.
- Instantiate the number of routers required to populate the topology.
- The routers need to be instantiated as processes, threads, tasks, or whatever your operating system or language will provide as concurrent entities.
- Provide each router entity with communication links to neighbouring routers. Those links shall resemble a "wire" semantic, i.e. synchronous forms of communication.
- Run a simple test by sending messages from every router to every other router and measure the actual number of hops the messages took.
- Summarize the results as maximal as well as average number of measured hops.

If this sounds too demanding for your chosen environment, then contact us and we will work out a plan which can be done – a concurrent implementation of the router tasks is obviously mandatory though.

Candidates include Erlang, Go (basic templates provided for both), Rust, Smalltalk, LabVIEW, Occam, OpenMP… just as long as you can rely on message passing (synchronous message passing might be emulated by asynchronous message passing) and support for concurrency.

If your environment cannot be reproduced by us with reasonable effort in order to run your program then we will potentially ask you to provide more detailed documentation of your tests.

Extension

As you have been attentive you will have noticed a command line parameter (`-x`) which controls the number of routers dropping out. After the initial configuration phase (and before normal traffic starts) a specific number of random routers can be powered down purposefully. Many network topologies offer multiple routes to the same node, so the resulting network may still be fully operational (minus any direct communication to the powered down nodes obviously).

Your extension assignment is to design routers which are robust with respect to such losses in the network and will still deliver the packets to the destination, even if the original/optimal path became unavailable. In the provided Ada framework routers will detect the loss of an immediate neighbour by receiving a `Tasking_Error` exception when trying to communicate to an already terminated router task. Handle this exception and provide an alternative route.

Reflect first for a moment on the following issues (all of which will be related to specific topologies):

- How many dropped out routers can you handle and still guarantee delivery?
- How does the number of dropped out routers affect the performance? You may want to distinguish the situation when you first detect the failure from the following routing operations.
- Can you respond locally or will you need to propagate the detection of the failure?

This extension is meant for the student who wants to dig deeper and find out more about real-world routing.

Deliverables

You do *not* need to provide a rationale or any report of sorts if you believe that your design is obvious and a few comments inside your code will make the idea sufficiently clear.

Yet you need to provide two items:

a. The sources for a working router (and an environment to test them in for the students not using the provided Ada framework). Zip up the `Router` directory (which should hold all sources which you manipulated or added) for submission.

b. A *graphical* representation of the workings inside your router (as a pdf file named `Diagram.pdf`). Add this file to your `Router` directory before you zip it up.

In any case, we will read your sources carefully. The core part of your router will be small – take your time to structure this part well.

Different entities in your graphical representation will have different meanings: Use different lines / arrows / colours / shadings / etc. to indicate what is what and provide a legend. Before you start drawing: decide what the main aspect of our diagram should be. Will you primarily provide a call-graph / data-flow-graph / synchronization-graph / dependency-graph / etc. or a mixture of multiple aspects. If you need multiple diagrams for multiple aspect, then please provide multiple diagrams (in the same pdf file). Your diagram will likely contain text labels, yet should not include full sentences or paragraphs.

There is no predefined template for your diagram as the idea is that you communicate your concept with it – whichever format supports this best is for you to decide.

Tutor-eyesight-health-warning: Make sure that your graphics will appear technically in vector (scalable) format inside your pdf file – *never* as an image made out of pixels!

Criteria

When you design your system, you might want to keep the following criteria in mind:

- Messages shall *arrive* at the destination router *eventually* (this is the minimal requirement).
- The *number of hops* should be *minimal*.
- The *setup time* should be proportional to the number of nodes or even better to the diameter of the network. Beware of combinatorial explosions.
- *Performance* of the solution. Did you arrange concurrent entities inside your router to guarantee minimal delivery delays? Assume that your router can be provided with any number of physical concurrent entities.
- *Elegance* of the solution. Elegance is often a measure related to compactness and simplicity but not always. You will see whether your design is elegant once you reevaluate/scrutinize your sources a few days after you achieved a running system.
- *Clarity* and *expressiveness* of the provided graph. A meaningful graph will use a clear legend of graphical elements to distinguish different issues.
- *Robustness* of the routing algorithm. Could you design an algorithm which handles dropped out routers gracefully and with minimal effect to performance? If you do not target a high distinction in this assignment, you may consider dropping this criterion and focus on the remaining ones (this one is hard).

Final words

Implementing this basic router will give you a good starting point to look further and deeper into the world of distributed systems. The assignment combines what you now know about concurrent systems and expands your practical abilities towards problems in distributed environments. Getting closer to the end of the course, try to recollect the new concepts and tools which you have now at your disposal. While you should not expect that we made an experienced designer of concurrent and distributed systems out of you quite yet, you do have now a

substantial amount of knowledge to already join in at discussions among professionals in the field.

Outlook

Students in Real-Time and Embedded Systems (final year course) will be working with actual hardware (below) and concurrent and distributed algorithms under strict real-time constraints. There is strong demand for outstanding professionals who can handle high integrity systems interacting with the physical world. Your course provides many foundations for any such career – academic or industrial.