Robust Overlay Network with Self-Adaptive Topology: Protocol Description

Loïc Baud, Nguyen Pham, Patrick Bellot

Institut TELECOM, TELECOM ParisTech & LTCI CNRS UMR 5141
46 rue Barrault, 75634 Paris Cedex 13, France
{baud,pham,bellot}@telecom-paristech.fr

***Abstract*—We introduce a new overlay network named ROSA[1]. Overlay networks offer a way to bypass the routing constraints of the underlying network. ROSA used this overlay network property to offer a resilient routing to critical applications. Unlike other overlay networks dealing with the routing resilience issue, we oriented our research towards building a robust overlay network topology instead of a robust routing function. We tried to maintain a path between any pairs of nodes of the network. The routing resilience is obtained by forcing nodes to choose and modify dynamically their neighbors set according to the ROSA protocol. Moreover, ROSA is highly scalable.**

***Index Terms*—Overlay network, Network resilience, Self-organizing system.**

I. Introduction

Computer-based systems are nowadays present in the economy, medical processes and equipment, power and communication infrastructures. However, building systems that are guaranteed to be secure or to remain secure over time is still an unachievable goal. Hence, security of information systems becomes naturally a crucial concern for the governments and the worldwide companies.

Studies from [13] and [18] have demonstrated that Internet connectivity failures are not rare. In 1.5% to 3.3% of time, failures prevent pair of hosts from communicating.

Let us imagine a distributed tool deployed on the network of a company. This network could be spread over many countries and therefore over many autonomous systems (AS) and sub networks. The tool will need to exchange information between these sub networks. We cannot tolerate that some failures prevent these sub networks from communicating. Such a tool needs to have guarantees on the efficiency and the resilience of the routing of its datas.

Since BGP [3] can take much time, many minutes sometimes, to discover a failure and to reorganize the routes in a consistent form [10], we have to find a solution in order to assure a resilient routing to these kind of tool. The solution that has caught our attention is the Overlay Network (ON).

This paper presents ROSA, an ON especially adapted to be used as a backbone for large applications that need a very robust routing. The rest of this paper is presented as follows: the section II presents the necessary definitions; the section III presents the ROSA ON and its properties; the section IV shortly presents an application of ROSA and the section V presents a short conclusion ans a brief synthesis of this article.

[1]This work has been granted through the IST FP6 DESEREC Project (CN: 026600) of the European Union.

II. Definitions

An Overlay Network (ON) is a virtual network built on another network, usually the Internet, to provide a service to a community of users. Significant examples of overlay networks are peer-to-peer networks (P2P) Kademlia [11], Gnutella [7], Chord [15], Skype [8], Domino [17] and Virtual Private Networks (VPN) [16]. Each one of these overlay networks has its own use. Kademlia and Gnutella are usually used for file-sharing. Chord is used in CFS [6] as a backbone for a distributed file system. Skype aims to provide VoIP to its users. Domino detects intrusion on a system. VPN protects its users from eavesdropping.

A node of ROSA is any computer of the underlying network following our protocol. We call virtual link between two nodes the protocol used (TCP, UDP, SSL, etc.) as well as the set of the elements of the underlying network taking part in the communication between these two nodes.

We call failures all events that prevent two nodes linked together by a virtual link from communicating. These failures can have numerous causes such as hardware failure, software failure, attacks, power cut, etc.

III. ROSA

A lot of Overlay Networks have already dealt with the problem of resilient routing. The most famous is undoubtedly RON (Resilient Overlay Network) [2]. RON maintains a virtual link between all pairs of nodes. Each node in RON checks the availability and the capacity of the virtual links between itself and the other nodes. Each node decides then, based on its knowledge, if it should let packets flow directly to other nodes, or if some indirections via other RON nodes could be useful. But since each node in RON is linked with all the other nodes, RON cannot exceed more than one hundred of nodes.

This limit on the maximum number of nodes does not allow to use RON as a backbone for our tool. To improve RON scalability, the solution chosen by DG-RON [14] consists in splitting the network into logical zones in such a way that

each node has to maintain and exchange information only with nodes of its logical zone.

We can obviously see that the number of neighbors that a node has to manage is decisive if we want our overlay network to be scalable. Since we cannot do anything in order to increase the number of neighbors that a node can manage, it only remains to efficiently choose these neighbors. A solution proposed in [1] consists in taking into account the underlay network topology in order to construct the Overlay Network. But the algorithm proposed can only be used to construct static Overlay Network and cannot be applied to the construction of self-organizing ones.

From these observations, we decided to build our overlay network called ROSA for Robust Overlay network with Self-Adaptive topology. ROSA, in order to maintain a path between any pair of nodes, dynamically reorganizes the neighbors set of the nodes according to:

- the topology of the underlying network ;
- the maximum number of neighbors of a node ;
- the failures on the elements of the virtual links.

A. ROSA architecture

We introduce the notion of lump. The lumps allow each node to compute the robustness of the network at its virtual neighborhood.

A lump is a set of fully virtually connected nodes. In Figure 1, we can see an overlay network with three lumps.

We notice that the network in the Figure 1 includes more than 3 lumps. Indeed, some lumps include sub-lumps. However, the interest of these sub-lumps being null for us, we ignore them. We only consider the maximum lumps according to set inclusion relation.

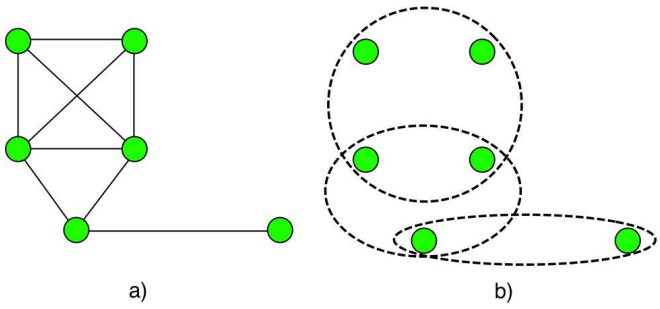

Fig. 1: a) An overlay network b) The 3 *lumps* of this network.

From this definition, we can consider an overlay network as an entanglement of lumps. RON, for example, can be seen as a single lump since each node is linked with all the other nodes of the network. We call member of a lump each node that belongs to it.

B. Density of a lump

Let l be a lump, we call density of l the quantification of its capacity to maintain a path between all its members in the presence of a virtual link failure. We assume that each node is able to identify and distinguish the elements of the underlying network composing the virtual links that connect it to its neighbors. In the case of IP, the `traceroute` utility [9] allows to do this. The density is defined as the minimal number of failures on the elements of the virtual links of the lump that are necessary to isolate a node of the lump. In such a way that if the number of failures is less than the density, we can affirm that there exists a path between any two nodes of the lump. This density will be noted $d(l)$.

One can remark that if all the virtual links are disjoints, the density of a lump is simply equal to the cardinal of the lump minus one, i.e. #l - 1. We also use the cardinal of the lump to compute the density in the case where the nodes are not able to know the composition of the virtual links. Nevertheless, this does not give good results since the topology of the underlying network is not taken in account.

In the future, we plan to use other formulas to compute the density. For instance, we could use a formula that takes into account the type of the elements that compose the virtual links. The only condition to respect is that each member of a lump must possess all the necessary information to calculate the density. The acquisition by a node of this information is performed when the node joins a lump.

Figure 2 illustrates a lump. We can see that all the virtual links share a common element. A failure of this element is sufficient to break down all the virtual links. Therefore, the density of this lump must be weak: indeed, it is equal to 1. If the virtual links connecting the members of the lump together were disjoints, then the density of this lump would be 3.

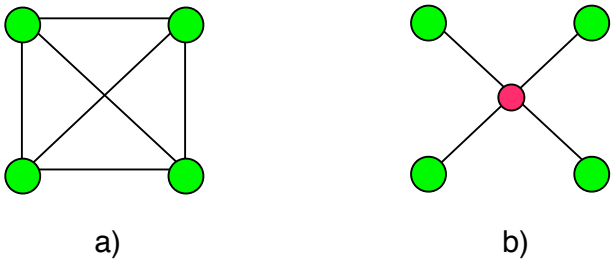

Fig. 2: a) A *lump* b) The underlying network.

We will call density of the ROSA network the density of the weakest lump. The weakest lump is the lump with the weakest density. Since each node belongs to at least one lump, we can be sure, that if the number of simultaneous failures on the virtual links is inferior to the density of ROSA, there exists a path between any two nodes of the network.

C. Principle of ROSA

As it will be explained in the following part, nodes in ROSA periodically exchange lump with poor density in such a way that each node is always in possession of a set of lumps of weak density. In order to enhance the density of the lumps, each node tries to join the weakest lump that it knows.

If a node wants to join a lump, it has to compute the theoretical density of the lump in the case of it joins it. Then, two cases are possible:

- The node can join this lump and the number of its neighbors does not exceed its maximum. Then, the node joins the lump and informs other members of the lump about it.
- Otherwise, the node has to leave some of its neighbors if it wants to be able to join the lump. This will be done only if the fact of leaving these neighbors does not imply that this node leaves a lump with a smaller density than the one it wants to join.

This principle ensures that unless if some failures occur, the density of ROSA cannot decrease.

D. Protocol of ROSA

1) Node, neighbor and lump representation: Each node in ROSA has an identifier, a list of lumps. a list of neighbors and a maximum number of neighbors. The identifier must be unique. We use a hash based of the IP address of the underlying network and some other node characteristics, in such a way that the hash obtained is unique. The list of lumps is composed of the lumps the node belongs to. The list of neighbors is composed of the nodes that are members of the lumps of the previous list. The maximum number of neighbors can vary according to each node capacity.

A neighbor is simply represented by its identifier. A lump is represented by the list of the identifiers of its member and by all the information that is needed to compute the lump density. Therefore when it is said that a node sends a lump the reader has to understand that the node sends the representation of this lump. By this way, a node receiving a lump is able to compute its density. A node can add a lump into its lumps set only if there is no lumps in the set that include the new one. When a node adds a lump into its lumps set, it removes all the lump included into the new one.

The pseudo code of a ROSA node is presented in Agorithm 1. If the node is the first node of the network we have to set the *create_rosa* to true. This will force the node to initiate the ROSA Network (line 2). Else *bootstrap_node_ip* must contains the IP address of a node in order to join ROSA, (line 3).

The state of a node determines whether or not the node is active on a ROSA network. The function `send_alive()` (line 3) aims to notify neighbors that the node is operational. It aims by the way to propagate knowledge about lumps to join. We deal more in details with this function in Section III-D4. The function `check_neighbors()` (line 4) is used to detect and handle failures. It is also used to keep the number of neighbors below the limit number. This is explained in Section III-D5. The function `reconfigure()` (line 5) is the most interesting. This function allows the node to join some lumps in order to enhance ROSA topology robustness. See Section III-D6 to learn more about it. Once these three functions are performed, the node waits a small amount of time before repeating these. During this waiting time, the node keeps listening and handling other nodes messages. This amount of time determines the reactivity to failures and the overhead generated by ROSA. The lower this amount of time is, the faster the reactivity is and the more important overhead is.

Algorithm 1: Node

```
   input
        time_interval ;                          // Time interval in seconds
        max_neighbors_number ;                   // Maximum number of neighbors
        create_rosa ;
        bootstrap_node_ip ;

   variables
        state ← OFF ;                                        // Node current state
        neighbors_set ← ∅ ;                  // Set of the neighbors of the node
        lumps_set ← ∅ ;              // Set of the lumps to which the node belongs
        known_lumps_set ← ∅ ;                        // Set of received lumps

   node
        if create_rosa then
1           init_rosa() ;
        else
2           join_rosa(bootstrap_node_ip);

        while state = ON do
3           send_alive() ;
4           check_neighbors() ;
5           reconfigure() ;
6           sleep(time_interval) ;

        handles receiveid request ...
```

2) Initiating ROSA: The first node has to initiate the ROSA network. It simply has to add a new lump to its lumps set. This lump has only the node for member. The pseudo code corresponding is described in Agorithm 2.

Algorithm 2: Node.init_rosa

```
   node.init_rosa
1      lump ← new Lump ;
2      lump.members ← lump.members ∪ {node} ;
3      node.lumps_set ← lump ;
4      node.state ← ON ;
5      return ;
```

3) Joining ROSA: In order to join the ROSA network a node must know the IP address of a node already connected to the network. We call this node the bootstrap node. Once this IP address is known, the node willing to join has to send a message `Hello` to the bootstrap node, see Agorithm 3.

Algorithm 3: Node.join_rosa

```
   input
        bootstrap_node_ip ;                      // IP address of a node of ROSA

   node.join_rosa
1      send(Hello) to bootstrap_node_ip ;
2      return ;
```

A node receiving such a message replies by a message *FirstLump* containing the lump to which it belongs that possesses the lower density. Therefore, the node that wants to join receives a message *FirstLump* from the bootstrap node. It joins the lump contained in the message. How the messages *Hello* and *FirstLump* are handled is described in Agorithm 4. We will explain how a node join a lump in the Section III-D7

Algorithm 4: Hello and FirstLump messages handling

```
   Messages handling
       ...
       upon receive(Hello(lump)) from node n do
1          send(FirstLump(get_weakest_lump(node.lumps_set))) to n ;
2          return ;

       upon receive(FirstLump(lump)) from node n do
3          if node.lumps_set ≠ ∅ then return ;
4          lump.members ← lump.members ∪ node ;
           foreach m ∈ lump.members do
               if m ≠ node then
5                  node.neighbors_set ← node.neighbors_set ∪ {m} ;
6                  send(JoinLump(lump)) to m ;
7          node.lumps_set ← node.lumps_set ∪ {lump} ;
8          node.state ← ON ;
9          return ;

       ...
```

4) Propagating lumps information: The propagation of lumps information is performed by the function `send_alive()`. It consists for a node to send a message *Alive* to each of its neighbors. This message contains a lump having the node as member. This lump will be used in the reconfiguration process to enhance the ROSA topology robustness. The pseudo code of this function can be found in Algortihm 5.

Algorithm 5: Node.send_alive

```
   variables
       lump_to_send ;

   node.send_alive
       foreach n ∈ node.neighbors_set do
1          lump_to_send ← find_appropriate_lump(n) ;
2          send(Alive(lump_to_send)) to n ;
3      return ;
```

The lump that has to be sent to a neighbor is selected as follows: that is the lump that have the lowest density among those that do not have the neighbor for member. This selection is performed by the function `find_appropriate_lump(neighbor)` (line 1). A node that receives a message *Alive* from one of its neighbors, considers that this neighbor is operational and adds the lump contained in the message to its set of known lumps. This is described in Algorithm 6.

Algorithm 6: Alive messages handling

```
   Messages handling
       ...
       upon receive(Alive(lump)) from node n do
1          if n ∉ node.neighbors_set then return ;
2          n.flag_alive ← true ;
3          node.known_lumps_set ← node.known_lumps_set ∪ {lump} ;
4          return ;

       ...
```

5) Failures detection: Nodes do not directly detect underlay network failures but detect when virtual links are broken. A node considers that the virtual link between itself and one of its neighbor is broken when it has not received at least one message *Alive* from this neighbor between two failures verifications.

When a node detects that a virtual link is broken, it sends a message *SplitLump* to some of its neighbors. The concerned neighbours are the members of lumps that have the node and the failing neighbor as members too. Then the node removes the failing node from its neighbors set. This message *SplitLump* contains the identifier of the failing node. The failures verfication is performed by the `check_neighbors()` function. The pseudo code of this function can be found in Algorithm 7.

This function also keeps the number of neighbors below the limit (lines 3-4). The function `find_neighbor()` returns the neighbor such that if node separates from it then it will cause the minimum of loss according to the ROSA topology robustness quantification. We will not describe in details this function in this paper, its pseudo code is in Algortihm 8.

Algorithm 7: Node.check_neighbors

```
   node.check_neighbors
       foreach n ∈ node.neighbors_set do
           if n.flag_alive = false then
               foreach l ∈ node.lumps_set do
                   if n ∈ l then
                       foreach m ∈ l.members do
                           if m ≠ node & m ≠ n then
1                              send(SplitLump(lumps,n)) to m ;
                           l.members ← l.members \ {n} ;
2                  node.neighbors_set ← node.neighbors_set \ {n} ;
           else
3              n.flag_alive ← false ;

4      while # node.neighbors_set > node.max_neighbors_number do
5          neighbors_set ← neighbors_set \ {node.find_neighbor()};
       return ;
```

How a node hanldes *SplitLump* messages is described in Algorithm 9. A node receiving a message *SplitLump* from a neighbor removes the lump contained in the message and adds two lumps to its lumps set. The members of these two lumps are the members of the removed lump without the neighbor that sends the message for the first one (line 2) and without the node contained in the message for the second one (line 3).

An example of failure detection can be found in Figure 3. We can see in the top left a ROSA network composed of two lumps A and B. In the top right we can see that the node in red does not send a message *Alive* to one of its neighbors. That means that the virtual link between these two nodes is broken. In the bottom of the Figure, we can see how the ROSA network has self-reconfigured. The two initial lumps were splitted in two parts each.

6) Reconfiguring: Once the messages *Alive* are received, the node knows a set of lumps with low densities. By definition the node, is not a member of these lumps. The reconfiguration is performed by the function `reconfigure()`. The pseudo code of this function is described in Algorithm 10. The

Algorithm 8: Node.find_neighbor

```
output
    selected_neighbor ;   // the neighbor which the separation will cause the
    minimum loss
node.find_neighbor
    foreach l ∈ node.lumps_set do
        if l.density > 2 then
            return(node.find_neighbor_by_density());
    return(node.find_neighbor_by_size());

variables
    max_min_density ;
    min_density ;
    tmp_lump ;
node.find_neighbor_by_density
    max_min_density ← 0 ;
    selected_neighbor ← null ;
    foreach n ∈ node.neighbors_set do
        min_density ← ∞ ;
        foreach l ∈ node.lumps_set do
            if n ∈ l then
                tmp_lump.members ← l.members \ {n} ;
                if tmp_lump.density < min_density then
                    min_density ← tmp_lump.density ;

        if min_density > max_min_density then
            max_min_density ← min_density ;
            selected_neighbor ← n ;
    return(selected_neighbor) ;

variables
    max_min_size ;
    min_size ;
node.find_neighbor_by_size
    max_min_size ← 0 ;
    selected_neighbor ← null ;
    foreach n ∈ node.neighbors_set do
        min_size ← ∞ ;
        foreach l ∈ node.lumps_set do
            if l.size < min_size then
                min_size ← lump.size ;

        if min_size > max_min_size then
            max_min_size ← min_size ;
            selected_neighbor ← n ;
    return(selected_neighbor) ;
```

Algorithm 9: SplitLump messages handling

```
Messages handling
    ...
    upon receive(SplitLump(lump,dead_node)) from node n do
        if n ∉ node.neighbors_set then return ;
        if lump ∉ node.lumps_set then return ;
1       node.lumps_set ← node.lumps_set \ {lump} ;
2       lump_1.members ← lump.members \ {n} ;
3       lump_2.members ← lump.members \ {dead_node} ;
4       node.lumps_set ← node.lumps_set ∪ {lump_1} ∪ {lump_2} ;
        return ;

    ...
```

reconfiguration consists for the node in joining the lump of this set that has the lowest density (line 1). The node will join this lump unless if joining this lump increases its density (line 7).

7) Joining a lump: In the previous sections, we said that the nodes join lumps without explaining how it was performed. In this section we will make this point clearer. When a node wants to join a lump, it first adds itself to the list of members. It adds then all the information that will be needed to compute

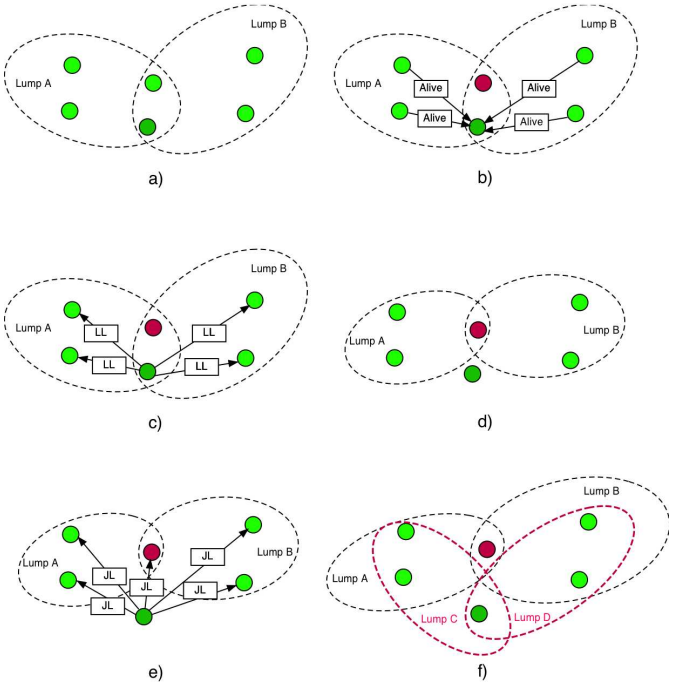

Fig. 3: Failures detection.

Algorithm 10: Node.reconfigure

```
variables
    lump_to_join ;          // lump_to_join that the node will try to join
    old_density ;     // density of lump_to_join before that the node join it
    new_density ;           // density of lump_to_join if the node join it

node.reconfigure
1   lump_to_join ← get_weakest_lump(node.known_lumps_set) ;
2   node.known_lumps_set ← ∅ ;
3   old_density ← lump_to_join.density ;
4   lump_to_join.members ← lump_to_join.members ∪ {node} ;
5   new_density ← lump_to_join.density ;
6   if old_density ≥ new_density then return ;
    foreach m ∈ lump_to_join.members do
7       node.neighbors_set ← node.neighbors_set ∪ m;
8       send(JoinLump(lump_to_join)) to m ;
9   node.lumps_set ← node.lumps_set ∪ {lump_to_join} ;
10  return ;
```

the density of this lump. It sends then a message *JoinLump* to the members of this lump and adds these members to its neighbors set. This message contains the lump that the node wants to join. To finish, the node adds the lump to its lump set. The pseudo code corresponding to this process is shown in Algorithm 10 lines 3 to 9.

A node receiving a message *JoinLump*, as it can be seen in Algorithm 11, checks that it is a member of this lump. If it is, the node adds the lump contained in the message to its lumps set and adds the node that has sent the message to its neighbors set.

8) Leaving ROSA: To complete the protocol description, we have to describe how a node leaves the network. It can simply stop sending messages *Alive* to its neighbors. The node

Algorithm 11: JoinLump messages handling

```
Messages handling
    ...
    upon receive(JoinLump(lump)) from node n do
        if node ∉ lump.members then return ;
1       node.lumps_set ← node.lumps_set ∪ {lump} ;
2       node.neighbors_set ← node.neighbors_set ∪ {n} ;
3       return ;
    ...
```

will be naturally removed from all the nodes neighbors sets. The second solution consists for the node that leave ROSA in sending a message *Leave* to its neighbors. This solution generates less overhead that the first one.

A message *Leave* does not contains anything. When a node receives such a message from one of its neighbor, it removes this neighbor form its neighbors set and remove this neighbor form all the lumps in the lumps set. See Algorithm 12.

Algorithm 12: Leave messages handling

```
Messages handling
    ...
    upon receive(Leave()) from node n do
        if n ∉ node.neighbors_set then return ;
        foreach l ∈ node.lumps_set do
            if n ∈ l.members then
1               l.members ← l.members \{n} ;
2       node.neighbor_set ← node.neighbor_set \{n} ;
3       return ;
    ...
```

E. Routing data in ROSA

ROSA proposes for now, two routing modes: routing by flood and routing by path. These two routing modes possess some advantages and some drawbacks. These advantages and drawbacks define in which cases we have to use the first routing mode and in which cases we have to use the second one.

1) Routing by flood: A node willing to send a message to a target using the flooding mode, sends this message to all its neighbors. The target could either be a single node than a set of nodes. This message contains, in addition to the payload, a unique identifier that allows to a node to distinguish this message from another, and the target identifier. Each node that receives a message by flooding first checks that it didn't already receive it. If the message was already received, the node simply discards it. Else, the node checks if it is a target of this message. If it is, the node handles the payload of the message. Then the node sends, using the flooding mode, the message to all its neighbors. The routing by flood offers numerous advantages. Especially, the one to send a message to a set of nodes without knowing anything about the localization of these on ROSA. An other advantage is that this routing mode ensures that all the targets will receive the message. On the other hand, there is a major drawback; this routing mode produces much traffic on the network.

2) Routing by path: A node willing to send a message to an other node using the routing by path has to already know a path in ROSA between itself and the target node. A path is a sequence of node identifiers; these nodes have to link the source node to the target node. We will see later how to obtain this kind of paths. The message to send has to be composed by the path and the payload. Each node receiving such a message seeks its identifier in the path contained by the message, and deduces from this the next node to which it has to send the message. The message, thus relayed along the path, reaches its destination. Contrary to the routing by flood, the routing by path produces less traffic. Nevertheless, it has the drawback to impose to the node that wants to use it to have knowledge about the ROSA topology. Moreover, ROSA is dynamic and so the virtual links are able to be broken and it is possible that old paths are not valids anymore.

3) Path discovery: The simplest way, for a node to discover a path between itself and an other node, is to use send, using the routing by flood, a message *PathDiscovery* to this targeted node. This message contains, as all the flooded messages, an unique identifier and the targeted node identifier. It also contains the list of the identifiers node by which the message was relayed. Each node receiving for the first time this message has to add its identifier in the end of this list. Therefore when the message reaches the targeted node, the list contains a path linking the source node to the targeted node. Then, the targeted node has just to reply to the source node by a message containing the path. The reply can be flooded or simply routed using the path recently obtained.

4) Uses of the different modes of routing: It does not exist a routing mode absolutely better than an other. We have to determine the routing mode to use according to what we want to do. There are two factors to take in account, the amount of traffic produced and the assurance that the message will effectively be received. We recommend using the routing by flood to send urgent and important messages. We also recommend using this routing mode when we want to send the same message to a large group of nodes. The routing by path has generally to be used only when two nodes want to exchange a huge amount of messages in a small amount of time.

IV. A TOOL FOR MANAGING A NETWORK

ROSA overlay network has been tested in real condition within the context of the DESEREC Project of the European Union. The goal of this project is to provide a framework that will increase the dependability of networked informations systems.

For this project Telecom Paristech has developed a distributed tool that aims to manage a network.

It can be splitted in three parts:

- a set of captors deployed over the network ;
- a security policy ;
- the resilient routing (ROSA overlay network).

The captors are used to detect: attacks, failures, services bugs, etc. In order to do it, Security Assurance (SA) are

continually computed for different components of the networks (workstations, servers, gateways, routers, sub networks, etc). The SA is the objective confidence that an entity meets its security requirements. More precisions about SA are given in [4] and [5]. When the SA value of a network component decreases going down, the security policy has to be applied. In the current implementation of our tool, the security policy has only 2 rules. The first rule is applied when the SA value of a router or a gateway is decreasing below a threshold level. This rule consist in finding an alternative gateway to replace the failing one. If such gateway is found, the routing tables of the network components are modified to use the new gateway. Else the tool waits that the failing gateway be repaired. The second rule is applied when the SA value of a sub network is decreasing. This can occurs if the sub network go through a DDoS or a worm attack. In this case the sub network is isolated in order to prevent attacks propagation.

ROSA is in charge of routing the data collected by te captors to the tool, the internal messages of the tool, and the network reconfiguration messages of the tool to the network components.

This tool was deployed on the INFRES (Computer Science and Networking Department, TELECOM ParisTech). This network consists in: three Foundry routers, five sub networks, different Sun stations, and four servers whose two acting as perimeter servers between sub networks. Each station and server runs the ROSA protocol and so is a ROSA node. This screenshot of this application is shown in Figure 4. The Sun stations are the green squares, the routers are oranges and the purple ones are the servers.

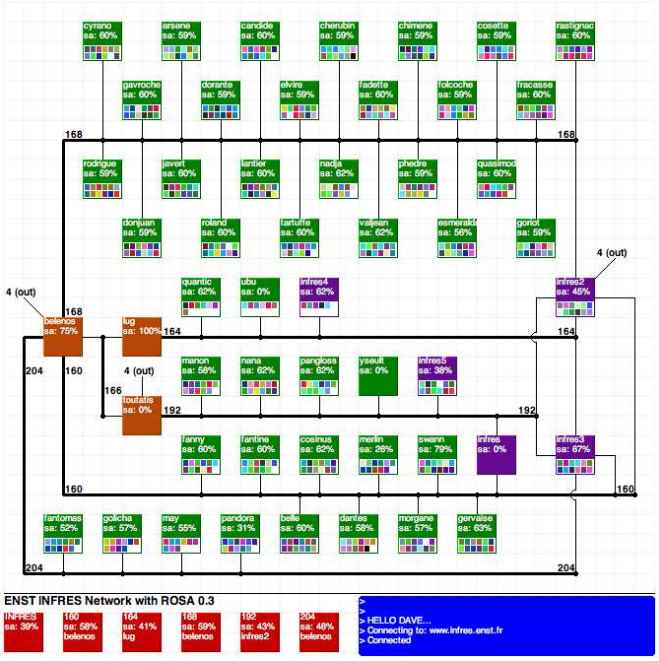

Fig. 4: The INFRES network architecture (security cockpit).

A more complete description of this tool can be found in [12]. We induced failures on the elements of the network and verify that with a small number of neighbors by nodes (only ten) ROSA was able to route the messages of the tool.

A demonstration movie of the experimentation can be found on: *http://www.telecom-paristech.fr/~bellot/FT.html*.

V. Conclusion

In this paper, we presented a new self-organizing and scalable Overlay Network ROSA. ROSA is designed to provide resilient routing. We validated this properties by testing ROSA upon the INFRES network. We only focused on the description of the protocol of ROSA.

The next step of our work will consist in studying in details all its properties. We will test ROSA on a very large network and with many nodes, in order to improve its scalability. We will measure the overhead generated by the reconfiguration and by the failures detection processes and compare it to those generated by other overlay networks. We will also validate our approach by model checking, this will be done by the Budapest University of Technology and Economics (BUTE). At the present time, our works aim to give ROSA a resilient distributed storage function.

References

[1] N. Akihiro, P. Larry, and B. Andy. Scalable routing overlay networks. *SIGOPS Oper. Syst. Rev.*, 40:49–61, 2006.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[3] BGP. A border gateway protocol 4 (BGP-4): RFC 1771.

[4] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[5] CC. Common criteria for information technology security evaluation. part 1: Introduction and general model, part 2: Security functional requirements, part 3: Security assurance requirements, 2005.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[7] Gnutella Community. Gnutella protocol specification v0.4, 2001.

[8] S. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer voip system, 2006.

[9] V. Jacobson. traceroute : ftp://ftp.ee.lbl.gov/traceroute.tar.gz, 1989.

[10] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *SIGCOMM*, pages 175–187, 2000.

[11] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge, USA.*, 2002.

[12] Nguyen Pham, Loic Baud, Bellot Patrick, Michel Riguidel. Towards a security cockpit. In *The 2nd International Conference on Information Security and Assurance*, pages 374–379, 2008.

[13] V. E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, Lawrence Berkeley National Laboratory, April 1997.

[14] S. Qazi and T. Moors. scalable resilient overlay networks using destination-guided detouring. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2007.

[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[16] VPN. A framework for IP based virtual private networks: RFC 2764.

[17] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system, 2004.

[18] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. *ACIRI Technical Report*, 2000.