RADAR: Ring-Based Adaptive Discovery of Active Neighbour Routers

Sylvain Martin and Guy Leduc

Research Unit in Networking, Université de Liège, Institut Montefiore B28, 4000 Liège 1, Belgium
{martin, leduc}@run.montefiore.ulg.ac.be
http://www.run.montefiore.ulg.ac.be/

Abstract The RADAR protocol and its underlying neighbourhood discovery framework extend the ANTS toolkit by giving active nodes the ability to discover dynamically other active nodes close to them without relying on any configuration file. Such an automatic discovery is the key to administration of large or sparse active networks and the first step towards an efficient active routing.
Active nodes will use their local IP routing table to run an extended ring search in their domain. An Additive Increase Multiplicative Decrease control allows RADAR to discover several neighbours per physical interface without searching too far away or fixing a maximum distance a priori. The protocol is complemented by a traffic-driven discovery that can grab capsules coming from unknown nodes (mainly outside the local domain) and trigger targetted probing of those addresses.

1 Introduction

1.1 Purpose of This Work

Active networks often require the network engineer to develop his own active routing that would better fit the application needs than what a default routing protocol would have done. One can, for instance, use active nodes *soft state* so that a multicast flow of capsules reaches all of its subscribers by simply following indications that have been stored by previous *subscribe capsules* (that follow the 'upstream').

However, to have such schemes work, we need an initial way to find routes to nodes. Clearly, in the previous example, the *subscribe capsules* cannot rely on some soft state to reach the stream source. We have different ways to deal with that kind of capsules :

1. if we know the stream source's IP address, we can rely on IP routing to reach the source.
2. if we know the stream source's active address, we could have an *active routing protocol* that would relay the capsules to the *next active hop* towards the stream source.

J. Sterbenz et al. (Eds.): IWAN 2002, LNCS 2546, pp. 62–73, 2002.

3. otherwise, we could forward the subscribe capsule to all the possible active next hop repeatedly until we find the source (that's awfully unscalable, though).

The first option is by far the simplest if its condition is met. However, it would lead to create a direct tunnel between the source and the subscriber, which prevents intermediate nodes from doing any processing on the capsules.

In both other options, we need to know a list of *next active hops* of the active node, either to exchange active-routing information, or to select one and forward it directly the data. Those active nodes will be referred to as *active neighbours* or simply *neighbours* in this paper. Most execution environments assume either that the whole network is made of active routers or that the list of active neighbours is provided through some configuration file. This limits the use of these environments to relatively small or very static testbed networks.

So the purpose here is to design a *neighbours discovery protocol* for the ANTS[1] framework that will let every active node *dynamically* discover its active neighbours to form an *overlay network* on top of the IP topology and possibly inform an active routing protocol of neighbour arrival or departure. Moreover, we want it to be fully plug-and-play and require no centralized component.

1.2 Defining Neighbourhood

Because active nodes run on top of IP, every active node can virtually reach any other active node. However, we want to reduce this "full mesh" to an overlay topology that better matches the real (physical) topology. Our protocol will perform that transformation by selecting a list of *neighbours* among all the reachable active nodes.

Active Neighbours are the equivalent of the next hops in IP routing and have to meet the following conditions:

- They must be active nodes,
- They must be directly reachable by the current node, which means if the current active node sends an IP packet to a neighbour, no other active router will receive the packet before it has reached the neighbour,
- They must be 'close enough' to the current node.

The latter condition enforces that a restricted amount of resource will be consumed to reach a neighbour node. For instance, we could fix a maximum TTL needed to reach a neighbour or define a time interval for the neighbour to respond. Note that the neighbourhood relationship is not necessarily symmetric, especially when routes aren't (i.e. if the route leading from A to B is not the route leading from B to A).

1.3 Structure of This Paper

After the state of the art summary in section 2, section 3 introduces the reader to the basic mechanisms that are used in our protocol.

The extended-ring-based discovery protocol - *RADAR* - is introduced and described in section 4. It is built on top of the framework of section 3, and adds an efficient capsules generation policy.

The architectural changes that have been made to the ANTS toolkit to make our protocol run on it are described in section 5.

2 State of the Art

Neighbourhood discovery techniques in overlay networks significantly differ from those in classical or ad-hoc networks by the fact that we cannot reach all the potential neighbours by just broadcasting a message through an interface. Although it re-uses the principle of the *extending ring search* [14], *RADAR* doesn't require any node to support multicast and works even without broadcasting facility.

Existing active network Execution Environments rarely address the problem of active neighbours discovery. In ANTS [1] and PLAN [4], routing tables and neighbourhood tables are read out of static configuration files. Using the DANTE [5] protocol from ABone [6] specifications, a single domain can join a statically configured backbone router and start exchanging routing updates, but these features don't help for auto-configuration of a dynamic overlay topology.

Other works such as the PROTEAN [7,8] project and its SPINE network infrastructure or the ALAN [9] project are based on a hierarchical database where all service providers will register. The hierarchy is usually based on domain names hierarchy and allows identification of which service is provided by which node. However, such databases don't provide a *neighbourhood* relationship and have virtually no information about components physical proximity [1]. So we can use them when a client has to find out which server it can connect to, but they aren't of much help when building dynamic active routing tables.

In the TAO dynamic overlay management algorithm [12], the neighbourhood discovery is based on DNS queries for a well-known name that returns a list of *cluster heads*, and each node uses ping statistics to join one of these clusters. Every node of such a cluster then becomes the neighbour of every other node in its cluster and an elected "cluster head" runs a routing protocol with peer clusters to build neighbourhood between clusters. Unlike what happens in our protocol, the TAO algorithm works at a coarse level: two nodes of a cluster can be neighbours even if they can't directly reach each other. Moreover, two nodes that are physically close to each other could use an excessively long path to communicate if they join distinct clusters.

3 Neighbourhood Discovery Framework

3.1 The AYA Capsules

The whole discovery process is based on very simple AYA (stands for Are You Active?) capsules - a sort of active *ICMP Echo* packet. Those capsules are sent

[1] unless assuming that a domain name is made of close devices, which is certainly false for domains as *.com*

to the node or network we want to probe and will be intercepted by the first active node they cross (see listing 1.1 for a pseudocode for `AYA.evaluate()`).

Only active nodes will recognize AYA capsules and reply to them. If such a capsule is received by a non-active IP node, this one will have no upper protocol that matches the ANEP protocol type written in the IP header and will just drop it (or possibly reply with an ICMP error message). Non-active intermediate nodes will just forward the capsule as they would forward any other IP packet.

Once the AYA capsule is received by an active node, this node will fill in the "neighbour" field with its own active address. Then the capsule stops searching its target and goes back home. The neighbour discovery application running on that home node will then learn that it has a (new) neighbour, but also that this neighbour is on the road to reach the target address.

Listing 1.1. evaluate() method for AyaCapsule

```
vars :
 neighbour = NOT_FOUND;
 target; /* probed address */
evaluate on node N:
  if node address = capsule source {
    if neighbour != NOT_FOUND
      "deliver to RADAR instance";
    else
      "route to target address";
  } else {
    if neighbour = NOT_FOUND
      neighbour= node address;
    "route back to capsule source";
  }
```

Listing 1.2. Pseudocode for RADAR probing policy

```
on every interface:
   repeat {
      testing = targets.nextRing();
      foreach (1..3) {
         testing.sendCapsules();
         wait(DELAY);
      }
      threshold+=testing.size;
   } until threshold.reached
        || testing.isEmpty;
   on "capsule c delivered" {
    "remove c.target from testing";
      threshold*=ALPHA;
   }
```

3.2 Neighbour Discovery as an Active Application

Rather than implementing the neighbour discovery protocol directly in the core of the ANTS execution environment, we decided to develop it as an *active application* that runs on top of that environment and uses it to send its capsules.

However, the neighbour-discovery application must have a privileged status in the ANTS execution environment to be able to do its job properly. For instance, it must be able to provide the enumeration of Neighbour nodes instead of the primordial node[2], and have the opportunity to inform the ANTS's routing table manager of neighbours arrivals and departures.

Special care should be taken while updating the ANTS environment to support extensions required by the neighbourhood discovery, so that appropriate security checks are performed and only "trusted" applications can modify the neighbourhood table.

[2] In ANTS, the PrimordialNode is the main component of the Execution Environment that deals with most communication with the NodeOS, including retrieving routing and neighbourhood tables

3.3 Capsules Grabbing

A protocol like the neighbour discovery protocol[3] can't work using only the regular capsule-over-udp scheme. In that scheme, a capsule is routed along a list of *udp tunnels* linking active nodes, and only the tunnel endpoints can do some processing on the capsule. This can be useful when trying to establish a service that only requires some of the network nodes (even if the whole network is made of active nodes) like transcoders or repeaters, but it completely ruins our plans when we're dealing with neighbour discovery.

Keeping this UDP service would mean that we'd have to send AYA capsules to every router on the path to a destination D for discovering the neighbour which leads to D. Indeed, sending an AYA capsule with D as IP destination would skip all intermediate active nodes and make D appear as a neighbour if it is active, which is far from what we expected.

To have things working, we need active nodes to catch the AYA capsule and evaluate them *even if they're not the IP destination of this capsule*. This is what we call *capsules grabbing*. With a NodeOS-based system [11], this means that some ANTS entity should create a new InputChannel that will inspect the IP packet header and accept them if they have the right *protocol type*, regardless of the IP destination they hold. Once such packets are received on that channel, they will be processed by ANTS as if they were coming from another channel: retrieval of the capsule class based on its Method IDentifier and then capsule evaluation. Moving to this scheme might require an update of the active environment, but it doesn't require a modification of the NodeOS itself.

As we want to keep backward compatibility with the UDP tunnel model, we don't want to grab any kind of capsule, but only those that 'require' or 'request' it. In this work, we decided to use *grabbable* capsules every time a node is requested to send a capsule to a destination for which it has no neighbour. This gives us a reliable "escape route" (falling back to IP route table) and prevents the nodes from dropping capsules because of incomplete routing or neighbourhood tables.

The capsules grabbing is mandatory for our solution, but it isn't sufficient to solve the neighbourhood problem:

1. The active nodes that will perform operations along the route might not be right *on* the default IP route and therefore would not grab the capsules they should process.
2. To be grabbable, the capsule's address must contain (or map to) a unique IP address. For capsules that don't know their destination a priori this will not be possible.
3. Only capsules whose target address can be mapped to a unique IP address can be made grabbable. Having all capsules grabbable would require that active addresses (used for ANTS active routing) *are* IP addresses.

[3] In the rest of this paper, we will give the generic name of *Neighbour Discovery Protocol*(NDP) to any software component that extends ANTS to support a dynamically-built neighbourhood table. Our RADAR protocol is one possible instance of the abstract NDP protocol.

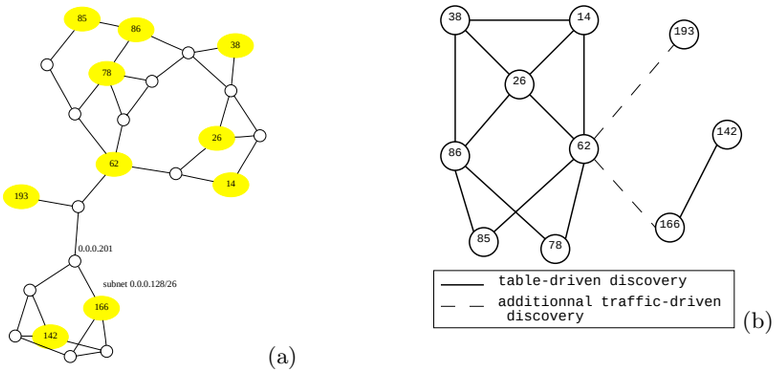

Fig. 1. *(a) topology used to test discovery protocol and (b) resulting neighbourhood mesh when table-driven discovery is used alone*

3.4 When to Send AYA Capsules ?

There are two possible approaches for the Neighbours Discovery Protocol: the *table–driven discovery* and the *traffic-driven discovery*. Better results are obtained when both discovery techniques are used together.

The table-driven approach is based on inspection of the underlying network-layer (IP) routing table, while traffic-driven discovery will try and learn from any protocol's capsules that the node receives or grabs.

Table-Driven Discovery

Every time a new entry is added in the routing table, the neighbourhood daemon is notified and will try to determine whether the *next hop* and the *destination* of this entry are active or not. When the Neighbour Application initializes itself, it will ask the *NodeOS* for a copy of the IP routing table and simply consider all the route table entries as 'new' and will process them for neighbour discovery.

In order to avoid the risk of high AYA traffic when a node boots up, route table entries are grouped by output interface and only one AYA capsule can be sent through an interface during a probing interval.

Figure 1a presents one of the topologies used for discovery testing, with active nodes highlighted. This network was in fact split into two independent domains (running shortest path algorithms) statically linked by a "backbone" node. No routing information about the domains internal routes crossed the backbone, as usual in hierarchical routing. Host 193 does not participate in any IP routing protocol and simply sends all its datagrams to its default router (the 'backbone').

As we can see on figure 1b, if the time to live (TTL) for the AYA capsules is correctly set, the discovery works quite fine: nodes within an area have found each other and the topology did not degenerate into a full-mesh. However, no

neighbourhood exists between nodes of different domains. An active node located on the backbone would probably discover the "upper" network because its gateway is an active router, but nothing similar could be guaranteed for the "lower" network. As such an active backbone router wouldn't know the details of the "lower" domain, it could only send an AYA capsule to the domain's address[4] (0.0.0.128/26) or to the domain access point (0.0.0.201). The problem is that, for both of them, there is no active router on the way while there are some active nodes in the domain. This clearly shows that something is still missing in our protocol: the *traffic-driven* discovery.

Traffic-Driven Discovery

First, we can observe that this lack of active neighbourhood doesn't completely break the reachability of the whole network. Indeed, for each destination that has no active neighbour discovered, the active router will simply forward the capsule to the final destination using a *grabbable* IP packet. Moreover, the default route is forced to have no neighbour. Thus, for capsules that rely on the active routing table to reach their destination, it's still possible to reach all active nodes, even if some of them appear to be unreachable in the active topology.

The extension of our protocol will consist of using such capsules to dynamically *learn* new neighbours. Every time a capsule is received by an active node, the node's neighbour discoverer will lookup the *previous node* address stored in the capsule in its internal tables and will try to determine to which of the following categories it belongs:

- The previous node is an already known neighbour,
- The previous node has already been tested and is not a neighbour (e.g. too far or routes from and to that node are distinct),
- The previous node hasn't been tested yet.

In the latter case, we will enqueue the previous node's address in the incoming interface's list of potential neighbours and send it AYA capsules when its turn has come.

3.5 Protocol Behaviour Summary

The Neighbour Discovery Protocol is designed for quick configuration of ANTS nodes. It probes the network with AYA capsules to discover which nodes are active using its IP routing table or other capsules to guess potential neighbours.

- Unlike most existing techniques, it doesn't require configuration files on each node and is completely "plug-and-play", but it needs a running "daemon" on every active node to send AYA capsules periodically.

[4] One of the characteristics of our discovery protocol is that it may send packets towards *network* addresses (found in the routing table) in order to reach the first active neighbour towards a given destination.

- Active nodes that don't have that daemon can still be discovered as neighbours, but can't discover other nodes.
- It can deal with neighbours crossing domain bounds if some data traffic goes from one domain to another
- Capsules with an IP destination can be routed even before discovery is completed.

In order to implement our neighbourhood discovery framework, the NodeOS must be able to provide a copy of the IP table[5]. This table will have to include at least *destination*, *next hop* and *cost* fields. Moreover, we will need a way for the NodeOS to notify changes in the routing table to the NDP framework. Finally, the addressing scheme for active nodes must include IP addresses, but is not limited to IP addressing.

4 R.A.D.A.R.: A Ring Search Using AYA Capsules

The generic neighbourhood discovery framework presented in section 3 must be completed with a *probing policy* that will choose in which order AYA capsules for table-driven discovery will be sent and which entries of the initial routing table will/won't be used for discovery.

4.1 Using Routing Table Information

In each active node, an independent instance of the probing policy is running for each network interface. At node initialization, the addresses of the routing table are distributed to each policy instance based on the interface used to reach each address.

The amount of capsules needed to discover a given topology is a crucial point of the protocol: sending useless[6] AYA capsules will lead to bandwidth and CPU waste when the network becomes larger (and thus when there are more potential targets). By ordering targets in an extending ring search rather than keeping the routing table order, *RADAR* can save some unnecessary probes.

In order to achieve a good scalability, *RADAR* is restricted to a n-hops neighbourhood for its table-driven discovery. Any entry which has a *cost* beyond the `MAX_HOPS` parameter will be automatically discarded. This prevents the protocol from using excessive resources on large networks with few active nodes. However, in sparse overlays, scanning the whole n-hops neighbourhood is already too costly, so we'll try to keep the `MAX_HOPS` only as a last resort limit.

4.2 Semi-persistent Searching

One simple way to reduce the discovery overhead is to stop searching if the physical neighbour is active. *RADAR* tries to extend this rule to the whole

[5] regardless of the protocol that built that table

[6] either AYA request duplicates asking for information we already have, or capsules trying to reach a target that is obviously too far for their limited resources.

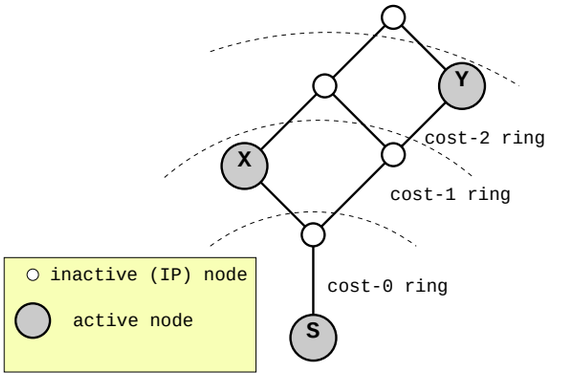

Fig. 2. *Example of impact of neighbourhood on routing performance*

discovery. Rather than stopping at the first neighbour it encounters, it has a half-persistent behaviour that makes it look a bit further and try to find more neighbours than just one per interface.

Figure 2 shows the point of doing so: if we stop our neighbourhood search when X is discovered, then routing capsules from S to Y will require 5 hops: 2 to reach X (nearest neighbour) and then 3 other hops to go from X to Y. If *RADAR*'s threshold is properly set up, we can make the discovery daemon continue and find both X and Y. Capsules will now be routable directly from S to Y without extra cost.

Restricting the search to the *cost ring* of the first discovered neighbour will often lead to sub-optimal routes, whereas a bit more probing overhead can easily discover better ones.

For each interface, a *threshold* value is maintained and defines how far the search will go on that interface. Technique used for correct threshold setup is presented at section 4.4

4.3 Tests Results

In order to validate our protocol, some tests have been issued on the brand-new *RUN Active Network Simulator* developed at Université of Liège - a cross between the legacy ANTS [1,2,3] toolkit and the generic SSFNET network simulator [10].

Compared to an earlier "naive" policy (testing every target in table order), *RADAR* is 6 times faster for a complete discovery on the topology shown in figure 1 and saves about 50% of the search overhead (only 234 Kbits of AYA capsules generated and forwarded against 439 in the naive approach), which lowers its cost roughly to the same level as capsule code download.

4.4 Setting the R.A.D.A.R. Discovery Threshold

The *threshold* value used to define how far[7] an interface must search is an important parameter for the whole protocol. It will define the trade off between search cost and routes efficiency, but also the 'complexity' of the discovered active topology. So a threshold-based limit allows us to set a less constraining absolute `MAX_HOPS` parameter, and lets the protocol search 'as far as it has to'.

Simulations have been carried out to find out a threshold definition algorithm that would be able to *adapt* to the underlying topology and that would achieve good resource use. Using only the cost of the first found neighbour to compute the threshold leads to a high dependence on parameters like active nodes density or spanning tree growth at each node.

4.5 Adaptive Threshold Setting Algorithm

The solution we adopted is based on the well known *additive-increase-multiplicative-decrease* generic algorithm widely used in network engineering. It tries to find a balance between discovered neighbours and potentially remaining neighbours.

Every time an AYA capsule successfully returns to an active node, the *threshold* of the sending interface is reduced multiplicatively by α. Keeping from 1/2 to 2/3 of the previous value has given good results on every tested configuration (see figure 3). When all targets of the current *cost ring* have been probed, *threshold* is incremented by the amount of non-responding[8] targets. A pseudocode for *RADAR* behaviour is given in listing 1.2.

Note that after the first neighbour has been found for an interface (and has reduced *threshold*), the threshold reduction depends on how many targets that neighbour covers (i.e. every time such a covered target is probed, the neighbour replies and the threshold is further reduced). So, finding a neighbour located at the 'edge' of the network will not have much impact on neighbours search while finding a neighbour in the 'core' of the network quickly stops the search.

Note also that the threshold is only used when a ring search wasn't completely successful. If every target of a ring has replied, then the search won't go any further, regardless of the current threshold.

4.6 Using R.A.D.A.R. on Dynamic Topologies

In order to support active nodes arrival or departure at any time *RADAR* needs a timeout mechanism that will force it to re-check some targets (for nodes that are still active neighbours or that have been active in the past). We still use AYA capsule to refresh the information, but the *threshold* parameter will only be modified if the situation has changed since the last known state.

RADAR will send refresh capsules only if it has received no traffic from the target address, thus avoiding unnecessary overhead on heavily loaded links. In

[7] in hops count from the searching node

[8] after 3 re-emission of the AYA capsules with a period of `DELAY`

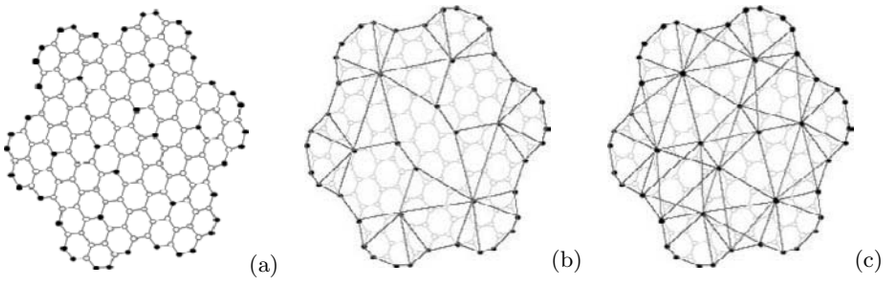

Fig. 3. *Discovery using adaptive (AIMD) threshold: (a) physical topology, (b)* $\alpha = 0,5$*, (c)* $\alpha = 2/3$

addition, a *binary exponential back-off* mechanism will make the refresh attempts more and more spaced if the target does not send data capsules between AYA probes. By doing so, a node will usually wait a long time before discovering that a silent node is down while the crash of a router that exchanges a lot of data will be discovered almost immediately.

5 ANTS Modifications

5.1 Output Channels

The *RADAR* protocol requires some extensions to the legacy OutChannel class from ANTS, mainly to support *grabbable capsules* facility. A derivate output channel using ANEP/IP rather than ANEP/udp will be automatically generated when the next hop address can be recognized as an "unknown" address (i.e. when the requested route hasn't discovered any neighbour).

Moreover, it is now possible for OutChannels to become invalid due to a change in the ANTS routing table (done by a routing manager in response of some active neighbourhood notification)

5.2 Route Table Copy

RADAR requires a copy of the IP routing table for its discovery process. The *next hop* addresses of the table entries will be modified dynamically to reflect the topology knowledge of *RADAR*. The entries in this table will have to carry a *route cost* (preferably a hop count) as well as an identifier of the network interface card (using the next hop address is no more possible as it is now prone to dynamic updates).

Conclusions and Future Work

We have extended the ANTS platform to support discovery protocols of active neighbours. The presented protocol - *RADAR* - performs that discovery with

a cost comparable to code download when active nodes are dense or slightly sparse, but it could still be improved on very sparse topologies through a local *host cache service* like what is done in decentralized peer-2-peer networks [13].

On a single domain, *RADAR* almost always results in connected graphs[9]. When considering a hierarchical network, traffic-driven discovery is mandatory to interconnect subnets, and it will work better if border routers are active.

A feedback loop could be investigated to modify the heuristic α constant when the amount of discovered neighbours isn't satisfactory.

Thanks to the *traffic-based* discovery, any active application can remotely "guide" *RADAR* by contacting some specific targets (querying a directory of the domain's active node, using a multicast group, etc.). *RADAR* will then detect the new potential neighbours from reply-traffic analysis.

References

1. D. Wetherall, A. Whitaker : ANTS - an Active Node Transfer System. version 2.0. *http://www.cs.washington.edu/research/networking/ants/*
2. D. Wetherall : Service Introduction in an Active Network. *http://www.cs.washington.edu/research/networking/ants/ants-thesis.ps.gz*
3. D. Wetherall, J. Guttag, D. Tennenhouse : ANTS - A Toolkit for Building and Dynamically Deploying Network Protocols. *IEEE OPENARCH'98*, April 1998
4. P. Kakkar : The Specification of PLAN (Packet Language for Active Networks) Draft 1, University of Pennsylvania (July 12, 1999)
5. S. Berson, B. Braden : DANTE : Dynamic Topology Extension for the ABone. *ABone: Technical Specs - http://www.isi.edu/abone/DOCUMENTS/dante2.ps*
6. S. Berson, B. Braden, L. Ricciulli : Introduction to the ABone. *http://www.isi.edu/abone/DOCUMENTS/ABoneIntro.pdf*
7. R. Sivakunnar, S.W. Han, Vaduvur Bharghavan : PROTEAN : A scalable Architecture for Active Networks *Proceedings of OPENARCH'2000*, March 2000
8. Raghupathy Sivakumar, Sungwon Ha, Sungwook Han, Vaduvur Bharghavan: The Protean Active Router: Design and Implementation. *The 14th IEEE Computer Communications Workshop (IEEE CCW'99), Invited Presentation*, October 1999.
9. A. Ghosh, M. Fry, J. Crowcroft : An Architecture for Application Layer Routing. *Lecture Notes in Computer Science 1942, "Active Networks", Springer, 2000 (IWAN 2000)*
10. Scalable Simulation Framework for modeling the Internet. *http://www.ssfnet.org*
11. Larry Peterson (Editor). NodeOS Interface Specification. *DARPA AN NodeOS Working Group Draft*, 1999.
12. Andy Collins, Ratul Mahajan, and Andrew Whitaker. The TAO Algorithm for Virtual Network Management. *Unpublished work.* December 1999. *http://citeseer.nj.nec.com/collins99tao.html*
13. Clip2. The Gnutella Protocol Specification v0.4 *http://www9.linewire.com-/developer/gnutella*
14. D.R. Boggs : Internet Broadcasting, *Ph. D. thesis, Electircal Engineering Dept., Stanford*, 1982 and *Technical Report CSL-83-3, Xerox PARC Palo Alto, California*

[9] The graph might be partitioned in some rare case, but traffic-driven discovery will usually reconnect them if some capsules cross the partition line