Distance Vector Routing using SCOOP

Semester project, ETHZ
December 2003 – March 2004

Student Emmanuel Python (7th semester)
epython@student.ethz.ch

Supervisors Sébastien Vaucouleur, Dr Arnaud Bailly

1 Index

2 Project Description and Management

2.1 Overview

The purpose of this project is to design and implement a Distance Vector Routing algorithm using SCOOP (SCOOPLI). The resulting application should be able to simulate the algorithm on a network where every router is modelled as a separate object.

2.2 Scope of the work

- The application (GUI) provides the following features:
    - Implementation of the Distance Vector Routing algorithm. (Every router has to be represented using a separate object)
    - Use of a file to read the network configuration
    - Send of a packet to a certain destination
    - Modification of the cost of a link between routers
    - Display of routing tables
    - Display of the route of forwarded packets
    - Display of the network topology
- Benchmarking of the application
- Report of the analysis of the SCOOP model and the Distance Vector Routing algorithm
- Report of the overall experience using SCOOPLI (ease of design, ease of debugging, suggestions for future releases)

2.3 Objectives and priorities

The main priority of the project is to provide the basic functionality of the application, namely the implementation of the Distance Vector Routing algorithm.

2.4 Method of work

The application has been developed using the cluster model of the software lifecycle [2] without the generalization step.

2.5 *Quality management*

2.5.1 Documentation

- A thesis report documenting and explaining the results

2.5.2 Validation steps

- All major changes to the application have been delivered to the supervisors
- Validation of the application using test cases

2.6 *Project plan*

Tasks	Week (2003-2004)																	
	48	49	50	51	52	1	2	3	4	5	6	7	8	9	10	11	12	13
project plan	8																	
SCOOP + SCOOPLI analysis	8																	
algorithm spec. and design		8	8															
algorithm implementation				8			8	8										
GUI spec. and design					Holidays				8	8					Examinations			
GUI implementation											8	16						
benchmarking of test cases													8					
documentation												16	24					8
defense																		8

Total numbers of hours (indicated in the small squares): 160

3 Theory

3.1 The SCOOP model

The SCOOP model (Simple Concurrent Object-Oriented Programming) provides a simple mechanism to build concurrent and distributed systems. SCOOP takes object-oriented programming and concepts of Design by Contract [2] as given with only one extension: the keyword *separate*. Since the programmers do not need to deal with low-level concepts like for example semaphores or mutexes, it should be very simple to write applications with SCOOP.

The following explanations about the SCOOP model are closely based on the papers [1] and [3].

3.1.1 Architecture

SCOOP is the top layer of a two-level architecture. This upper level is platform-independent and contains the implementation of the separate mechanism to perform concurrent computations. The bottom layer is a practical concurrent architecture like .NET that provides remoting and threading mechanisms.

3.1.2 Processors

SCOOP uses the concept of processor to add concurrency to the framework of sequential object-oriented computation. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. A processor can be for example a peace of hardware, a process, a thread or an application domain (in .NET). Note that a concurrent system may have any number of processors. We can see in the figure 1 an example with two objects (o_1 and o_2) that use different processors. The introduction of the processor concept allows the asynchronous call of the feature *f* (o_1 doesn't wait for the call on o_2 to terminate).

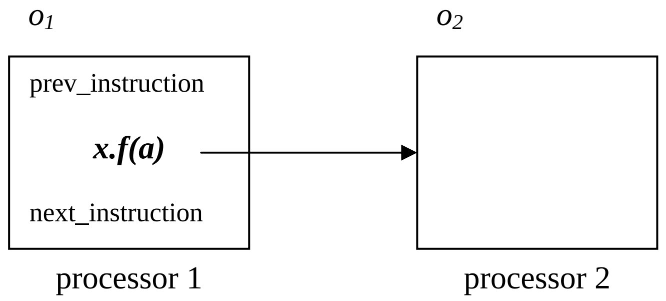


Figure 1 : the processor concept

3.1.3 Separate objects and calls

To unambiguously indicate that an object is handled by a different processor, we use the keyword *separate* like in "x: **separate** SOME_CLASS". With such a declaration, any creation instruction will spawn off a new processor to handle separate calls on the object. We say that *x* is a separate entity and *x.f(a)* a separate call.

The validity of separate calls (its target is a separate entity) is governed by four consistency rules [2] :

- If the source of an attachment (assignment instruction) is separate, its target entity must be separate too.
- If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.
- If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.
- If an actual argument of a separate call is of an expended type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

3.1.4 Access control policy

For a separate call to be valid, the target of the call must be a formal argument of an enclosing routine. Such "embedding" of separate call in routines allows exclusive locking of separate objects. When one or more arguments of a routine are separate objects, the client must obtain exclusive locks on all these objects before executing the routine. The access control policy becomes very simple: at any given time, a processor in charge of the separate object can execute at most one routine.

It is also possible to introduce *wait conditions* using (enclosing) routine preconditions and to interrupt the execution of a routing using *duel.*

3.1.5 Synchronization

SCOOP uses the mechanism *wait by necessity* for interactions between a client and its supplier. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in "value:=x.some_query".

3.1.6 Distributed computation

One of the strongest ideas behind SCOOP is that the application could be executed on a local machine and on several distributed machines without changes in the code. The mapping between processors and physical resources is specified in a Concurrency Control File and not directly in the software text. All the tedious work to make the distributed architecture completely transparent for the application is done by SCOOP.

3.2 SCOOPLI

SCOOPLI is a library implementation of SCOOP using the .NET framework where the processor concept of SCOOP is directly mapped to application domains. The SCOOPLI library is the upper layer of the SCOOP architecture that contains the implementation of the general concurrency mechanism.

3.2.1 Basic concepts

The library relies on the concepts of *separate client* and *separate supplier* with the following meaning:

Let S be a class. A class C, which contains a declaration of the form x:S, is said to be a client of S. S is then said to be a supplier of C. [2]

3.2.2 Interface

Declaration of a separate supplier

Instead of using the keyword *separate*, the SCOOPLI library uses multiple inheritance. All suppliers must inherit from SEPARATE_SUPPLIER:

```
class SEPARATE_S
        inherit
                SEPARATE_SUPPLIER
                S
        …
end

x: SEPARATE_S
```

Declaration of a separate client

In SCOOPLI we must explicitly declare the separate client. It is done by using multiple inheritance, every client must inherit from SEPARATE_CLIENT:

```
class MY_CLASS
        inherit
                SEPARATE_CLIENT
        feature
                x: SEPARATE_S
        …
end
```

Separate procedure call

If we want to do a separate call like “x.f(a)”, it is requested to embed the call in a routine:

```
-- in class MY_CLASS
r(a_x:SEPARATE_S; a: SOME_CLASS) is
                -- execute a_x.f(a)
        do
                        sep_handler.execute(Current,a_x, agent a_x.f(a))
        end
…
sep_handler.execute_routine(Current,[x], agent r(x,a),Void)
```

Let’s have a closer look at the routines *execute* and *execute_routine*:

execute(caller: SEPARATE_CLIENT; supplier: SEPARATE_SUPPLIER; procedure: PROCEDURE[])

- *caller:* denotes the current separate client object.
- *supplier:* denotes the separate supplier object on which the separate call to procedure is made.
- *procedure:* denotes the routine to be called on the separate supplier object.

execute_routine(caller: SEPARATE_CLIENT; requested_objects: TUPLE[]; action PROCEDURE[]; wait_condition: FUNCTION[])

- *caller:* denotes the current separate client object.
- *requested_objects:* denotes the tuple of objects on which exclusive locks should be acquires before calling *action*
- *action:* denotes the routine to be called on the separate client object
- *wait_condition:* denotes the Boolean function representing the wait condition for the call.

3.3 Distance Vector Routing

Routing in networks is an instance of the problem of path finding in graphs. The Bellman's shortest path algorithm developed in 1957 provides the basis for the distance vector method. Ford and Fulkerson converted bellman's method into a distributed algorithm suitable for implementation in large networks in 1962. Now, protocols based on their work are often referred to as 'Bellman-Ford' protocols.

3.3.1 The Distance Vector Routing Algorithm

Figure 2 shows the routing tables for each routers of the network represented in the figure 3 after the convergence of the algorithm. Each row provides the routine information for packets addressed to a given destination. The link field specifies the router attached to the outgoing link. The cost field is simply a calculation of the vector distance.

Routings from 1		
To	*Link*	*Cost*
1	Local	0
2	2	10
3	2	30
4	4	30
5	2	40

Routings from 2		
To	*Link*	*Cost*
1	1	10
2	Local	0
3	3	20
4	1	40
5	3	30

Routings from 3		
To	*Link*	*Cost*
1	2	30
2	2	20
3	Local	0
4	2	60
5	5	10

Routings from 4		
To	*Link*	*Cost*
1	1	30
2	1	40
3	1	60
4	Local	0
5	1	70

Routings from 5		
To	*Link*	*Cost*
1	3	40
2	3	30
3	3	10
4	3	70
5	Local	0

Figure 2: routing tables of the network

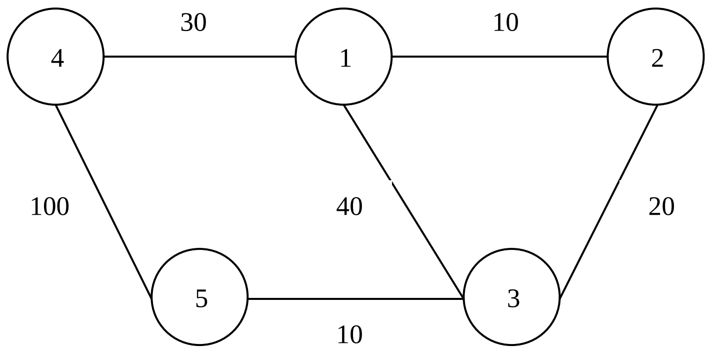


Figure 3: graph of the network

The routing tables contain a single entry for each possible destination, showing the next hop that a packet must take towards its destination. When a packet arrives at a router, the destination address is extracted and looked up in the local routing table. The resulting entry identifies the outgoing link that should be used to route the packet.

Now lets us consider how the routing tables are build up. The routing information is proceeded in a distributed fashion and routers exchange information about the network with theirs neighboring nodes by sending their routing table. The actions performed at a router are described informally as follows [4]:

1. Periodically, and whenever the local routing table changes, send the table to all accessible neighbors.
2. When a table is received from a neighboring router, if the received table shows a route to a new destination, or a better (lower cost) route to an existing destination, then update the local table with the new route. If the table was received on link *n* and it gives a different cost than the local table for a route that begins with link *n*, then replace the cost in the local table with the new cost. This is done because the new table was received from a router that is closer to the relevant destination and is therefore always more authoritative for routes that pass through it.

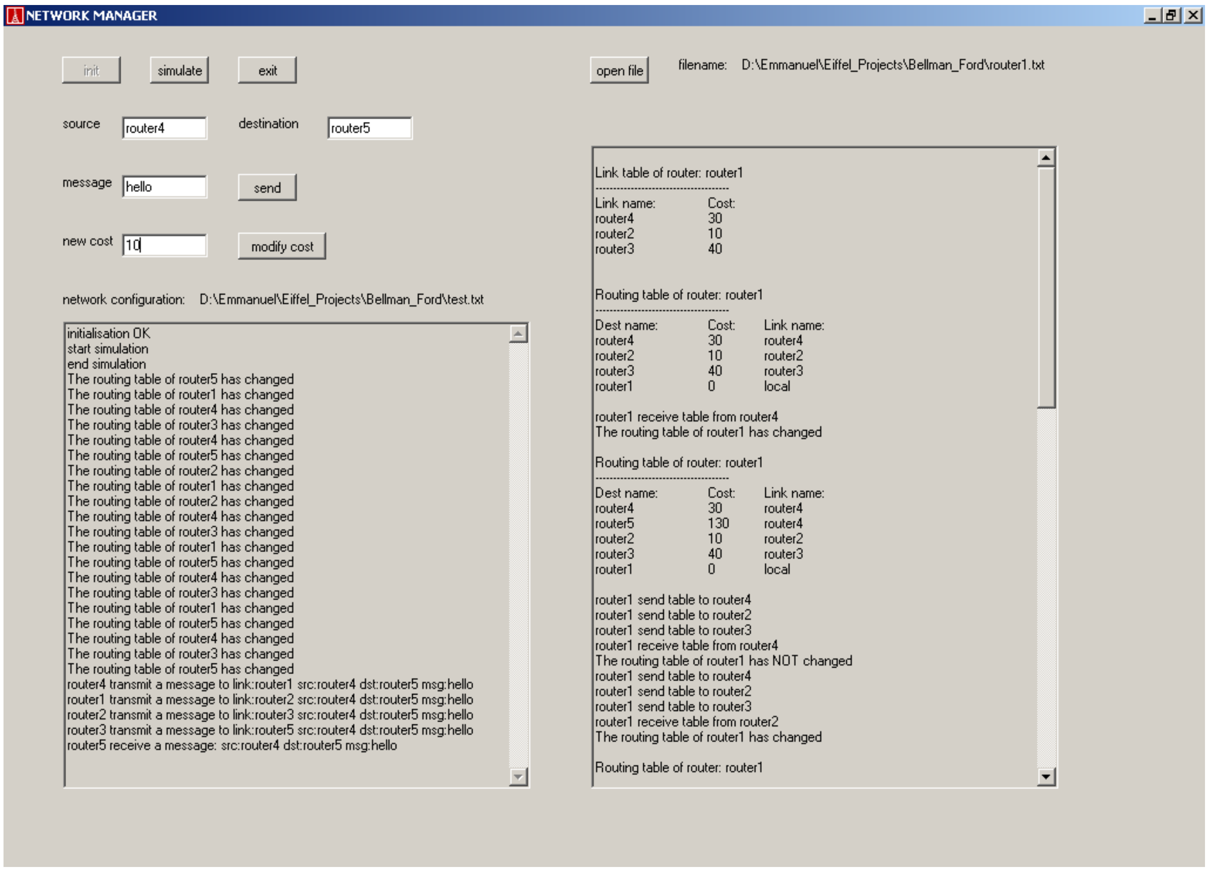
The routing tables will converge on the best routes to each destination whenever there is a change in the network. The algorithm is more precisely described by the following pseudo-code [4]:

```
Send: each t seconds or when the local routing table Tl changes, send Tl on each non-faulty outgoing
link.

Receive: whenever a routing table Tr is received on link n:
    from
        the first row Rr in Tr
    until
        the last row in Tr
    loop
        if Rr.link /= n then
            Rr.cost := Rr.cost + link's cost
            Rr.link := n

            If Rr.destination is not in Tl then
                add Rr to Tl      -- add new destination to Tl
            else
                from
                    the first row Rl in Tl
                until
                    the last row
                loop
                    if (Rr.destination = Rl.destination) and
                    (Rr.cost < Rl.cost or Rl.link = n)
                    then
                        Rl:=Rr
                    end
                    -- Rr.cost < Rl.cost : remote node has better route
                    -- Rl.link = n : remote node is more authoritative
                end
            end
        end
    end
```

Note that the test "*Rr*.link /= *n*" allows solving the *count-to-infinity* problem explained in [5]. The cost to the destination *D* has no effect on the link (router) that packets for *D* are sent on. This solution is named *Split Horizon*.

4 The Application

Two different parts compose the network manager shown in Figure 4. The left part contains several buttons to control the execution of the Distance Vector Routing algorithm and the right part allows opening and displaying log files.



source: router4   destination: router5

message: hello

new cost: 10

network configuration: D:\Emmanuel\Eiffel_Projects\Bellman_Ford\test.txt

```
initialisation OK
start simulation
end simulation
The routing table of router5 has changed
The routing table of router1 has changed
The routing table of router4 has changed
The routing table of router3 has changed
The routing table of router4 has changed
The routing table of router5 has changed
The routing table of router2 has changed
The routing table of router1 has changed
The routing table of router2 has changed
The routing table of router4 has changed
The routing table of router3 has changed
The routing table of router1 has changed
The routing table of router5 has changed
The routing table of router4 has changed
The routing table of router3 has changed
The routing table of router1 has changed
The routing table of router5 has changed
The routing table of router4 has changed
The routing table of router3 has changed
The routing table of router5 has changed
router4 transmit a message to link:router1 src:router4 dst:router5 msg:hello
router1 transmit a message to link:router2 src:router4 dst:router5 msg:hello
router2 transmit a message to link:router3 src:router4 dst:router5 msg:hello
router3 transmit a message to link:router5 src:router4 dst:router5 msg:hello
router5 receive a message: src:router4 dst:router5 msg:hello
```

filename: D:\Emmanuel\Eiffel_Projects\Bellman_Ford\router1.txt

```
Link table of router: router1
------------------------------------
Link name:          Cost:
router4             30
router2             10
router3             40


Routing table of router: router1
------------------------------------
Dest name:          Cost:     Link name:
router4             30        router4
router2             10        router2
router3             40        router3
router1             0         local

router1 receive table from router4
The routing table of router1 has changed

Routing table of router: router1
------------------------------------
Dest name:          Cost:     Link name:
router4             30        router4
router5             130       router4
router2             10        router2
router3             40        router3
router1             0         local

router1 send table to router4
router1 send table to router2
router1 send table to router3
router1 receive table from router4
The routing table of router1 has NOT changed
router1 send table to router4
router1 send table to router2
router1 send table to router3
router1 receive table from router2
The routing table of router1 has changed

Routing table of router: router1
```

Figure 4: main window of the application

The left part

Init: This button is used to load the network configuration file (.txt) to initialize the link table and the routing table of each router. A configuration file (see Figure 5) contains the name of each router, the links between them and the cost of each link.

Simulate: This button starts the execution of the algorithm. In fact all the routers send their routing table to theirs neighbors and then apply the algorithm for the received routing tables. Note that the routers don't periodically send their routing table. If we modify the network (modify the cost of a link), we must re-execute the algorithm.

Exit: This button is used to exit the application.

Send: This button allows sending a message between routers. It uses the source and the destination indicated in the corresponding labels.

Modify cost: This button is used to modify the cost of a link between two executions of the algorithm. The link is specified by the destination and the source indicated in the corresponding labels. The value of the cost is specified between 0 and 1000 and we can easily use the maximum value to simulate a faulty link.

Text area: The text area display information about the algorithm's execution. It shows if a routing table has been modified, the route of a packet, several errors in the input, etc.

```
router: router1
router: router2
router: router3
router: router4
router: router5
link: router1 router2 10
link: router1 router3 40
link: router1 router4 30
link: router2 router3 20
link: router4 router5 100
link: router3 router5 10
```

Figure 5 : network configuration file of the figure 3

The right part

Open file: This button is used to display a text file in the text area. It allows in particular showing the log file of each different router. The name of a log file is simply the name of the router followed by the extension ".txt". Each log file contains all the information about the execution of the algorithm on the current router (sent tables, received tables, sent packets, received packets, updated local routing table, updated link table, etc.).

5 Design

5.1 Design with SCOOP (SCOOPLI)

At the moment, the SCOOPLI library doesn't provide facilities to create an application that works in a concurrent as well as in a distributed context. The actual version of SCOOPLI is limited to the concurrent context but it isn't a problem for the application: all the future tedious work for the distributed context will be done transparently by the library and the design will remain the same for both contexts.

The design of the concurrent application using SCOOPLI is rather simple (see Figure 6). Each router (ROUTER) is a separate object (SEPARATE_SUPPLIER, SEPARATE_CLIENT) and all these routers are controlled by a manager (MANAGER). With the benefits of this unique manager, it is possible to control the entire network in the same place. A graphical user interface (MANAGER_WINDOW) allows the communication between the user and the network manager. The GUI and the manager are also designed as separate object because all the interactions between the actors of the network are concurrent.

The router has two different hash tables: a routing table (ROUTING_TABLE_ROW) and a link table (LINK_TABLE_ROW). The most important classes will be discussed in the following sections.

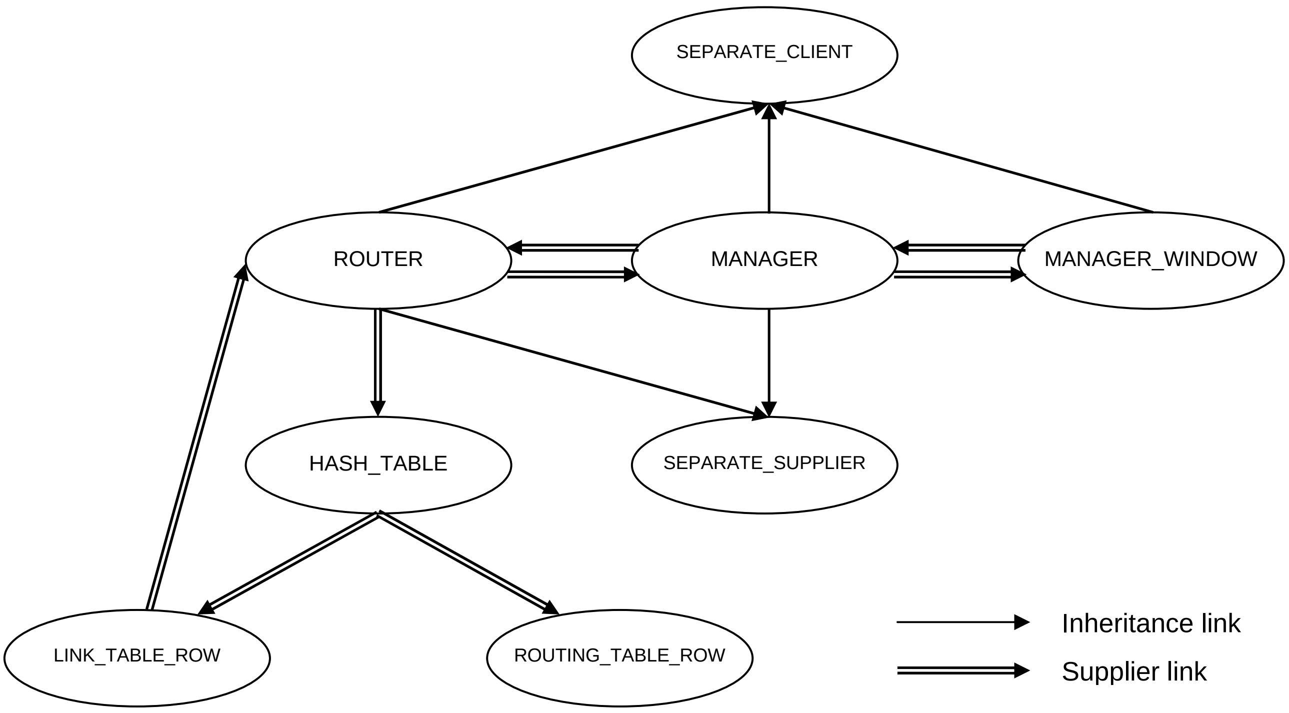


Figure 6: class diagram of the application

5.1.1 The router

As written before, a router is a separate object: it is a separate supplier and a separate client of the manager and other routers. A router has the following attributes: a ***name***, a ***file*** to log information, a reference to the ***manager*** and two hash tables: the ***routing_table*** and the ***link_table***.

Each row of the routing hash table is accessible through a key specified by the ***destination name*** (the name of a router). A row is an object of type ROUTING_TABLE_ROW that contains two attributes, the ***cost*** to the destination and the ***link name***. In the same way, the link hash table has a key specified by the ***link name***. A row of the link table is of type LINK_TABLE_ROW and has also two attributes: the ***cost*** of the link and a reference to the ***link*** (a router).

The different features of the ROUTER class are described below:

add_link:	This routine is used by the manager to add a link to the current router.
init_routing_table:	This routine is used by the manager to initialize the routing table of the current router.
modify_cost:	This routine is used by the manager to modify the cost of a link. It also updates the link table and the routing table to take the new cost into account.
print_link_table **print_routing_table**	These routines are used by the router and the manager to save the link table or the routing table into the log file.
send_routing_table	This routine is used by the router (and the manager only at the start of the algorithm’s execution) to send the local routing table to all the neighbors.
send_message	This routine is used by the manager to start the sends of a message between routers. It is also used by routers to send and receive message between them.
receive_routing_table	This routine is used only by the router to execute the Distance Vector Routing algorithm at the reception of a routing table.
table_to_string **string_to_table**	These functions are used by the router to transform a routing table in a string and vice-versa. A routing table is always sent as a string.
log_message	This routine is used by the router to do a separate call to the routine *log_message* of the manager.
snd_message **rcv_routing_table**	These routines are used by the router to do a separate call to the routines *send_message* or *receive_routing_table* of another router.

5.1.2 The manager

The manager is a separate supplier of the routers and the GUI, it is also a separate client of the routers. The task of the manager is to provide the control facilities of all the routers to the GUI. The manager has two attributes: a reference to the GUI named ***gui*** and a table named ***routers*** to save the name and the reference of each router.

The different features of the MANAGER class are described below:

init	This routine is used by the GUI to initialize the entire network specified in the configuration file.
simulate	This routine is used by the GUI to start the algorithm's execution
send	This routine is used by the GUI to start the sends of a message between routers.
modify_cost	This routine is used by the GUI to modify the cost of a link.
log_message	This routine is used by the router to display a text in the GUI.
add_link **init_routing_table** **print_link_table** **print_routing_table** **send_routing_table** **send_message** **mod_cost**	These routines are used by the manager to do a separate call to the routines *add_link*, *init_routing_table*, *print_link_table*, *print_routing_table*, *send_routing_table*, *send_message* or *modify_cost* of a router.

5.1.3 The GUI

The GUI is separated in two classes. The first class named MANAGER_GUI only creates the application and initializes a window of type MANAGER_WINDOW. The class MANAGER_WINDOW provides facilities to control the manager. This class has a lot of attributes but the most important is a reference to the ***manager***.

The different features of the class are described below:

initialize	This routine is automatically called by the MANAGER_GUI to initialize the window.
display	This routine is used by the manager to display a text in the GUI.
exit **close_window**	These routines are called if the user pushes the exit or the close_window button. The execution of the separate objects is stopped and the application is destroyed.

init_simulation **simulate** **send** **modify_cost** **open**	These routines are called if the user pushes the corresponding button.
build	This routine is used by the GUI to create the graphical interface.
init_sim **sim** **snd** **mod_cost**	These routines are used by the GUI to do a separate call to the routines *init*, *simulate*, *send*, or *modify_cost* of the manager.

6 Benchmarking

The algorithm's execution has been done on a local machine. The processor of this machine is a Pentium m with 1.5GHz and the operating system is Windows XP.

6.1 *Simple network*

The simple network shown in figure 3 is the first test case that we want to consider. The execution of the algorithm with the application has produced the same routing tables than in figure 2. The algorithm took 120 ms to converge and there were 20 updates on the routing tables.

6.2 *Linear network*

The next interesting case is a linear network as shown in figure 7. In this case, the costs of the links are negligible and could be the same for all because there is only one path. In the figure 8, we can see 5 different measures of the execution time of the algorithm compared to the number of routers in the linear network. For example, a network with 20 routers needs 1.5 second to converge and a network with 5 more routers needs twice more time. We can see that the execution time doesn't increase linearly when the number of routers increases. This result is correct compared to the complexity analysis of the algorithm done in [6]: in the worst case, the time complexity is exponential in the number of routers.

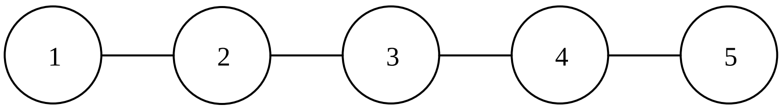


Figure 7: linear network with 5 routers

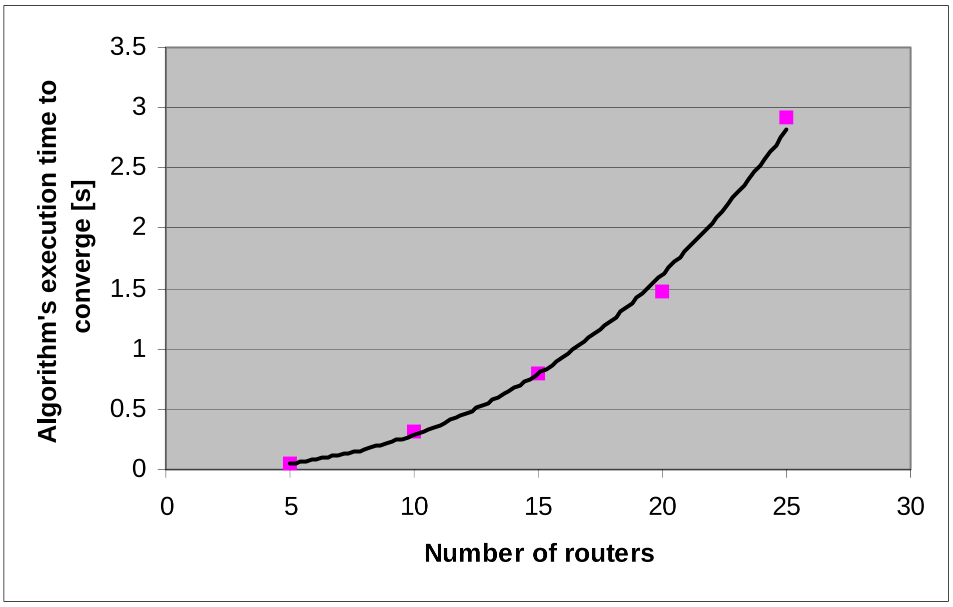


Figure 8

The figure 9 shows the average number of routing table updates for each router compared to the number of routers in the linear network. It is very interesting to see that a small network with for example only 25 routers has more or less 400 changes of routing tables during the algorithm! Again, this increase of updates is correct compared to the complexity analysis of the algorithm [6]: in the worst case, the exchanged message complexity is exponential in the number of routers and the number of exchanged messages is always greater than the number of updates because there are messages that don't modify the routing table.

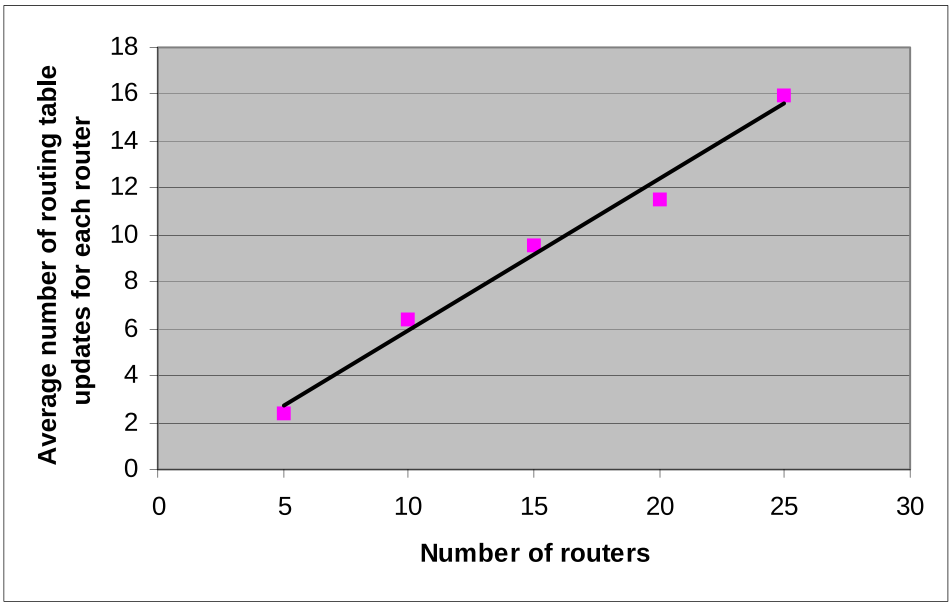


Figure 9

6.3 Binary tree network

Another interesting case is a network with a structure of a binary tree as shown in figure 10. We have executed the algorithm for 4 different networks with 3, 7, 15 and 31 routers. The algorithm has always converged and the resulting routing tables were correct.

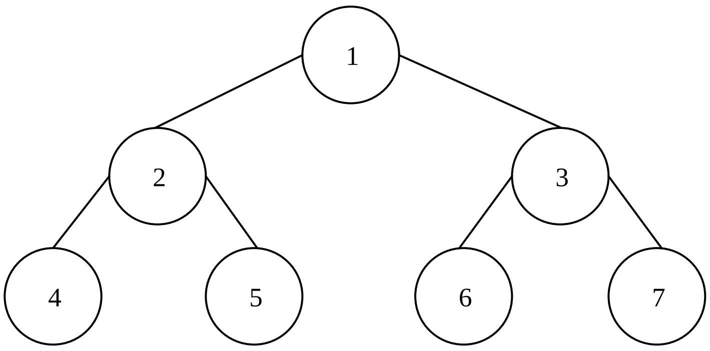


Figure 10: binary tree network with 7 routers

We can see in figure 11 that the execution time for the algorithm increases almost exponentially when we increase the numbers of routers in the network. This result is very close to the theoretical time complexity in the worst case.

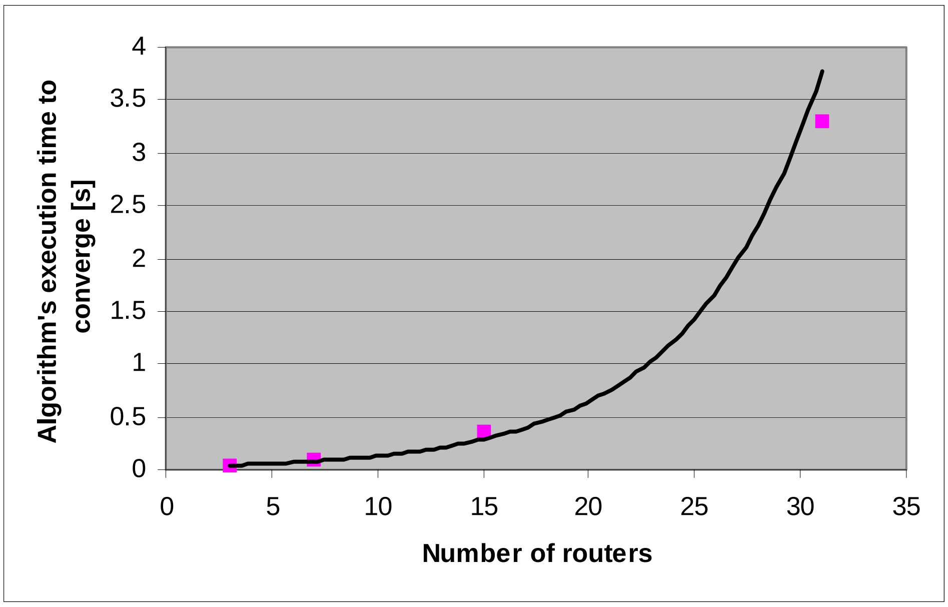


Figure 11

In figure 12, we can see again the impressive number of routing table updates. Note that the binary tree network needs less update than the linear network but the result is again almost exponential and thus correct compared to the theory.

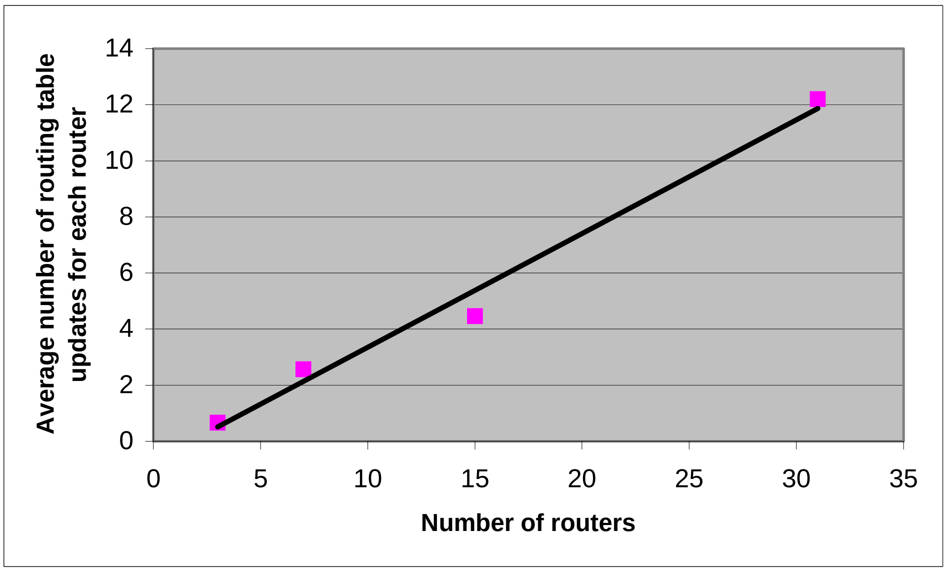


Figure 12

7 Feedback

7.1 Design

Without the SCOOPLI library, it would be more difficult to design a distributed application: relying for example on message-passing primitives with explicit *send* and *receive* (implemented using sockets, for example), the semantics of the SCOOP separate call would not be maintained. This would have for consequence that *receive* instructions would be spread in the object methods code. Using remote procedure call primitives (such as the ones in ORBs) would be better, but they would not guarantee that reasoning about the consistency of supplier objects between two calls would be sound. Only the SCOOP separate call semantics with locking of the supplier object guarantees that. In essence, this gives to the client object an (mutually) exclusive access to the routing tables of the supplier router.

Here we could concentrate only on the main task of the application development, not really thinking about internal concurrency issues. Otherwise (as in e.g. Java) we should have designed the application using concurrent processes, threads or application domains, and also synchronized them with the assistance of semaphores, mutexes, monitors or barriers. Moreover, to be closer to the original implementation solutions found in the literature, we should have designed a message passing mechanism to communicate information between the different actors of the application. SCOOP provides a unified primitive (the separate call) that can be implemented in both concurrent and distributed frameworks. Without the SCOOPLI library, we should rewrite the code and use an available distribution framework. For example, existing .NET implementations allow access to a framework named ".NET remoting" that provides mechanisms to distribute objects on different machines.

As SCOOPLI uses a separate call instead of a message-passing to communicate between objects, we should ask whether this mechanism would in some circumstances show a footprint larger than necessary. Our tentative answer is that, in an environment where processes are loosely coupled, the SCOOP call mechanism can be used instead of message passing with almost no overhead, while allowing to reason more easily about programs.

7.2 Debugging

The most frequent bugs in concurrent applications are in the implementation of concurrency mechanisms. In this project, SCOOPLI does this tedious work and so it was very easy to debug the code. The only problem that we had is a bug in the SCOOPLI library. We found a deadlock problem in the feature *execute_thread* of the class PROCESSOR_HANDLER. P. Nienaltowsky helped us to fix this bug.

7.3 Suggestions for future releases

It would be very useful to simplify the separate call in SCOOPLI because now we must always use the features *execute* and *execute_routine* with agents to do an asynchronous call. The use of the syntax of SCOOP [3] would be more elegant.

It would be also interesting to provide a basic library for separate types like STRING, ARRAY, etc. It isn't very powerful to implement these separate types by oneself because we must always inherit from the SEPARATE_SUPPLIER or the SEPARATE_CLIENT class.

8 Conclusion

The application to execute the Distance Vector Routing algorithm using SCOOP has been designed and implemented in accordance with the specifications: the application is able to read the network configuration using a file, to compute the algorithm, to send packets between routers and to modify the cost of a link. A basic GUI allows controlling the application and displaying information like the routing tables, the link tables and the route of forwarded packets. In future release, there would be useful to add an extension to display the network topology. We can also imagine an extension to dynamically add or remove routers and links.

After the implementation, the application has been tested and benchmarked with different interesting test cases to show the correctness of the program. All these test cases have been reported in the current document.

Finally we can say that this project is a very positive experience in object-oriented technology. It was very interesting to work on a research topic and we hope that this project will be useful for the future release of the SCOOPLI library.

9 References

[1] P. Nienaltowsky, V. Arslan, B. Meyer: *SCOOP: Concurrent Programming Made Easy.*

[2] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[3] P. Nienaltowsky, V. Arslan: *SCOOPLI: a library for concurrent object-oriented programming on .NET.*

[4] G. Coulouris, J.Dollimore, T. Kindberg: *Distributed Systems, Concepts and Design*, 3rd edition, Addison Wesley

[5] Andrew S. Tannenbaum: *Computernetzwerke,* 4th edition, Prentice Hall

[6] Nancy A. Lynch: *Distributed Algorithms,* Morgan Kauffmann