Catholic University of Louvain
Faculty of Applied Sciences
Computer Science Department

# Discovery of Internet topology through active probing

Thesis submitted in partial fulfillement
of the requirements for a degree of
Civil Engineering in Computer Science
by Gregory Culpin

Promoter : Professor Olivier Bonaventure
Readers : Benoit Donnet & Bruno Quoitin

Louvain-la-Neuve
August 2006

# Acknowledgements

I wish to express my gratitude to everyone who contributed to this thesis, with particular thanks to Olivier Bonaventure, Bruno Quoitin, Benoit Donnet, Angélique Baclin, Alain Guillet and my parents who have all contributed with their time and effort.

# Contents


**1 Introduction** **1**

1.1 Motivations for discovering the Internet topology . . . 3
1.2 Challenges and goals . . . 6
1.3 Thesis contributions . . . 7
1.4 Thesis organization . . . 8

**2 Background** **9**

2.1 Internet topology overview . . . 9
2.1.1 AS-level topology . . . 9
2.1.2 IP-level topology . . . 12
2.1.3 Crossing the levels . . . 15
2.2 Internet routing policy concepts . . . 19
2.2.1 Inter-AS routing with BGP . . . 19
2.2.2 Intra-AS routing . . . 21
2.3 Alternative topology information sources . . . 22
2.3.1 Routing registry information . . . 22
2.3.2 BGP routing information . . . 23

**3 Active probing and related work** **25**

3.1 Classic approach to active measurements . . . 25
3.1.1 Traceroute tool . . . 26
3.1.2 Existing discovery systems . . . 28
3.1.3 Limitations and issues . . . 31
3.1.4 Summary evaluation of current active probing systems . . . 34
3.2 Doubletree . . . 35
3.2.1 Discovery process using trees . . . 35
3.2.2 Two-phase probing . . . 35
3.2.3 Stop sets and simple stopping rule . . . 36

3.2.4 The Doubletree algorithm . . . 36
3.2.5 Choice of initial distance . . . 38
3.2.6 Inter-monitor communication load and Bloom filters . . . 38
3.2.7 Advanced stopping rule using CIDR prefixes . . . 39
3.2.8 Capping and clustering . . . 39
3.2.9 Evaluation . . . 40

**4 Design and implementation** **41**
4.1 Design choices . . . 41
4.1.1 Choice of a network simulation framework . . . 41
4.1.2 Functional requirements . . . 43
4.2 Implementation . . . 44
4.2.1 Discovery preparation . . . 45
4.2.2 Discovery simulation . . . 49
4.2.3 Discovery performance evaluation . . . 51

**5 Test design and results** **55**
5.1 Test design . . . 55
5.1.1 Identification of heuristics . . . 56
5.1.2 Experimental planning approach . . . 57
5.1.3 Effect tests layout . . . 59
5.1.4 Choice of topologies . . . 59
5.2 Tests and results . . . 60
5.2.1 Standard deviation between repeated experiments . . . 61
5.2.2 Impact of source and destination placement . . . 61
5.2.3 Impact of clustering and capping . . . 66
5.2.4 Summary results . . . 67

**6 Summary conclusions** **69**

**Appendices** **79**

**A Interaction with a C-BGP instance** **81**

**B C-BGP patch source code** **83**
B.1 Added/modified functions to `bgp.c` . . . 83
B.2 Added/modified functions to `as.c` . . . 85

**C XML discovery files** **87**
C.1 XML Schema – `discovery.xsd` . . . 87
C.2 Example of generated discovery file – `example1.xml` . . . 87

**D Guidance for script usage** **89**
D.1 AS classification script usage . . . 89
D.2 Discovery setup script usage . . . 89
D.3 Discovery simulation script usage . . . 89

**E Perl source code** **91**
E.1 AS-level topology classification script – `classifyAS` . . . 91
E.2 Discovery setup script – `setupDiscovery` . . . 96
E.3 Discovery simulation script – `performDiscovery` . . . 104
E.4 Doubletree algorithms module – `doubletree.pm` . . . 110
E.5 Topology model module – `topology.pm` . . . 119
E.6 Tool module – `tools.pm` . . . 126

# Chapter 1

# Introduction

The Internet is a massive network used and built by many people. It is composed of well-known hardware and software that allow large amounts of information to circulate within it. *Links* carry IP data packets from one router to the next, *routers* receive packets from a link and send them onto another, *routing protocols* determine the paths these packets will take. Although unexpected interactions and vulnerabilities are still being discovered in these components, published standards such as [1, 2, 3, 4, 5] contain well-documented information about how they work in order to ensure their interoperability.

On the other hand, the way these components are arranged and configured to form networks is much less well-understood. At an organisational level, the Internet is composed of more than 18,000 independently-administered networks [1] that interact to coordinate the delivery of IP traffic. Each network is called an autonomous system ($AS$) and uses the textitBGP [6] global routing protocol to exchange information about how to reach individual blocks of destination IP addresses (*prefixes*). An AS applies local policies to select the best route for reaching each prefix and to decide whether to propagate this route to neighbouring ASs.

The distributed structure of the Internet allows it to grow organically, without a centralized administration and without the need to reveal internal policies or topology to others. Although this structure has undeniable benefits, it makes precise knowledge of its topology difficult to obtain. As a consequence, the general strategy of topology discovery is constrained to acquiring local views of the network from several vantage points and merging these views in order to get a presumably accurate map.

[1] Source CIDR report - http://www.cidr-report.org

Internet topology information can be gathered from several sources, e.g. BGP table dumps [7, 8], routing registry databases [9], active measurements . According to the current state of research [10], it seems that no one source gives a better view of the network. Not only do they each have their own perception of the same network, but hazardous features of the network further complicate the task by leading to conflicting or incoherent views in between their own vantage points. Moreover, the actual topology of the Internet remains unknown making it difficult to compare, benchmark and optimise the results of the various discovery algorithms.

Internet topology can be investigated at different granularity levels, i.e. interface, router or AS, with the final aim of obtaining an abstract representation. This representation is typically a *graph* of *nodes* (vertices) and *links* (edges) which represent respectively the set of interfaces, routers or ASs and their physical connections. Acknowledged information about the Internet's components, their interactions (e.g. high-level policy routing, low-level IP forwarding), its structure (e.g. characteristic hierarchy of AS roles in the network [11]) and information sources is supplied in Section 2.1.

This study concentrates on one of the most popular sources, namely *active probing* [12], which consists of sending out probe packets into a network from a sender to a receiver. The standard way of obtaining such measurements is to make use of the *traceroute* tool (proposed and implemented by Deering and Jacobson in 1989) from a small number of monitors to a large number of destinations. Acquired paths are then agglomerated and an approximate topology is inferred. Several papers have however, underlined the fact this may introduce unwanted biases [13, 14, 15] and that scaling up the number of monitors would give a more accurate view of the topology. This current approach and its shortcomings are discussed in Section 3.1.

The current trend towards a relevant highly distributed system [16] by increasing the number of monitors is not a trivial matter. Duplication of effort close to the monitors wastes time by re-exploring previously discovered parts of the network, not to mention that probes converging from a set of sources towards a given destination can resemble a distributed denial-of-service (DDoS) attack as the probes converge from a set of sources towards a given destination. Prior work at LiP6 has elaborated a cooperative topology discovery algorithm, *Doubletree* [17], which deals with these issues while keeping high network coverage and which is presented in Section 3.2.

The above-mentioned difficulties further justifies the usefulness of performing the discovery in a known, controllable and predictable environment, i.e. by the means of a network simulator. Its utilization facilitates detailed analysis of a discovery algorithm's performance on a known topology, and also avoids the practical problem of limited host access in the real network. In order to model this process in the most realistic way, C-BGP [18], an advanced open source network simulator, is used and provides a complete solver for the inter- and intra-domain routing decision process.

## 1.1 Motivations for discovering the Internet topology

In its early years, monitoring Internet topology was a tractable problem. However, after experiencing exponential growth during the 1990's, inferring connectivity from any source of information has become a daunting task. Although Internet access growth is slowing in most developed countries (North America 110.4% during 2000-2005)[2], it is still increasing rapidly in the developing world (Asia 232.8% during 2000-2005). In addition new and existing users are placing greater confidence in existing functions such as e-Commerce and e-Government, together with emerging applications such as VoIP will tend to increase network loads giving traffic growth rates in excess of the simple growth rate in the numbers of users. Thus knowledge of the Internet topology is critical for engineering, research and many collaborative activities.

An accurate model of the topology and structure of the Internet is important in order to help researchers determine and fix problems that it faces [19]. To this day, over 150 publications have been based on or have used data from CAIDA's *Skitter* [12], a well-known tracing infrastructure, in many areas such as security (e.g. topology robustness) against attacks [20], IP traceback architecture [21]) and routing (e.g. BGP configuration anomalies [22], multicast scaling [23] understanding). Recent interest underlines this need, because such modelling can more generally provide awareness about how traffic flows, how resources are used, and where infrastructural vulnerabilities may lie.

Hereafter are described a series of motivations for discovering the Internet's topology.

[2]Source Internet World Stats *http://www.internetworldstats.com/stats.htm*

### Growth Modelling

The growth and uptake of the Internet has consistently confounded the most optimistic predictions. Access to good quality topology data permits researchers to show that the Internet is not as chaotic as previously assumed. Faloutos et al. [24] demonstrated that a number power laws apply to its topology, with further work [25] providing additional confirmation. Such work enhances the ability to understand the drivers and other key factors influencing Internet growth [26]. This is now of more than pure academic interest, as the Internet is a key infrastructure of the global economy.

### Simulations

One consequence of the magnitude of the Internet and its commercial ubiquity is that it is difficult to change the software it uses: installation or updates of software that run inside the network must not only be backward compatible, but it must also have been sufficiently well tested to prove its value and that its deployment will cause no or limited interruption of connectivity. These requirements are very difficult to fulfil when developing new functionality, therefore commonly leading to the current patchwork of incremental solutions. As a result, evaluation of solutions in a simulated environment are becoming more frequent thus attaching even greater importance to accurate knowledge and understanding of Internet design.

Simulations can take place in inferred topologies from real network data, e.g. Skitter measurements. They can alternatively be performed inside topologies constructed by generators such as BRITE [27], GT-ITM [28], TIERS [29], IGen [30] or GHITLE [31]. These generally base themselves on previously observed data to heuristically generate realistic environments.

### Protocol Effectiveness

Internet topology should be transparent to the network protocols used across it, nonetheless their effectiveness can be compromised. Topology has been shown to impact in several ways for example; on the effectiveness of denial-of-service countermeasures [32, 33] and on routing protocol performance Labovitz et al. [34]. Such understanding will have a contribution to future protocol development.

### Network Management

Network topology information is useful in deciding whether to add new routers and to determine whether current hardware is configured correctly. In large corporations and administrations, delocalisation, outsourcing and collaborative teams working in many differing locations, makes dependent than ever on networks and their effective management. In these environments where a short network failure or significant congestion can result in significant implications for businesses and administrations, the ability for network managers to analyse the topology in real-time will enhance their ability to respond effectively.

Within small networks where the topology is well documented and relatively static, real-time tools are less of an issue, but within large scale environments probably active 24/7, changes in requirements can be significant and rapid. Topology discovery tools will contribute significantly to the forward planning of capacity and performance for example by assisting in understanding neighbouring ASes' BGP Policies and peering agreements which may not otherwise be readily available.

### Siting

In this context, siting consists of locating the place where a structure is to be located. For example, a network map can help users determine which ISP or AS to join in order to provide best connectivity to and from the rest of the Internet, i.e. minimize latency, maximize available bandwidth, etc.

As another example, consider a company managing a content distribution network (CDN) and wanting to place replicas of website content in data-centres hosted by different ASs. It can identify the IP prefixes and ASs responsible for a large portion of the site's traffic. Given an accurate view of the topology, it can then identify the best locations for its replicas.

### Topology-aware algorithms

Topology information enables a new class of protocols and algorithms that exploit knowledge of topology to improve performance. Examples include topology-sensitive policy and QoS routing, and group communication algorithms with topology-aware process group selection, e.g. building of overlay multicast trees.

## 1.2 Challenges and goals

The first objective of this thesis is to provide a complete yet clear view of the current state of Internet topology knowledge, while focusing enough on active probing methods to further be able to, firstly, determine the functional requirements with which the implemented framework must comply, secondly describe the simplifying hypotheses within which its simulations would take place.

Once the requirements are set, the second objective is to design and implement, on top of an appropriate network simulator, a simple traceroute-based discovery algorithm and the more elaborate Doubletree discovery algorithm. An important part of the design is that it must allow efficient discovery processes to be performed on large topologies and must supply detailed information about its outcome. It should also be extensible by allowing easy modifications or additional features to be included. Compatibility by providing support for other tools' formats can also be a goal.

The third objective is to analyse the implemented algorithms' performance and, if applicable, establish a comparison with previous work results. Indeed, detailed analysis could undermine them as many studies state only hypothetical node and link coverage as they have partial knowledge on the underlying topology.

The fourth and final objective is to find ways to optimise performance results in the light of previous Internet topology structure studies. The possibility of applying statistical tools should be considered for laying out such experiment plans ultimately leading to a optimizable model of the discovery process, or through the selection of more intuitive heuristic-based experiments by notably grasping opportunities of easy siting inside the simulated network and of total knowledge of its behaviour.

## 1.3 Thesis contributions

In this thesis, an effective framework is designed for analysing in detail active topology discovery. The framework uses an external network simulator to compute the network's response to the execution of two different active probing discovery processes, i.e. a classical multiple source traceroute approach and the distributed Doubletree collaborative algorithm aimed at being deployed at a large-scale level. Basic features of the framework not only allow it to characterize according to a relevant classification method unknown input AS-level topologies but also extend its realism by enabling an underlying router-level topology to be additionally specified. The implemented network discovery processes can then executed inside the simulated network by taking into account performance considerations, such as memory consumption, since such large-scale simulations can lead to time and memory hungry applications. The design of detailed router-level node discovery distribution and a inter-level link discovery matrix analysis also gives the user an upper-hand on, for instance, which internal, peering or provider-customer links have been discovered in any part of the network, or what amount of load has been applied to individual levels or edges throughout the probing process.

Using the designed tools, the elaborate Doubletree algorithm is then subjected to an in-depth analysis of its performance at discovering a large-scale AS-level topology. Following the observation that such simulations were costly and that blindly choosing tests to optimise their performance would be a difficult task and suboptimal solution, the generation of statistically optimal plans for the discovery process's modelling was considered and described, although regretfully set aside after the identification of unresolvable issues in the given amount of time. However, a series of specific tests were designed and deployed during a two-month period in order to attempt to optimise coverage results by making the most of the benefits of operating inside a simulated framework and accordingly developed tools. Heuristics are therefore sought after evaluating several source and destination influences on the discovery process while being based on the topology's inferred AS-level hierarchy. These evaluations are then carried further through the simulation of source clustering and destination capping techniques which are also shown to provide efficient ways of probing the network in an even more friendly-manner.

## 1.4 Thesis organization

- **Chapter 2** provides some background for understanding the contributions made in the thesis, with a top-down description of Internet topology, routing policy concepts and available information sources for topology discovery.
- **Chapter 3** describes related work regarding traditional active probing schemes, the network-friendly Doubletree algorithm, and their respective limitations.
- **Chapter 4** explains the design choices and further details the implementation used for the simulations.
- **Chapter 5** elaborates on the objectives initially set out by the thesis and identifies ways to reach them. The description of the performed tests is given with their results.
- **Chapter 6** summarizes the thesis's results and contributions and gives further directions in which to develop the study.

# Chapter 2

# Background

*This chapter provides a top-down overview of Internet topology discovery through the two main views that characterize it: the AS-level and IP-level topologies. The importance of knowing the latter is further shown by the description of crossing-level methods. Internet routing policy concepts are then covered in support of the summary of the alternative information sources, leaving aside the main source for topology information, namely active probing, for the following chapter.*

## 2.1 Internet topology overview

In a communications network, network topology is the pattern of interconnection between nodes. The Internet has different types of network topology, most of which can be organized by level of granularity. Four distinct levels are commonly made besides the general distinction; starting at the highest these are: AS-level, PoP-level, router-level and interface-level.

This section describes these levels in a top-down fashion, the information sources available for their study, the difficulties in acquiring their respective topologies, the current state of knowledge about them, and finally the ways in which they relate.

### 2.1.1 AS-level topology

An *autonomous system* (AS) is either a single network or a group of networks that is under the control of a single administrative entity, typically an Internet service provider or a very large organization (e.g. a university, a business enterprise or division) with independent connections to multiple networks. An AS is also sometimes referred to as a *routing domain*. Each AS is identified by a unique 16-bit number assigned by IANA.

This section describes the relationships observed between ASs and their impact on routing. It also details the evolution of AS topology discovery, the methods used for inferring it and the analysis made about its structure.

### AS relationships

In the Internet AS topology graph, an edge between two ASs (nodes) represents a business relationship or *policy* which results in the exchange of Internet traffic between them. An AS can have one or more relationships with different kinds of neighbouring ASs. Each relationship may correspond to several distinct physical links.

On one side, an AS's *access links* connect to customer networks. Customer networks buy Internet connectivity from the AS. On the other hand, *peering links* connect to transit providers and private peers with whom exchange of traffic is negotiated in order to obtain better connectivity to them and their customers. As a consequence, no transit traffic is allowed through peering links except traffic with the peer or its customers. These are the most observed relationships in the network and are usually referred to as the *provider-to-customer* (p2c), *customer-to-provider* (c2p) and *peer-to-peer* (p2p) relationships.

A less common relationship found in the Internet is called the *sibling-to-sibling* (s2s) relationship. This relationship generally resides between two ASs of a same company. The key difference with peering is that siblings exchange all kinds of traffic, not only between their respective customers. An s2s relationship covers everything except the p2c, c2p and p2p relationships. It can appear in various cases such as, when two ASs act as backups for each other, or when two ISPs merge and decide to become siblings instead of merging into a single AS which can be very complex. Two peering ISPs have a special agreement for specific prefixes for which they transit all kinds of traffic for each other.

These relationships have a major impact on routing in the Internet as shown by a study from Tangmunarunkit et al. [35]. Inside an AS, routing uses *hop-count* as a metric, but because intra-domain protocols support hierarchies, the resulting paths are not always the shortest in terms of *hop-distance*. Between ASs, routing is determined by policy. Many Internet path lengths thus may also benefit from a detour [36, 37] which would incur more router-level hops than shortest-router-hop path routing. For simulation purposes, it is therefore most appropriate to model the network with policy-based routing rather than AS shortest path-based routing.

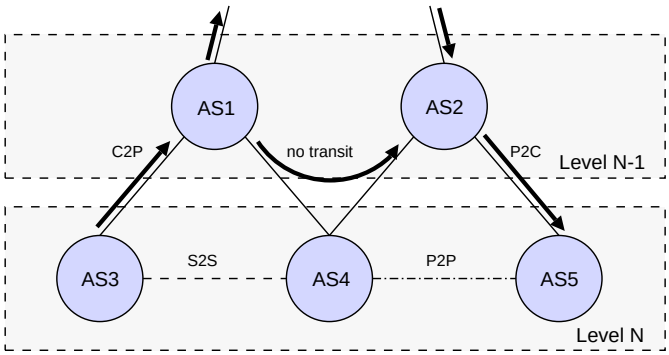


Figure 2.1: Relationships between ASs

### Inferring an AS-level topology

In the absence of a global registry, the first AS-level structures were inferred from publicly available BGP table dumps like those at Oregon Route-Views [38], RIPE-NCC [39] and RouteServer [40]. Early research assumed that two ASs were linked if their AS numbers were adjacent in an AS path. Gao and Rexford [41] then made a substantial advance by noticing customer-to-provider links create a hierarchy. Gao [11] went on to identify the peer-to-peer and sibling-to-sibling relationships.

Inferring these relationships is a problem of its own. In his study, Gao [11] first tackled the problem by developing an inference mechanism which extracts information from BGP tables and summarized the *valley-free*[1] property of AS paths. Subramanian et al. [42] formulated AS relationship assignment as an optimisation problem, Type of Relationship (ToR) problem. Battista et al. [43] proved its NP-completeness and presented an approximately optimal solution. Gao and Xia [44] then evaluated the accuracy of these algorithms and improved them by introducing techniques on inferring relations from partial information. This improvement was possible thanks to the study of Bonaventure and Quoitin [45] which show that uses of the *BGP community* attribute indicate relationships amongst ASs.

### AS hierarchy

ASs in the topology may differ according to the role which they address. They can vary in size, type and number of relationships they have with their neighbours. Such differences have lead to various informal definitions of AS hierarchy. The most common is known as the *tier hierarchy*.

[1] After traversing a provider-to-customer or a peer-to-peer edge, the AS path cannot traverse a customer-to-provider or peer-to-peer edge. In other words, an AS does not provide transit between any two of its providers or peers.

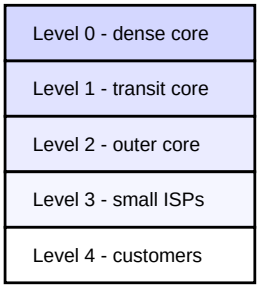


Figure 2.2: AS hierarchy

In [42], Subramanian et al. propose a 5-level classification of ASs based on a previously inferred topology graph. Typically, a customer should be at a lower-level than its provider(s). The algorithm detailed in the authors' paper is applied to a directed graph to discover the belonging of ASs to the hierarchy represented by Figure 2.2.

It identifies a clique[2] of *tier-1* ISP networks, which are the largest network providers and which compose the *core* of the inter-AS topology. A tier-1 network must therefore peer with every other tier-1 network. *Tier-2* networks, many of which are regional network providers, peer with some networks but purchase transit in order to reach at least some portion of the Internet. Far away from the core are the *stub networks* that are the leaves in the topology, meaning they solely purchase transit from other networks to reach the Internet.

A first attempt to classify the AS topology was performed by Govindan and Reddy [46] and based on node degree; ASs with a large number of neighbours are placed above ASs with small node degree. However, a simple degree-based approach does not capture the essence of the tiers in the hierarchy.

### 2.1.2 IP-level topology

As shown in Figure 2.3, an AS is composed of a collection of routers that are interconnected. Routers are decision-making entities composed of multiple interfaces. An *interface* is a router's attachment to a link, e.g. an Ethernet interface. A *link* is a communication medium offered by the underlying link-layer[3] protocol over which the main Internet Protocol (IP) may transmit packets, e.g. an Ethernet network.

Links between routers belonging to the same AS are called *core links*. Links between routers of different ASs are known as *edge links*. A router ending an edge link is a *border router*. Remaining routers inside the AS are its *core* or *internal routers*.

IP-level topology is therefore often referred to as both *interface-level* and *router-level* topologies.

[2] A clique is a fully connected subgraph, meaning everyone in the clique interacts with every one else.

[3] These point-to-point links may not be point-to-point beneath IP: a layer-2 switch or other multiple-access medium may be used.

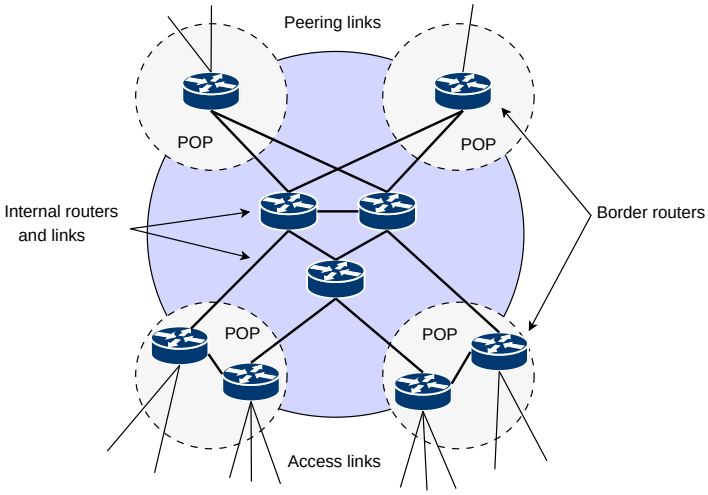


Figure 2.3: Detailed topology of an AS

Methods for inferring an interface-level topology are typically based on hop-limited probing, i.e. traceroutes, sent from one or more locations, and are the matter of the next chapter. A router-level topology is then inferred from the interface-level topology, while the only yet significant issue separating them is the one of assigning interfaces to routers, known as the *alias resolution* problem, and is further described in Section 2.1.3.

A *point of presence* (PoP) is a collection of routers owned by an AS in a specific location (city or suburb). This distinction can lead to an intermediate topology, i.e. the PoP-level topology, which can be produced by adding information about geographic location to the inter-AS topology. Different ASs also sometimes have routers in the same building, such places are known as a co-location facilities or *exchange points*. An analysis at this level is useful for understanding the geographic properties of Internet paths, e.g. it can provide constraints about latency between two PoPs.

### IP overview and addressing

IP [47] is the network-layer protocol used by the Internet. Currently IPv4 is mostly used, but a growing number of nodes and networks also implement IPv6 [48]. The routing behaviour of IPv4 and IPv6 are very similar, as are the routing architectures and topologies used within the two protocols; thus in the context of this thesis, it will simply be referred to it as "IP" since most of what follows is applicable indifferently to either protocol.

Interfaces are assigned IP addresses. In IPv4 an interface usually has one IP address, while in IPv6 an interface typically has more than one address assigned to it. IPv4 addresses are 32 bits

long and are written in "dotted-quad" decimal notation, such as 130.104.1.233. IPv6 addresses are 128 bits long and are written in colon-separated hexadecimal notation, such as 2006:075:4::1 (where :: is the abbreviation of two or more consecutive groups of zeros).

Since IP addresses are difficult to remember, many hosts have associated names to them. The Domain Name System (DNS) [49] handles the mapping of these names to IP addresses and vice versa. Domain names are maintained in a hierarchical tree structure stored in a distributed database. A DNS query is then typically sent to a local name server which, if the answer is not available in cache, forwards it to one of several DNS root servers at the top of the tree. These servers maintain an authoritative list of DNS servers responsible for top-level domains, (such as .org), which in turn delegate responsibility for individual domain names, (such as traceroute), which in turn answer the queries for IP addresses of their subdomains or hosts (such as www).

### IP routing

IP provides the transport layer with best-effort, connectionless delivery of a packet from one node to any other node (or set of nodes in the case of multicast) in the Internet, regardless of network topology or underlying link-layer technologies used.

If a packet is sent to a destination which is not on the same link as the source, it will traverse one or more routers on the way to its destination. Every router typically maintains a *routing table* which holds, for every destination, the address of the next router and outgoing interface through which to send the incoming packet. When receiving a packet on an interface, routers consult their routing table and send the packet to the next router or to the destination. This is known as *destination-based* routing. As a consequence, the path taken by a packet is unknown at the time of sending but is independently determined by each router along the way. Note that *source-based* routing, which consists in specifying a list of routers a packet should traverse, is inbuilt within IP although most routers discard such packets because of security issues.

These routing tables can either be configured manually, which is feasible for small networks or, most coomonly, generated automatically and handled by routing protocols which propagate information on how to reach destinations in the network. The functioning of these protocols is further detailed in Section 2.2.

### Address prefixes

In order to minimize the size of routing tables, IP addresses are gathered in contiguous blocks of addresses known as *prefixes*. All addresses with the same prefix have their first n bits in

common, where n is the prefix length. A common notation for a prefix is the first address of the prefix followed by a slash and the prefix length, such as 130.104.0.0/16 for an IPv4 prefix. In the early days of the IPv4 architecture, addresses were divided into classes according to where they were located in the address space and prefixes were 8, 16 or 24 bits long according to the class of addresses they contained. Prefix length is no longer constrained to these values since the introduction of Classless Inter-Domain Routing (CIDR) [50].
When looking up a destination in the forwarding table, more than one prefix may match; in this case the entry for the longest prefix is used. At the top of the table is a default route, the entry to use when no other entry is matched. The default route is associated with the prefix 0.0.0.0/0, which is the least-specific prefix possible: any other matching prefix will override this default route by virtue of being more specific.

### 2.1.3 Crossing the levels

#### From interface-level to router-level

In order to infer a router-level map from IP paths, a means must be found to assign interfaces to routers. This problem is known in literature as *resolving aliases*. It has received recent attention as Gunes and Sarac [51] underline that alias resolution is an important yet often overlooked component of the traceroute-based Internet map construction process. Consider the following example in Figure 2.4 in which various interfaces are joined by IP links. The assignment of interfaces C1 and C2 to different routers significantly changes the topology as additional router-level nodes and links appear.

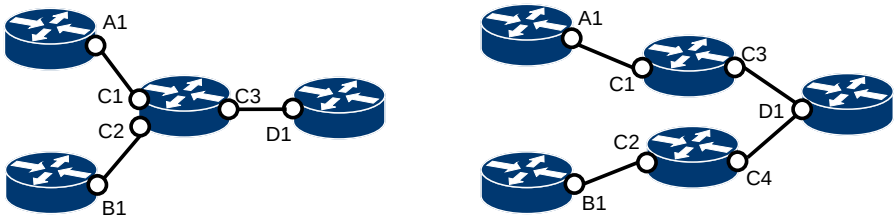


Figure 2.4: Alias resolution : assigning interfaces to routers

Such inaccuracies may indeed lead to the appearance of non-existing links or disappearance of existing links in the sample topology. This would quite possibly have an impact on the results of papers such as [52, 13] having used the resulting maps.

Several approaches currently exist for alias resolution:

- **Address based method**: this method consists of probing several interfaces of a suspected router in order to make it reply with an ICMP [53] error message. Certain implementations of ICMP error reporting reply using as source the IP address of the router interface that is on the shortest path back to the initial probe source. Their comparison can thus determine that they belong to the same router although other implementations or alternative configurations may alter that router's behaviour, e.g. an administrator can configure a router to ignore probes directed to them. Mercator [54] and Iffinder [55] are well known tools using this address based method.

- **IP identification based method**: when an IP packet is generated, the kernel puts a 16-bit value into its IP identification field, typically implemented as a monotonically increasing counter. This can be used to ascertain if ICMP error messages are coming from a same router after being sent to different IP addresses. Rocketfuel's *Ally* tool uses this approach as well as complementary techniques not described here such as TTL value comparison.

- **DNS based method**: this method is based on similarities in router host names and works when an AS uses a systematic naming scheme for assigning IP addresses to router interfaces. This method is especially interesting as it can work even if a router does not respond to probes directed to itself. Ally uses this technique against unresponsive routers with the help of the *undns* tool [56].

- **Graph based method**: this method extracts from traceroute outputs a graph of linked IP addresses in order to infer likely and unlikely aliases as described by Spring in [57]. It is based on two assumptions: (1) if two IP addresses precede a common successor IP address, then they are likely to be alias, and (2) two addresses found in a same traceroute are unlikely to be alias. This method is mainly used as a preprocessing step to reduce the number of probe pairs for an active probe approach.

- **Analytical Alias Resolver (AAR) method**: Gunes and Sarac [51] recently presented a graph theoretic formulation of the alias resolution problem and developed the AAR algorithm to solve it. Given a set of path traces, the algorithm utilizes the common IP address assignment scheme to infer IP aliases from the collected path traces.

### From interface-level to PoP-level

The pioneering work of Rocketfuel [58] provided techniques for inferring detailed PoP-level topologies using traceroutes; IP addresses appearing in traceroute paths are mapped to their corresponding PoP by performing reverse DNS lookups. In later work, Teixeira et al. [52] found that inferred topologies had significantly higher path diversity[4] and they suspected that the large number of false links were due to imperfect *alias resolution*[5]. However this could not explain the false PoP-level edges. Recent developments in [59] show DNS misnamings to be a major source of these false edges and offer ways to fix them.

### From interface-level to AS-level

Jamin and co-authors [7] showed that many existing links do not actually appear in BGP. Fortunately, BGP tables are not the only source for AS-level topology information. *Active probing* results, from tools such as CAIDA's Skitter, can also be used. IP addresses gathered in IP paths can be mapped to AS numbers by using, for example, BGP table dumps or Internet registry data. Broido et al. [60] reports that the obtained topology differs to BGP inferred ones in that it has much denser inter-AS connectivity. It is also richer because it is capable of exposing multiple points of contact between ASs, this is in contrast to BGP table dumps which only provide information on whether two ASs peer or not.

Chang et al. [7] proposed a means to identify border routers of an AS, but this is not a trivial problem. Indeed, the IP addresses of a border router might either belong to its own AS, to the AS of a peer, or to that of a third party such as an *Internet eXchange Point* (IXP) whose core is typically composed of one or several Ethernet switches. Following the example shown in Figure 2.5, AS1, AS2 and AS3 are all connected to an IXP with one of their router's interfaces being part of the IXP's subnet. However, both AS2 and AS3 have dotted peering links with AS4 and of which the end-point addresses could well belong to the latter's address range.

Mapping IP addresses to AS number is not as simple as it may seem. WHOIS data is often incomplete and out of date, while approaches based on BGP table dumps also have significant limitations. A common problem is the origination of a same prefix by multiple ASs (MOAS). Zhao et al. [61] showed that the number of conflicts is not trivial (the median value in 2001 was 1294 conflicts).

[4] Distinct number of AS paths that exist between an AS and the rest of the Internet

[5] Task of identifying and grouping the IP addresses belonging to the same router. see Section 2.1.3 for further details

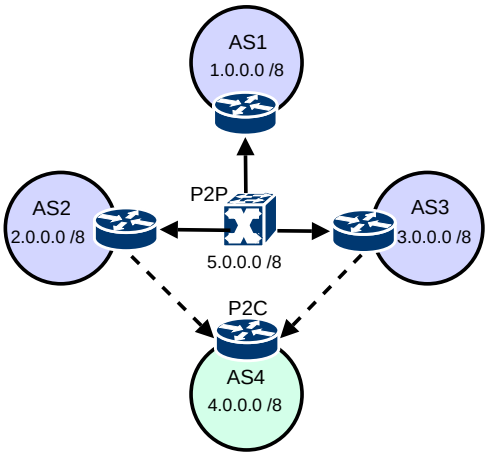


Figure 2.5: Border routers of peering ASes at an Internet exchange point (IXP)

An IP-to-AS mapping study by Mao et al. [62] identified that 10% of traceroute paths contained one or more hops that did not map to a single AS number. Furthermore, mapping IP addresses to AS numbers paths resulted in loops in the inferred AS-path for about 15% of the node-level paths examined. Loops not being permitted by BGP and this indicates an error in mapping.

The authors improved accuracy by proposing heuristics comparing BGP-derived AS paths against traceroute-derived AS paths and by performing reverse DNS lookups. The heuristics, though effective, are labour-intensive and mostly ad hoc; in [63] the same authors improved on this result and proposed a systematic way to perform the same tasks using dynamic programming and iterative improvement.

Although these topologies inferred from various sources present substantial differences, their comparison in [10] seems to have underlined fundamental characteristics of the network, such as its *joint degree distribution*[6]. However, the question of which most closely matches the actual Internet AS topology remains open; these methodologies appear to be quantitatively but not qualitatively different although each one approximates a different view of the Internet looking at the *data* (Skitter), *control* (BGP), and *management* (WHOIS) planes.

[6]else known as *node degree correlation matrix* can be summarized as being its average neighbor connectivity.

## 2.2 Internet routing policy concepts

Routing in the Internet is carried out by routers, each one using its forwarding table to determine on which link a packet should be forwarded. Manually maintaining these forwarding tables would be an impossible task. Fortunately, *routing protocols* are used to determine how packets traverse the different levels of Internet topology. They do so by exchanging information about the state of the network and deciding which paths to use to reach every destination.

The primary role of a routing protocol is link failure detection and avoidance. In addition, it allows preferences for different paths to be expressed by operators, shaping the way traffic flows through a topology. These paths can be preferred because they are cheaper, less utilized or more reliable. These preferences make up what is known as *routing policy*.

### 2.2.1 Inter-AS routing with BGP

#### Routing protocol basics

The routing protocol used in the Internet to communicate between ASs is the *Border Gateway Protocol* (BGP) [64, 2]. BGP behaves like a *distance vector* routing protocol: it tells its neighbours the length of the best path to reach each prefix. A router maintains a *routing table* in which it stores forwarding table information but in which it includes each path's length. Each router then sends a copy of this table to all of its neighbours. Each routing table is then updated with the available information from its neighbours; routes that are no longer available are removed and new shortest path routes are chosen.

In BGP, this information is exchanged between peers[7] by using messages known as *updates*. An update can either be an *announcement* which advertises a prefix, or a *withdrawal* which warns that a prefix has become unreachable.

BGP is however a variant of distance vector routing, namely *path vector* routing. Instead of exchanging path lengths and incrementing them, routers send each other whole paths and add themselves to a path before propagating it. This ordered list, the *AS-path*, constitutes one of the main attributes of a BGP announcement.

[7] Routers are said to be BGP *peers* if they exchange BGP routing information between them. A distinction is made between iBGP peers belonging to a same AS or eBGP peers belonging to different ASs.

### BGP route propagation

Typically, a prefix announcement is first made by its *origin AS*, the AS to which the prefix belongs. Routers in the origin AS announce the prefix to their peers with an empty AS-path. The announcement is then propagated from AS to AS, or not, depending on each AS's policy.

Each router maintains the latest route received from each peer in its routing table, also known as *Routing Information Base* (RIB). In the event of a prefix update received from a peer, the peer's route is updated and the *best route* to the prefix is recalculated. If the best route changes, the router announces it to its peers according to its routing policy. If the last known route to a prefix is removed, the router sends a withdrawal to its peers.

When a route is propagated to another AS, the current AS number is prepended to, i.e. added to the front of, the AS-path: the AS-path lists the ASs that a given announcement has traversed. This is used to avoid looping and for path selection: the shortest AS-path to a prefix is chosen as its best route, excluding local policy decisions. This is the route used to forward traffic for the given prefix: traffic is sent to the first AS in the AS-path. Traffic to the prefix therefore flows along the AS-path in the opposite direction to its advertisements.

### BGP routing policy enforcement

If several paths for a prefix are accepted from different neighbour ASs, BGP can be used internally to choose the appropriate path according to local policies. Its choice can be influenced using the *local-pref* parameter which allows an AS to rank routes in any order. This parameter can be defined for specific or groups of routes, e.g.. all routes announced by a particular neighbour. Common practice is to first choose routes from customer ASs, then from peer ASs. If neither have the sought route, a provider is chosen as last resort.

Another BGP attribute that can be used for applying policies is the *community* attribute which provides a way of grouping destinations to which routing decisions (such as acceptance, preference, and redistribution) can be applied. As opposed to the local preference attribute, a community stays attached to routes announced to other ASs. According to studies in the literature [45], a common utilization of the attribute is to affect the down-hill distribution of a route, thus enabling cross-AS policy coordination. Another is the tagging of incoming routes at all border routers which enables routing by all AS routers to be performed mainly on the basis of these communities. The idea is that, for example, customers with specific policies that require the modification of local policies in a provider network, set the corresponding community values in their routing updates.

While stub ASs mainly need to select among its access links to find the one to reach a given prefix, transit ASs need to care about the distribution of traffic inside their AS as well as among their inter-AS links. When a transit AS has multiple peering points with a destination AS, it typically has two available approaches for choosing the router-level path used to reach it:

- *early-exit* or *hot-potato* routing is the default and consists in choosing the closest peering point to the destination AS.
- *late-exit* or *cold-potato* routing consists in choosing the peering point closest to the destination. Although BGP information about internal topology is hidden, an AS can export Multi-Exit Discriminators (MEDs) to provide an order of preference for the different available peering points. However MEDs use is optional and therefore requires cooperation from both ASs to work.

Routing policies are the main cause of *asymmetric* routing in the Internet which means that paths used by packets on the way to their destination often differ to those in the opposite direction. This observation was first made by Paxson [65] upon data from 1995 and confirmed with data from 2002 by Amini et al. [66] that a considerable amount of paths in the Internet are asymmetric: most recently almost 70%. Asymmetric routing also varies according to which type of ASs are dealt with as He et al. [67] show that asymmetry is far more common between commercial networks than academic networks.

### 2.2.2 Intra-AS routing

In contrast to BGP used between distinct ASs, an Interior Gateway Protocol (IGP) is used inside an AS to choose paths within its own network. Typical IGPs like OSPF and IS-IS, use *link state* routing which consists in exchanging parts of the topology so that each router can assemble and obtain a globally-consistent view of the network. Path choice is then carried out based on complete knowledge of the topology.

Links are individually assigned a weight or cost on which the routing protocol relies to select the paths having the least cost, i.e. smallest sum of the weights of the links. Link cost can be modified in order to encourage or discourage its use, for example in case of congestion or unreliable conditions. As new information is sent through the network, inconsistent states may inject transient faults.

Some ASs explicitly determine the path between each pair of routers through the use of the Multi-Protocol Label Switching (MPLS). With MPLS, a label is assigned to a packet that enters

the network. Further routers make their forwarding decisions based on this label or on other fields in the packet, but not solely according to the packet's destination. This can cause trouble in network mapping as discussed in Section 3.1.3.

## 2.3 Alternative topology information sources

Several sources of Internet topology data are available and three of them have been predominantly used in recent years: Internet registries, BGP routing information, and active probing techniques such as traceroute measurements. This section describes the first two along with their advantages and limitations. Active probing is more thoroughly covered by the next chapter.

### 2.3.1 Routing registry information

Many publicly-available registries share information about the Internet and its topology. Regional Internet Registries [9] are organisations responsible for allocating AS numbers and IP address blocks, all of which are accessible using the WHOIS protocol. Internet Routing Registry (IRR) is another group of databases maintained by several organisations and containing documented routing policies. These policies are also available through the WHOIS protocol and are expressed in Routing Policy Specification Language (RPSL [68]).

Topology discovery using Internet registry information has several advantages. Firstly, they are simpler and more efficient to implement than active probing methods since they do not have to explore the network to obtain the topology and because information is grouped at specific locations. Secondly, they provide high-level information such as routing policies which are otherwise more difficult to obtain.

This information source also has limitations mainly due to the fact that they are based on data provided by ISPs and not based on the real state of the network. Firstly, the provided information is often incomplete for various reasons such as confidentiality and administrative overhead. Secondly, as shown in RIPE consistency check reports [69] registry data quality is questionable and often inconsistent as information about a same object in one registry overlaps and sometimes even contradicts information in other registries. Thirdly, due to their inherent nature, these registries are not able to precisely reflect the actual state of routing in the network. For example, it can not determine whether portions of the Internet are reachable or not, or whether backup links exist and are being used, etc. These limitations are the reason why current work has tended to focus on other information sources for topology discovery. Having said that, routing registries are still a useful source of information when combined.

### 2.3.2 BGP routing information

As opposed to link-state protocols such as OSPF[70] or IS-IS[71], BGP does not maintain any unified view of the network. Each BGP router chooses its best path for a specific destination which it propagates to its neighbours, leading to an individual view of the network for each router. This view depends on factors such as the choices made by its neighbours, such as the order in which it received their announcements, etc. This distributed nature calls for the use of information gathering methods in order to obtain the most complete common view of the topology.

Common BGP information sources are *looking glasses* and route servers [72]. A looking glass is a web interface to a BGP router which usually allows BGP data querying and limited use of debugging tools such as *ping* or *traceroute*. A *route server* is a BGP router offering interactive login access permitting to run most non-privileged router commands. Both are usually made public to help network operators in their debugging tasks, but they can also provide BGP information to properly crafted network discovery tools. A list of accessible looking glasses and route servers is available at [73].

A second source of BGP information is *BGP dumps*. Projects such as RouteViews [38] or RIPE-NCC [39] provide collected information from BGP routers around the world. Route collectors are deployed in various locations and peer with BGP routers from multiple ASs. They then periodically save snapshots of their state, known as *table dumps*, along with all routing updates received between the preceding and current snapshot, known as *update traces*.

There are several advantages to topology discovery methods based on BGP routing. Firstly, like routing registries, data has been gathered and is available at specific places. There is therefore no need to deploy an infrastructure for exploring the network. Secondly, unlike routing registry data, provided information by BGP corresponds to the actual state of the network, even though it only provides local views of it. Finally, BGP update traces further allows dynamic behaviour analysis such as backup link detection.

The disadvantages of using BGP traces is that, firstly, they do not reveal any information about router-level topology information of the Internet. Secondly, they require a non-trivial infrastructure for update collection and storage. Thirdly, they seem to provide a less complete picture of interdomain routing as for example using node-probing, confirmed by studies such as Broido et al.'s [60].

# Chapter 3

# Active probing and related work

*This chapter covers the active probing class of topology discovery methods and the manners in which they are currently deployed. Inherent limitations of active measurements are explained in addition to the problems encountered by their existing implementations. A summary of current issues then paves the way to the description of a needed larger-scale distributed discovery system, namely Doubletree, which will be the object of simulation and analysis in the following chapters.*

## 3.1 Classic approach to active measurements

Active probing is the third class of topology discovery methods which consists of deducing network topology from the behaviour of the network itself. A primitive approach is to use debugging facilities of network protocols such as IP. For example, in an IPv4 header, the *source routing* option can be used to specify a list of routers traversed by a packet, but such packets are often dropped by Internet routers as there are security implications and inter-AS policies could be breached. The *record-route* option allows a packet's path to be progressively stored by each router in the option data field, but this is limited to nine addresses, which is quite insufficient for discovering large networks such as the Internet. In addition, most routers also filter out packets with this option enabled.

Another protocol has revealed itself to be far more useful: the Internet Control Message Protocol (ICMP [74] and ICMPv6 [75]). It provides simple debugging features such as sending *echo requests* and receiving *echo reply* messages. These messages are typically used by the *ping* tool to discover if a node is present on the network and to measure the *round-trip time* (RTT) of the packet. Other ICMP functionalities such as subnet mask requests [76] could provide supplementary information sources but are also often discarded by administrative control.

Given the limited availability of helpful debugging facilities, the classic approach has been to

design tools sending specially crafted probe packets throughout the network in order to collect and analyse error messages sent back in response. The best-known example is the *traceroute* tool [77] which is described in the following section and which has been used by several projects worldwide in similar topology discovery quests. These projects' implementations are discussed along with their limitations partly due to the classic approach they take.

### 3.1.1 Traceroute tool

Traceroute is a simple and popular tool proposed and first implemented by Jacobson [77] at the Lawrence Berkeley National Laboratory in 1989. Traceroute returns the path taken by packets sent to a particular IP address. It does so by using the *time-to-live* (TTL) field of an IPv4 header or the *hop limit* equivalent of an IPv6 header. These fields are meant to prevent routing loops which would otherwise lead to infinite forwarding of packets, potentially reducing the network to a crawl. When a packet is sent, the TTL field is given an initial value which is decreased by one every time it is forwarded by a router and eventually dropped when its value reaches zero. This ensures that the packet will be dropped once the maximum number of hops is reached. The router that discards the packet returns an ICMP *time exceeded* message to the packet source, thus unveiling its IP address.

The traceroute process starts by sending a packet to a given destination with a TTL equal to 1. The first router reached by the packet will then discard it and reply by a *time exceeded* packet showing in its source field one of the router's IP addresses. This IP address corresponds to the interface on which the reply packet was sent, most probably the interface through which the traceroute source address is routed. An IP address of the first router is thus known, but not necessarily the destination address of the probe packets. Traceroute then sends a packet with a TTL of 2 to the same destination, discovers an IP address of the second router, and so on. The probing ends when the maximum number of hops is reached or when a reply indicates the destination has been reached. An example of a traceroute output is given in Figure 3.1. Its output shows up as a list of IP addresses belonging to routers which responded to the probe packets at each TTL with their response times. Note that this list is not necessarily the path taken by all (or, indeed, any one of) the packets, because each probe packet is routed independently and routing changes may occur while traceroute is running.
Several features have since then been added and modified to measure more properties [78, 79, 80]. TCPTraceroute [81] is for example a traceroute implementation that uses TCP packets instead of UDP or ICMP packets to send its probes. It can be used in situations where a firewall blocks ICMP and UDP traffic. It is based on the "half-open scanning" technique that is used by NMAP, sending a TCP with the SYN flag set and waiting for a SYN/ACK (which indicates that something is listening on this port for connections). When it receives a response, the

```
1 kirby-FE4-13.cac.washington.edu (140.142.15.225) 0 ms 0 ms 0 ms
2 uwbr-ads-01-vl1998.cac.washington.edu (140.142.155.23) 1 ms 1 ms 0 ms
3 hnsp2-wes-ge-0-0-0-0.pnw-gigapop.net (209.124.176.12) 1 ms 1 ms 1 ms
4 abilene-pnw.pnw-gigapop.net (209.124.179.2) 1 ms 1 ms 1 ms
5 dnvrng-sttlng.abilene.ucaid.edu (198.32.8.50) 26 ms 26 ms 26 ms
6 kscyng-dnvrng.abilene.ucaid.edu (198.32.8.14) 40 ms 47 ms 41 ms
7 iplsng-kscyng.abilene.ucaid.edu (198.32.8.80) 47 ms 47 ms 46 ms
8 chinng-iplsng.abilene.ucaid.edu (198.32.8.76) 50 ms 50 ms 50 ms
9 nycmng-chinng.abilene.ucaid.edu (198.32.8.83) 74 ms 73 ms 69 ms
10 198.32.11.51 (198.32.11.51) 70 ms 70 ms 70 ms
11 so-7-0-0.rt1.ams.nl.geant2.net (62.40.112.133) 153 ms 153 ms 153 ms
12 belnet-gw.rt1.ams.nl.geant2.net (62.40.124.162) 156 ms 157 ms 160 ms
13 oc192.m160.core.science.belnet.net (193.191.1.1) 157 ms 156 ms 157 ms
14 oc48.m20.access.lln.belnet.net (193.191.1.198) 157 ms 157 ms 157 ms
15 ucl-1.customer.lln.belnet.net (193.191.11.10) 157 ms 157 ms 157 ms
16 CsPythagore.sri.ucl.ac.be (130.104.254.238) 158 ms 164 ms 158 ms
17 CsHalles.sri.ucl.ac.be (130.104.254.201) 158 ms 158 ms 158 ms
```

Table 3.1: Traceroute from the University of Washington to the Catholic University of Louvain

TCPTraceroute program sends a packet with a RST flag to close the connection.

Another traceroute derivative is the NANOG[82] traceroute is derived from the original traceroute program, but adds a few features such as AS (Autonomous System) number lookup, and detection of ToS (Type-of-Service) changes along the path.

Traceroute explorations may be performed using packets other than ICMP echo requests, such as UDP packets. In this case, a UDP packet is sent to a destination host at a specific destination port. If this destination port is unused, the host will reply with an ICMP *port unreachable* message whose source address is once again the address of one of the host's interfaces. An important implication of this behaviour is that it allows the probing host to infer that the return message's source and the traceroute destination belong to the same router.

### 3.1.2 Existing discovery systems

Initial topology discovery systems, such as Mercator [54], based on hop-limited probing methods sent probes out from a single location. Probing in such a way is likely to result in a tree-like topology and will tend to miss out "cross-links", i.e. traversal branches. Subsequent work therefore adopted multiple source approaches, in which multiple probing sources are used.

Quite like looking glasses, a large number of public traceroute servers have been made available and can be found at [73]. Initially their use was mainly intended for debugging purposes but they can also be used by topology discovery tools, although their web interfaces sometimes lack convenience. An example of such a platform is PlanetLab [83] which has currently 694 machines hosted on 335 different sites which provides a network testbed for many active research projects of all kinds, covering file sharing, content distribution networks, QoS overlays, anomaly detection mechanisms, and network measurement tools. An implementation making use of PlanetLab is Scriptroute [56] aimed at providing traceroute servers' accessibility but with the flexibility of running a variety of measurement tools.

Other approaches have deployed their own measurement infrastructure, like the well-known Skitter [12] project of CAIDA which has between 20 and 30 available monitors[1] world-wide providing traffic measurements services for researchers. The Active Measurement Project (AMP) [84] of the National Laboratory for Applied Network Research (NLANR), although recently terminated in July 2006, took a slightly different approach by letting 150 monitors, mainly deployed in the United States, probe each other in a full mesh in order to obtain dense coverage of the underlying network. Another project, Rocketfuel [58], concentrates on discovering the internal topologies of ISPs.

A more recent trend has been to move towards a more distributed system as demonstrated by a project like DIMES [85] from the Tel-Aviv University. DIMES relies on the distribution of small portable agents to the community, in a similar spirit to ones such as SETI@home, which perform typical traceroute and ping probes around the globe.

[1] A "monitor" is the common name for an active probing node as opposed to a passive destination node

### Mercator

Mercator was introduced by Govindan and Tangmunarunkit in [54]. It explores the network from a single location by using hop-limited probes to determine paths and infer a topology. Although a single source may seem restrictive, they use source routing as a means to increase discovery coverage. Once a router allowing source routing is found, which was surprisingly the case for 8% of their probed routers (i.e. nearly 10.000 routers), it provides the capacity to probe the network from another location, thus providing new "virtual sources".

Mercator implemented a heuristic approach named *informed random address probing* which explores the IP address space more efficiently than exhaustive probing and without requiring input. It starts off from a seed prefix and then probes randomly chosen prefixes adjacent to previously encountered prefixes which are maintained in a list by the probing host. This assumes Internet registries sequentially allocate address spaces. A primitive approach for determining prefix lengths was hazardously performed according to primitive IP address classes [47].

Another point is that Mercator additionally conducts a check before probing the path to a new address of a given prefix: if paths to an address of the same prefix are known, Mercator starts probing at the highest hop count for a responding router seen on those paths. Despite this heuristic's proposal, no results were published about its performance.

### Skitter

CAIDA's Skitter [12] is a topology discovery system that performs hop-limited probes from a certain number of large servers strategically placed in the Internet, some of which are DNS root servers. Similar alias resolution techniques to Mercator's were implemented in their *iffinder* tool developed by Ken Keys, but probed addresses are not chosen randomly anymore. Instead, probed addresses are specified by a predefined list generated using several data sources, e.g. web server and DNS root server client lists, and by selecting one IP address for every prefix announced in BGP.

The predefined list is constantly updated and periodically probed: Skitter makes results available to researchers at each end of probing cycle being the time to finish probing its destination list. Skitter data has been used for many purposes since 1998, e.g. visualizing network AS-level network connectivity, evaluating the quality of data provided by traceroutes [86].

### Rocketfuel

Rocketfuel [58] is a tool developed by Neil Spring et al. which aims at building a detailed map of an individual ISP's topology by employing some 800 traceroute servers to gather as much topological information possible with the minimal amount of measurement and without relying on confidential information. They use several heuristics for identifying router aliases, including DNS naming, time-to-live, IP identification field, and instances of rate limiting triggered by earlier probes.

Unnecessary traceroutes that are likely to follow redundant paths through the ISP network are suppressed by using BGP routing information. By selecting only traceroutes likely to transit the network, this heuristic is able to reduce the number of measurements by three orders of magnitude compared to an all-to-all approach. The IP identification field is used as a pioneering method for alias resolution, as explained in Section 2.1.3. Geographical information from DNS naming is used to divide the router-level topology into POP-level backbone and access components.

Results were validated by authors by asking ten different ISPs to evaluate the discovered topologies and to compare them to previously discovered ones by systems such as RouteViews and Skitter. Their detailed analysis resulted in the discovery of seven times more links than previously inferred maps.

### DIMES

DIMES is a ongoing distributed scientific research project launched in September 2004, aimed to study the structure and topology of the Internet, with the help of a volunteer community. The DIMES agent performs Internet measurements such as traceroute and ping at a low rate, consuming at peak 1KB/S. Since then more than 10000 agents have been registered at their website (www.netdimes.org) and the most current result analysis available was performed on 2005 data [87] when the system was composed of 5000 agents located in over 570 different ASs.

The results lead to a map of about 61000 AS edges connecting over 15000 AS nodes. Out of these edges, almost half are not present in BGP table repositories such as RouteViews, thus making the Internet 50% denser than previously observed, although unfortunately no comparison with Skitter was performed. In addition, the project is working on means to identify non-responding hosts not identifiable by traceroute-like measurements, although they are known to exist.

### 3.1.3 Limitations and issues

Although active probing methods have key advantages, they have their own share of limitations due to inherent inaccuracies of hop-limited probes. Implementation and network load issues are also prone to arise, particularly when a distributed infrastructure is needed.

#### Speed

Classic traceroutes are slow as they have to progressively probe through each hop in order to reach a given destination. *Backward probing* which consists in probing with a maximum hop and gradually working its way back to the source was proposed by [54]. More recent Moors' [88] evaluates a method for sending a *scout* packet to the destination and examining its response packet's TTL in order to guess the original TTL (typically one of a few standard values). These methods have been shown to work well, despite not working when a destination does not reply, which is more common than an isolated mishap.

#### Inherent inaccuracies

Even though hop-limited probes are a very useful tool, they have their drawbacks. One immediate limitation is the fact these probes only discover forward paths towards a given destination, although reverse paths may differ because prefix-based routing policies and hot-potato routing can cause asymmetry. Asymmetric routing also stops inference being made on the link delay between two consecutive hops from the reported round-trip times: the difference between the RTTs could be due to the link, or to one of the routers using a different return path. A way to partially circumvent this issue is to perform a mesh measurement as carried out by AMP [84].

As evaluated by Teixera et al. [89], a second limitation is that active probes can potentially follow several paths in the event of failure or due to load balancing amongst equal paths, and that such path diversity especially appears in the core levels of the network. As a result, probes tend to follow primary paths and thus miss out on backup paths if an insufficient number of probes or undersized time-window is provided.

A third limitation is that the path reported by a tool like traceroute is potentially a non-existing path for a packet to traverse. Although unlikely to be a completely false path, load-balancing and route changes can cause spurious links to appear in the resulting output. This is a consequence of the non-atomic way paths are determined and a danger of many such discovery tools is to pretend to be atomic when they are not. Moreover, when in this context we talk of links, we are not talking of IP links between two end-point IP interfaces; traceroute only reports

the address of one of the interfaces of a given link, the one belonging further away from the source. This weakness can be partially addressed by using a ping with the record route option set, but it is unable to provide information about routers more than 8 hops away.

A fourth problem can occur inside an AS internally using MPLS (see Section 2.2.2) which gives it the ability to hide the underlying topology by disabling the TTL used by traceroute. Routers using MPLS may be configured either to decrement the TTL, as traceroute requires, or to ignore the TTL field: because the switched paths of MPLS are configured to have no loops, the IP TTL is not needed. The MPLS specification, however, recommends that the TTL be decremented where possible [90].

Lastly, active measurements are at the mercy of inconsistent behaviour from networked elements: misconfigurations sometimes lead to private non-routable addresses appearing in intermediaries, non-RFC complying implementations cause different interfaces to respond depending on their router's vendor, and strict debug policies or firewalls prevent some probed routers from responding.

**Spatial bias**

A natural question that arises is how many measurement points are necessary in order to obtain good coverage of the Internet, or at least its backbone. The problem was addressed by Barford et al. [91] who studied the marginal utility of probe sources by examining Skitter traceroutes. They show that indeed by adding more vantage points, new links are revealed, and that the marginal utility of adding more sources rapidly decreases. On the other hand, the marginal utility of adding destinations is much higher, with an almost linear increase of the number of discovered nodes with the number of added destinations.

Another point is that these findings show that, for instance, in presence of a dozen vantage points, although there is only a small advantage in adding a few more, there is still a significant advantage to add thousands of points as they will add a significant percentage of new links. Furthermore, using a small number of such points gives a strong bias in the topology towards customer-provider links while missing many peer-to-peer links.

This issue is actually part of a broader one, relevant to the Faloutsos brothers' findings [24] in 1999 which showed that the degree distribution of nodes in the Internet followed a heavy-tailed[2] distribution. They used AS- and router-level Internet topologies to show that power laws can be used to characterize this node degree distribution. Broido and Claffy [60] further went on

[2] A heavy-tailed distribution is one where high-degree nodes appear with a higher probability than would be expected in a standard random graph

to show that Weibull distribution can be used to approximate the outdegree distribution of the routers. While some analysed the difficulties in sampling methodologies [92] and others questioned the validity of using degree distribution as the main metric for characterizing the Internet topology [93], Lakhina et al. pointed out how such sampling biases be caused by the use of a small number of sources [13]. Indeed, they show empirically how the subgraph formed by a collection of shortest-paths from a small set of random sources to a larger set of random sources can easily appear to show a degree distribution remarkably like a power-law. Recent results from Clauset and Moors [94, 14] confirm the sampling bias after having systematically studied it for a very general class of underlying degree distributions.

**Network load**

Consequently, these seemingly biased observations are making the case for a highly distributed measurement system, but not at any cost. Actual Internet topology knowledge could highly benefit from such an infrastructure with thousands of measurement points spread around the globe, but if carelessly deployed, could impose an unnecessary load on network ressources.

Donnet et al. [17] analysed this issue and distinguished two types of redundant measurements carried out by a multiple source system using traceroute-like probing mechanisms.

- **Intra-monitor redundancy** is redundant measurements made by an individual monitor, i.e. probe source, and appears near each monitor as probes will tend to cover close interfaces several times. In a large-scale system, the degree of such redundancy could be very high as a nearby interface could be visited for each probed destination.
- **Inter-monitor redundancy** is redundant work carried out by different monitors and takes place closer to destinations. Such redundancy can also potentially be quite large as it would be expected to grow linearly with the number of monitors.

Although some attention in previous work had been paid to minimize the number of traceroutes performed, many current discovery systems naively carry out traditional active probing or try to maintain reasonable load on ressources by reducing their tools' performance. In addition to imposing additional load on internal nodes and links of the network, redundant measurements can also appear to destination nodes as DDoS attacks since probes would be received from a large number of different sources.

### 3.1.4 Summary evaluation of current active probing systems

The discovery of network topology actively using probe packets has many advantages. Not only does it provide up-to-date and relatively reliable information, it is the only method for discovering interface- and router-level topologies of the global Internet. The discovered topologies may not be complete, but this is also the case with other methods described in the previous chapter.

The main drawback of active probing is the necessary time taken to obtain a snapshot of the topology: on one hand, the accuracy and flexibility of topology discovery using probe packets makes it extremely useful for targeted explorations such as those performed by Rocketfuel. On the other, mapping a large network like the Internet requires several days or weeks of probing, during which significant network changes can take place and seriously affect results. Employing measurement systems that use a larger number of monitors can help this issue, as well as reduce observed sampling biases. Indeed, while more monitors probe the same space, each one can take a smaller portion and probe it more frequently. Network dynamics such as routing changes that might be missed by smaller systems could be more readily captured by the larger ones while keeping the workload per monitor constant.

Another difficulty with active probing is the inconvenience of having to deploy a measurement infrastructure or interact with public traceroute servers. This issue has however been addressed by DIMES which has managed to deploy thousands of light-weight measurement agents and which offers an insight into future measurement systems.

In the meantime, issues of deploying a large-scale network-friendly measurement systems have been partly addressed by Donnet et al. [17]. Following their observations about redundant measurements in a typical active probing system, they proposed an elegant solution described hereafter, i.e. the *Doubletree* algorithm, which attempts to minimize network usage and avoid overloading destinations while keeping a high level of node and link coverage. Unfortunately, this solution requires collaboration between monitors at the cost of a communication overhead.

## 3.2 Doubletree

The *Doubletree* algorithm [17], which is part of the Traceroute@home [95] project, is the first attempt to efficiently perform large scale topology discovery in a network-friendly manner through co-operation between monitors. Methods to reduce communication overhead caused by this approach have also been proposed by the same authors in [96] and in [97] through the use of Bloom filters [98] and CIDR address prefixing [99]. Heuristics, for further increasing coverage are under proposal [100, 101] as well as effectiveness by creating clusters of monitors and capping the number of times a destination gets probed, are a matter of discussion as they are still being investigated.

### 3.2.1 Discovery process using trees

A tree can be used by a probing algorithm to keep track of its discovery progress. The rule is that probing is carried out from the leaves to the root, i.e. decreasing probe packet TTLs, as long as it is probing a previously unknown part of the tree. It then stops when a previously discovered node is encountered. The assumption is made that the remaining path to the root is known, leading to potential coverage loss. Note that, in the context of Internet topology, discovering a new node would correspond to a router's interface responding to a probe packet within the constraints previously described in Section 3.1.3.

Doubletree tackles both types of redundancy described in Section 3.1.3 and is based on the tree-like structure of routes emanating from a single source or converging on a same destination. For this reason, Doubletree uses two trees, one *monitor-rooted* and one *destination-rooted* trees, as illustrated by Figure 3.1. The monitor-rooted tree is composed of outgoing routes from a single monitor to multiple destinations. The destination-rooted tree is composed of routes from multiple monitors converging to a common destination.

The stopping rule based on the former aims at reducing intra-monitor redundancy, and the latter which requires inter-monitor communication in order to reduce their shared redundancy.

### 3.2.2 Two-phase probing

Doubletree acts in two phases:

1. Forward probing proceeds from an initial hop count h to h+1, h+2, and so forth, applying a stopping rule based on the destination-rooted tree.

2. Backward probing then follows by taking the hop count back to h-1, h-2, etc., applying a stopping rule based on the monitor-rooted tree.

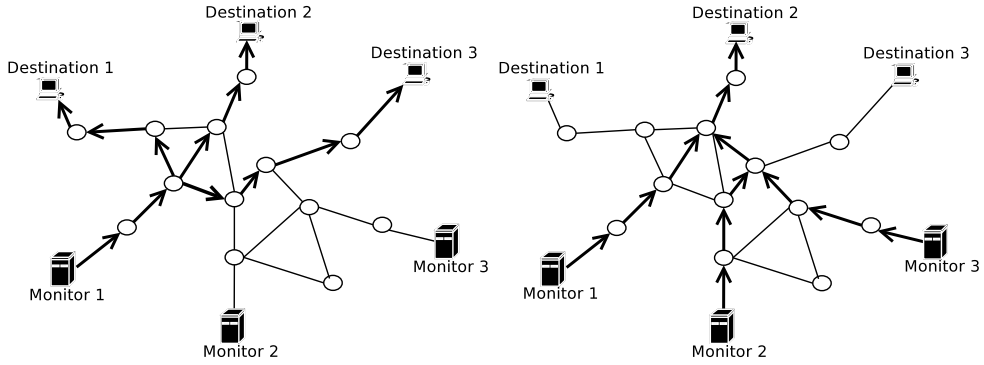


Figure 3.1: Monitor-rooted and destination-rooted trees from [17]

During the backward probing phase, once the root or stopping rule is reached, the algorithm moves on to the next destination and restarts probing at initial distance h. In the special case where there is no response at a certain distance, this distance is halved, and halved again until there is a reply. Probing then continues forwards and backwards from that point.

### 3.2.3 Stop sets and simple stopping rule

Rather than maintaining information about the tree structure, it is sufficient for the stopping rules to store sets of *(interface,root)* pairs, the *root* being the root of the tree in question, i.e. the monitor- or destination-based tree. These sets are called *stop sets*. A single monitor uses two distinct stop sets:

- The first stop set is used when probing forwards and is called the *global stop set*.
- The second stop set is used when probing backwards and is called the *local stop set*. This set can be reduced to a list of interfaces since the root item never changes as it is the monitor itself.

The stopping rule for each phase is simple: it is to stop when the encountered pair is already a member of the relevant set, otherwise it is added.

### 3.2.4 The Doubletree algorithm

Algorithm 1 is the formal definition of Doubletree and assumes the existence of two functions:

- *response()* returns true if an interface replies to at least one probe sent to it.
- *halt()* returns true if probing must be stopped for various reasons. These reasons include the detection of a loop or the observation of a gap in the case of a non-responding interface.

**Algorithm 1** Doubletree

**Require:** F, the global stop set received by this monitor.

**Ensure:** F updated with all (interface,destination) pairs discovered by this monitor.

```
1:  procedure DOUBLETREE(h, D)
2:      B ← ∅                                          ▷ local stop set
3:      for all d ∈ D do                               ▷ destinations
4:          h ← ADAPTHVALUE(h)                         ▷ initial hop
5:          TRACEFORWARDS(h, d)
6:          TRACEBACKWARDS(h − 1, d)
7:      end for
8:  end procedure

9:  procedure ADAPTHVALUE(h)
10:     while ¬response(v_h) ∧ h ≠ 1 do                ▷ v_h the interface discovered at h hops
11:         h ← h/2                                    ▷ h an integer
12:     end while
13:     return h
14: end procedure

15: procedure TRACEFORWARDS(i,d)
16:     while v_i ≠ d ∧ (v_i, d) ∉ F ∧ ¬halt() do
17:         F ← F ∪ (v_i, d)
18:         i + +
19:     end while
20: end procedure

21: procedure TRACEBACKWARDS(i,d)
22:     while i ≥ 1 ∧ v_i ∉ B ∧ ¬halt() do
23:         B ← B ∪ v_i
24:         F ← F ∪ (v_i, d)
25:         i − −
26:     end while
27: end procedure
```

### 3.2.5 Choice of initial distance

In order for a monitor to avoid excess intra-monitor redundancy by probing too close and excess inter-monitor redundancy by probing too far, Doubletree starts off at what is hoped to be an intermediate point $h$ between the monitor and the given destination. Each monitor having a different location in the network, a reasonable value for $h$ is to be determined for each one.

A choice for $h$ is typically based on the distribution of path lengths as seen individually from the perspective of each monitor. One easily estimated parameter by a monitor is its probability $p$ of hitting a responding destination on the first probe. By fixing p, the individually obtained values of $h$ correspond to a similar level of incursion of each monitor in the network.

Note that choosing an initial distance can only be done for responding destinations and can thus use quicker ways to evaluate this initial distance is by using previously described solutions such as Moors' [88]. This is also the technique mentioned by a study concentrating on single source route tracing [102].

### 3.2.6 Inter-monitor communication load and Bloom filters

Keeping basic knowledge about previously discovered nodes in the network does not require sharing their whole sets of (interface, destination) pairs. *Bloom filters* [98] are bit vectors that can be used to encode such membership. An empty Bloom filter is a vector of all zeros. A key is registered in the filter by hashing it to a position in the vector and setting the bit at that position to one. Multiple hash functions may be used, setting several bits to one. Membership is then verified if all hash positions are set to one.

A Bloom filter will never falsely return a negative result for set membership, but might return a false positive. The size of the filter and number of hash functions can influence its performance. The risk in using them is thus limited to halting a forward probing scheme earlier than expected, i.e. a positive membership answer for a yet undiscovered interface would comply with the stopping rule.

This data structure was proposed by Doubletree papers [96] to share the global stop set using Bloom filters as they would otherwise represents a significant communication overhead. Variations exist such as *Retouched Bloom Filters* (RBF) [103] which is currently under review.

### 3.2.7 Advanced stopping rule using CIDR prefixes

In order to further reduce the load on destinations and communication overhead while maintaining an acceptable level of accuracy, another study of Donnet et al. [104] proposed the use of stopping based on CIDR prefixes [99]. Instead of storing the full IP address of destinations, the idea is to agglomerate individual destination IP addresses into subnetworks with the use of CIDR address prefixes, thus storing *(interface, prefix)* pairs in the global set instead of *(interface,destination)* pairs.

Tests were performed using all prefix lengths from /8 to /24, as well as /28 and /32 (full IPv4 addresses). Results show a gradual improvement in performance, as expected, from the most generic prefixes to the most specific. However, an interesting observation was that both node and link coverage stabilized around prefix /24 reaching nearly the same accuracy as classic Doubletree. The loss of accuracy seems to be mostly located in the subnetworks containing the destinations and in the core network where duplicate links (and associated nodes) are missed due to the prefix-based rule.

### 3.2.8 Capping and clustering

Additional ways of limiting the risk of DDoS attacks appearing at destinations were also proposed in [96] which explains how Doubletree's performance could perhaps be improved by additional techniques such as:

1. limiting the number of monitors per destination (i.e. *capping*)
2. establishing clusters of monitors within which all have a common destination set, with no overlapping destinations (i.e. *clustering*)

Capping is easily implemented by limiting the number of destination sets in which each destination appears, assuring an upper-bound on the potential number of monitors hitting end nodes. However optimal deployment of clusters remains obscure as many questions remain open, e.g. how many clusters should be created, how to assign them their sources and destinations.

An original idea, shared by Doubletree's author, would be to materialize the concept of *proximity* between monitors, in other words find a means to quantify the difference between their views of the network. By somehow comparing their individually discovered topologies, e.g. measuring the *Hamming distance*[3] between their local stop sets represented as Bloom filters, a new metric could be defined opening up doors to the application of known clustering algorithms such as K-means, PCA, Gauss-mixture, self-organizing maps or binary vector quantizers. An objective

[3]The Hamming distance between two vectors of equal size is the number of bits in which they differ

function can then be specified such as minimizing the similarity of monitors belonging to a same cluster. Similar to evaluating the initial hop count of Doubletree algorithm, this would imply a learning period during which each monitor would perform initial probing before sharing its local stop set with a central coordinator responsible of determining appropriate clusters.

### 3.2.9 Evaluation

Doubletree authors' showed how to deploy the algorithm in reality [105] as additional studies have shown that the algorithm was deployable in reality [106]. Note that these results appeared after the thesis' aims had been set out. Performance evaluation difference between multiple source traceroute and doubletree will therefore be summary in order to focus on discovery heuristics.

# Chapter 4

# Design and implementation

*This chapter covers the design choices leading to the implementation of the active discovery simulation framework based on an appropriate network simulator. Simplifying hypotheses due to its use are evaluated in regard to the behaviour of a real-life network. Requirements are then set out and the resulting implementation is described per module.*

## 4.1 Design choices

A large-scale Internet topology discovery infrastructure requires thorough testing and performance evaluation before being deployed, otherwise one risks network overload and poor results. Being able to realistically model the behaviour of a network is a difficult task as many aspects described in the previous chapters have to be covered. The problem has typically been separated in two issues: on one hand, network simulation has to capture the complexity of protocols used at the different levels of the Internet, e.g. inter- and intra-domain routing. On the other hand, fictive yet representative network topologies must also be provided to simulators in order to obtain a functional model of real-world networks such as the Internet. The choice of network topologies is a topic covered in the next chapter as it is not an implementation-specific matter.

### 4.1.1 Choice of a network simulation framework

Existing network simulation frameworks have been an ongoing research subject for years, giving what can be mostly considered as full-protocol network emulators. Examples of which are: BGP++ [107] which is a port of the Zebra BGP daemon [108] onto the popular NS-2 simulator [109], SSFNET [110] or J-Sim [111]. Unfortunately the level of detail is a burden for large topology simulations from a time and ressource consumption point of view. More recently, an open-source C-BGP [18] solver tool provides an implementation of a complete BGP model while not being hindered by the transmission of BGP messages over simulated TCP connections.

### C-BGP overview

Although the main use of this tool is currently to study interdomain traffic engineering techniques, its ability to fully and efficiently support most notably IGP, eBGP and iBGP sessions and the full BGP decision process, makes it the best contender for studying a router-level discovery process influenced by high-level AS policies.

C-BGP takes a description of the network topology at IP layer 3, thus describing all present routers, links connecting them and their configuration. This configuration includes the assignment of IGP weight to all links, the BGP peerings of each router and the BGP policies enforced on each peering. The solver then takes the BGP routes learned by all border routers and for each router, i.e. border and internal, computes the routes towards all interdomain announced prefixes. In order to facilitate the setup of large simulations, C-BGP can also load AS-level interdomain topologies produced by the University of Berkeley.

C-BGP is written in the C programming language and is available on different platforms, from which Linux has been chosen. Interaction with a C-BGP instance is performed through a CISCO-like command-line interface. Several interfaces to programming languages are provided for ease of use, i.e. Java, Python and Perl. The choice of Perl is made for interacting with the C-BGP instance. A concrete example of a typical interaction is detailed in Appendix A.

### Simplification hypotheses

Simulation implies making simplification hypotheses, as all the features and sometimes erratic behaviour of Internet components cannot be efficiently reproduced. The following elements have been identified:

- **Alias resolution**: in the C-BGP version used, routers are assimilated to a single node and interface, i.e. one IP address. The problem of different interfaces responding to a traceroute probe is therefore non-existent.
- **Responsive routers**: all reachable routers in C-BGP respond to probing as opposed to routers in the Internet which may not do so for various reasons, e.g. unexpected link failures or packet filtering. In addition this makes path length evaluation exact when initially computing the initial hop value for Doubletree.
- **Coherent configurations**: a properly engineered topology avoids router misconfigurations which can be observed in the real world, however, this does not prevent another study from using C-BGP to, for example, analyze their impact.

- **Stable routing state**: propagation of BGP messages across routers in C-BGP is done in a static way until a stable routing state is reached, if it exists and is reachable[1], only then are simulations performed eliminating the appearance of routing anomalies such as spurious links. All discovered paths in a probing process are therefore existing ones in the topology.

#### Simulation benefits

Apart from being dispensed of unexpected behaviour, there are other benefits to performing discovery processes in a simulated environment:

- **Omniscience**: complete knowledge of the interdomain topology and policies, as well as their often unshared internal router topologies are available for deciding discovery attributes, such as the placement of monitors in the network. After the discovery process, exact knowledge of its performance can also be obtained, for instance, concerning node and edge coverage or network load.
- **Determinism**: as opposed to previously cited discrete-event[2] simulators, the message propagation model of C-BGP relies on a single global linear queue which consequently leads to the same outcome for any run. Multiple discoveries can therefore be performed on exactly the same model, leaving the differences in their setup as the only reason for obtaining varying results.
- **Topology variation**: an identical discovery process need not only to be tested on a single model, it can be performed using different topologies which may have different properties.

### 4.1.2 Functional requirements

Here is a summary of the features required by the implementation:

- **Network model**: the implementation must allow efficient use of an internal model of the network. This model is based on a given AS-level topology, with an optional and coherent underlying router-level topology. Since AS-level routing policies have a significant impact on probe paths (see Section 2.1.1), they should also be specified to avoid less representative shortest-path routing.
- **Node and edge classification**: the model must allow the characterization of the network's nodes and edges, at both AS and router levels, to provide the means to appropriately place monitors and obtain detailed location-based results once the discovery process is over.

[1] BGP is indeed not assured of convergence, for further details please refer to Griffin and Wilfong's [112]
[2] The outcome of a discrete-event simulator may depend on a pseudo-random number generator (PRNG) seed

- **Source and destination siting**: this includes choosing the number of monitors and destinations according to their location in the network while optionally applying to them clustering methods and/or capping constraints.

- **Discovery execution**: the implementation must allow the choice between classic and Doubletree discovery, as well as tuning their parameters, e.g. hit probability and number of evaluation probes for determining initial hop, optional Bloom filter capacity and error rate.

- **Performance evaluation**: adequate information must be provided in order to evaluate the performance once a discovery has finished, such as detailed router-level node and link coverage, number of used probes, etc. Optional storage of the monitors' state could also be of interest for further analysis.

- **Modular implementation**: the preceding requirements should be, as far as possible, independently constructed for performance, validation and reusability purposes.

- **Debugging feature**: access to a debug option should also be proposed, for instance, to provide highly detailed information about the individual steps performed by an implemented probing algorithm.

## 4.2 Implementation

The preceding functional requirements naturally result in a modular implementation layout which divides itself in three executable Perl scripts:

1. The `classifyAS` script is responsible for classifying an AS-level topology according to the Subramanian hierarchy described in Section 2.1.1.

2. The `setupDiscovery` script is responsible for setting up a discovery process by generating an XML file containing clusters, monitors and their respective destinations.

3. The `performDiscovery` script finally performs the discovery in coordination with the C-BGP module provided on the tool's website and computes performance figures.

In addition, three support modules, i.e. `topology.pm`, `doubletree.pm` and `tools.pm`, respectively provide the scripts with access to the network topology model, discovery algorithms and generic conversion tools. Note that the source code of scripts and modules is available in Appendix E.

### 4.2.1 Discovery preparation

#### AS classification script

The role of the AS classification script is to classify a given AS-level topology and is invoked in the following way:

```
./classifyAS <as_topology_file>
```

This script takes as input an AS topology file containing `[as1 as2 relationship]` entries with relationship values $-1$, 0, 1 respectively standing for customer-provider, peering and provider-customer relationships. A directed graph is created according to Subramanian's method: a provider-customer edge is represented by adding single directed edge from the first AS to the second and conversely for a customer-provider edge. A peering relationship is represented by two edges of opposite direction.

Note that sibling relationships are not included in this model. Indeed they are non-trivial to accomodate in Subramanian's framework because they not only lack the directionality of customer-provider relationships, but also do not have the export policy constraints of peers. However this should not significantly affect the overall accuracy of the model since sibling relationships are uncommon compared to other peering relationships.

The Subramanian classification of ASs then takes place as follows:

- A first pass labels the leaves[3] of the graph as **customer ASs** and removes them.
- A second pass applies a reverse pruning mechanism to recursively remove remaining leaves until none are left and label them as **small ISPs** which are ASs providing one or more customers.
- A third pass greedily classifies the **dense core** of the network by relaxing the definition of a clique and identifying instead a subgraph of $N$ nodes that each have an in-degree and out-degree of at least $N/2$.
- A fourth pass greedily identifies the **transit core** defined as the smallest set of ASs containing the dense core which induces a *weak* in-way *cut*, that is, one having a small number of edges compared to the total number of ASs in the transit core.
- A final pass labels remaining ASs as part of the **outer core**.

[3] In a directed graph, a leaf is a node with out-degree 0. However in an undirected graph, a multi-homed customer would not be considered a leaf as distinction between in-degree and out-degree could not be made.

The script finishes by outputting the classification to `<as_topology_file>.classification` in the following format `[as_num level]` with level values from 0 to 4 respectively corresponding to the dense core, transit core, outer core, small ISP and customer classifications.

### Topology model module

This module provides both the setup and simulation modules with subroutines for configuring the topology model, i.e. its AS-level and router-level graphs.

The first subroutine `init_asgraph` has the role of reconstructing the classified AS graph previously computed with the AS classification module. It does so by extracting node and link information from the AS topology file, similarily to the previous module, and by labeling nodes using the classification file. Furthermore, external modules can request the display of the AS topology's per-level distribution and inter-level connectivity. The following examples are based on the AS-level topology further described in Section 5.1.4. Table 4.1 summerizes the number of ASs at each level in the hierarchy.

| **dense** | **transit** | **outer** | **ISPs** | **customers** |
|---|---|---|---|---|
| 22 | 155 | 1336 | 1513 | 13921 |

Table 4.1: Per-level node distribution of an AS topology

Table 4.2 summarizes the connectivity between various levels in the AS hierarchy in what we call the *edge connectivity matrix* of the AS topology. Each number in the table is the total of *edges* from one level to another. In the inter-level connectivity table must be read from left to right. For instance, dense core ASs have 208 peering edges *towards*[4] transit ASs and 683 provider-customer edges towards transit ASs. Note that the underlined peering numbers are symetrically sited around the diagonal figures. Indeed, one peering edge is assigned to each AS since, in the AS graph, a peering relationship is represented by a double edge. As a further consequence, provider-customer values also correspond to the number of *relationships* between levels, whereas diagonal peering values must be halved in order to obtain the same interpretation. Note that intermediate peering totals are not interpretable in such a way.

A second subroutine `init_rgraph` can then be invoked in order to initialize an *undirected* router-level graph. If the optional router-level topology, i.e. a C-BGP script as illustrated in Appendix A, is specified, the module imports the specified routers and links and adds them to the graph.

[4] As opposed to "with"

| level | dense | transit | outer | ISPs | customers | total (PP,PC) |
|---|---|---|---|---|---|---|
| **dense** | 356,2 | 208,683 | 1,1774 | 0,1274 | 0,9071 | 565,12804 |
| **transit** | 208,4 | 726,647 | 76,2324 | 0,1118 | 0,6377 | 1010,10470 |
| **outer** | 1,0 | 76,11 | 1412,1137 | 0,796 | 0,5927 | 1489,7871 |
| **ISPs** | 0,0 | 0,0 | 0,0 | 0,438 | 0,4546 | 0,4984 |
| **customers** | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 |
| **total (PP,PC)** | 565,6 | 1010,1341 | 1489,5235 | 0,3626 | 0,25921 | 3064,36129 |

Table 4.2: Inter-level peering and provider-customer edge connectivity matrix of the AS topology

If no router-level topology is specified, the module constructs the router-level graph by adding a unique router for each AS present in the AS topology and by assigning it, according to a C-BGP convention, the IP address equal to the domain's number multiplied by 65536[5]. Note that in the model, an interface[6] is identified by its 32-bit integer representation of its IP address. Links, on the other hand, use a 64-bit big integer representation of their end IP addresses. Conversion tools between the various formats are provided by the `tools.pm` module.

The identification of intra-AS links is then performed as well as the classification of border and internal nodes; this is performed on-the-fly as it is a far less computation-intensive task than the AS classification. The router-level topology's per-level and per-type[7] distribution and inter-level connectivity matrix can both be displayed in a similar way to the AS topology connectivity matrix.

Both tables differ in the fact they additionally distinguish internal links besides peering and provider-customer "links". Note that the hypothesis is postulated that such a characterized "link" between two routers is inherited from the relationship between their respective ASs. As a consequence of the router-level topology's undirectiveness as opposed to the AS topology, a peering "relationship" between routers is represented by a *single* edge/link in the router-level graph. Peering values may thus appear halved on either side of the diagonal in order to maintain easily interpretable the diagonal figures, and most importantly the bottom-right total. The interpretation of inter-level peering links must therefore be made with caution.

[5]This correponds to a 16-bit left-shift of the 16-bit AS number. For example, the unique router representing AS3 would be given the IP address 0.3.0.0

[6]In the context of these simulations, "interface" and "router" terms are interchangeable

[7]The terms *levels* and *types* respectively correspond to the node classification at the AS-level (dense, transit, outer, ISP and customer) and at the router-level (border and internal)

### Discovery setup script

The role of the discovery setup script is to generate a configuration of sources and destinations by taking into account optional capping constraints and/or clustering assignments.

```
./setupDiscovery [options] <as_topology_file>
```

The main options that can be specified at the script call are:

- Router-level topology (-$r$) file
- Number of sources (-$s$) per type and per level
- Number of destinations (-$d$) per type and per level
- Number of clusters (-$n$) in which to divide the sources
- Cap-limit (-$c$) specifying the number of sources probing the same destination

Note that in order to facilitate the passing of multiple-value arguments, the number of sources and destinations are implemented as "compulsory" options, i.e. the script will output an error if they are not specified. For complete lists of the scripts' available options, please refer to Appendix D.

The setup script starts by initializing the topology model, thus calling the two topology initialization subroutines of the topology module. It then randomly selects sources and destinations from the router-level topology according to the specified per-level and per-type values. The selected sources are stored in a container, selected destinations in another. The script then has four ways of associating them to generate a discovery setup:

1. **Independent destination set discovery** is the default setting if no clustering or capping is specified. The script randomly assigns destinations to the sources grouped inside a single cluster. A specific destination therefore only appears in the individual destination set of one source.
2. **Clustered discovery** occurs when only a number of clusters is specified. Sources are then randomly distributed into clusters and destinations are randomly assigned to the clusters' common destination sets, as sources within a same cluster share the cluster's common destination set. Note that these common destination sets are also independent, i.e. no destination appears in two different sets.

3. **Capped discovery** occurs when only a cap-limit $c$ is specified. In this case, each destination is randomly assigned to $c$ sources with all sources being assigned to a single cluster.

4. **Clustered and capped discovery** occurs when both a cap-limit $c$ and a number of clusters are specified. In this case, sources and destinations are both randomly assigned to clusters. Each destination is then randomly assigned to $c$ of its cluster's sources. Note that while sources are distributed into clusters, they still use their individual destination sets due to capping constraints.

Once the source and destination allocation process is over, the resulting data structure is converted into an XML representation which is printed at the standard output by default or optionally saved in a specified file using the *export* option. Note that its corresponding schema `discovery.xsd` has been defined and is available in Appendix C followed by an example.

### 4.2.2 Discovery simulation

#### Discovery simulation script

The role of the discovery simulation script is to initialize the topology model and C-BGP instance, import the previously generated discovery setup available as an XML file, execute the chosen discovery algorithm (classic traceroute or Doubletree) and finally provide post-execution information about its performance. The script is launched using the specification of the AS topology and the XML discovery files:

```
./performDiscovery [options] <as_topology_file> <xml_discovery_file>
```

The main options that can be specified at the script call are:

- Router-level topology (-$r$) file
- Disable Doubletree (-$t$) perform classic traceroute discovery execution
- Number of path length evaluations (-$h$) for Doubletree initial hop
- Doubletree probability (-$p$) of hitting a destination on the first probe
- Bloom filter capacity (-$c$)
- Bloom filter error rate (-$e$)
- Disable use of Bloom filter (-$n$)

The script starts off by initializing the topology model, as explained previously, and imports the XML discovery setup into a data structure representing the *global discovery system's state*. The script then sets up a C-BGP instance with the help of the `tools.pm` module which typically implements low-level subroutines used for tasks such as number format conversions, file information extraction, initializing, setting up and checking the state of the C-BGP instance. Once the instance is running and has been initialized with the specified AS-level or optional router-level topology, the script launches the execution of the classic traceroute or Doubletree algorithm to the `algorithms.pm` module. Once the discovery is over, the script is in charge of computing performance results as further described in Section 4.2.3.

**Discovery algorithms module**

The discovery algorithms' module contains the implementations of possible discovery schemes. The module has three main subroutines, `traceroute`, `compute_hops` and `doubletree`, to which are passed the global discovery state and the topology model.

The classical discovery `traceroute` subroutine cycles through the set of clusters present in the discovery hash. If a cluster has a common destination set, one prefix of /16 length[8] per destination is propagated into the simulated network and traceroute requests from all the cluster's sources are then sent to the C-BGP instance. If no common destination set is available, the subroutine first cycles on the sources and then on their individual destination sets.

If the C-BGP instance's response has a successful status, each address in the route, apart from the source address, is considered to have successfully responded to a probe and has thus been discovered. In addition, each link leading to a responding interface is also considered to have been successfully discovered.

Note that the router topology graph is actually *vertex-counted* and *edge-counted* meaning the graph will count the number of times each node or link has been added to it. This can be used to efficiently keep track of not only discovered interfaces and links, but also of the number of times they were individually probed. This is why, once an interface successfully responds to a probe during a discovery scheme, it is added again to the topology model, as well as the link leading to it. Thus, once a discovery scheme is over, identifying nodes and links with a count larger than 1 yields the discovered interfaces and links.

The `compute_hops` subroutine relative to the Doubletree algorithm is in charge of evaluating

[8] If the destination router's IP address is A.B.C.D, the announced prefix will be A.B.0.0/16. If needed, please refer back to Section 2.1.2.

the initial hop for each source in the discovery setup. It does so by selecting a specified number of random destinations, from the source's available common or individual set. It then performs their traceroutes and uses the discovered path lengths to compute their cumulative distribution. Choosing the initial hop count to ensure that there is the probability $p$ of hitting a destination on the first probe is then straightforward.

The Doubletree subroutine `doubletree` has a similar structure to the classical one, cycling through clusters, then either through its common destination set if present, or else through its sources. Ideally individual interface probes should then be performed, unfortunately C-BGP does not provide this feature. Classic traceroute requests are thus performed and the difference lies in the way they are handled by the subroutine.

In order to perform forward and backward probing, the global discovery state contains one global set $F$ per cluster and their sources' local $B$ sets. By default these are respectively implemented as a Bloom filter and hashes, although global sets can also be stored as hashes if specified. Forward and backward schemes then take place by considering that an interface is responding at a specific hop count if it exists at that same index value in the response route. Note that, when an interface is not responding and its hop value is larger than 1, then it is halved. Within the context of the C-BGP environment, the only reason such a case might appear is if the current hop is beyond the current destination.

### 4.2.3 Discovery performance evaluation

Once the discovery algorithm's subroutine has finished, the discovery simulation script computes performance figures based on the resulting global discovery state. By default, it computes and outputs the following values:

- Global interface coverage: $\frac{|PI \cup S|}{|TI|} = \frac{|PI|+|S|-|PI \cap S|}{|TI|}$
- Global link coverage: $\frac{|PL|}{|TL|}$

such as $S$ is the set of sources used for the discovery process, $PI$ and $PL$ are respectively the sets of probed interfaces and probed links throughout the discovery, $TI$ and $TL$ are respectively the sets of all interfaces and links of the topology.

Note that in order to obtain a correct computation of interface coverage, the removal of the number of "discovered" sources, i.e. $|PI \cap S|$, is required before adding the known *a priori* number of sources.

In addition to these, detailed analysis provides the following values demonstrated as an example in Table 4.3:

- Number of probed interfaces per type and per level: $|PI \cap TI_{t,l}|$
- Number of interface probes per type and per level: $\sum_{i \in PI} probes_i$
- Interface coverage per type and per level: $\frac{|PI_{t,l} \cup S|}{|TI_{t,l}|} = \frac{|PI_{t,l}|+|S_{t,l}|-|PI_{t,l} \cap S_{t,l}|}{|TI_{t,l}|}$

such as $TI_{t,l}$ is the set of interfaces from level $l \in [0, 4]$ and of type $t \in [0, 1]$, and $probes_i$ the number of probes received by the interface $i$ during probing.

| level | **dense** | **transit** | **outer** | **ISPs** | **customers** |
|---|---|---|---|---|---|
| coverage | 0.909 | 0.774 | 0.315 | 0.382 | 0.358 |
| probed | 19 | 120 | 421 | 578 | 4991 |
| probes | 1309 | 2912 | 1816 | 1099 | 4991 |

Table 4.3: Per-level discovery distribution of router topology

Finally, the number of inter-level and intra-level *links* discovered is provided by another connectivity matrix as shown by the example from Table 4.4. Its layout is similar to the inter-level table previously described in Section 4.2.1.

| **level** | **dense** | **transit** | **outer** | **ISPs** | **customers** | **total (I,PP,PC)** |
|---|---|---|---|---|---|---|
| **dense** | 0,18,0 | 0,15.5,33 | 0,0,124 | 0,0,151 | 0,0,1642 | 0,33.5,1950 |
| **transit** | 0,15.5,0 | 0,8,50 | 0,0.5,202 | 0,0,188 | 0,0,1099 | 0,24,1539 |
| **outer** | 0,0,0 | 0,0.5,0 | 0,1,93 | 0,0,165 | 0,0,1283 | 0,1.5,1541 |
| **ISPs** | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,74 | 0,0,967 | 0,0,1041 |
| **customers** | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 | 0,0,0 |
| **total (I,PP,PC)** | 0,33.5,0 | 0,24,83 | 0,1.5,419 | 0,0,578 | 0,0,4991 | 0,59,6071 |

Table 4.4: Discovered internal and inter-level link connectivity matrix of the router topology

Note that currently, all these figures give an evaluation of the interface/router-level topology discovery. An evaluation of the AS-level discovery could however be implemented and is the matter of a discussion in Section **??**.

### C-BGP notes and modification

The first point is that C-BGP, performing a traceroute from a source router to a destination router only requires the announcement of a prefix originating at the destination router, propagating its route through the whole network. Although in reality a route back to the source has to be available for the return packets, in C-BGP, only the forward path needs to be known as this is the information returned by traceroute.

A second point is that C-BGP routers are fitted with data structures,i.e. a local and adjacent RIBs, which are used to respectively store the best BGP routes and the routes exchanged with neighbor routers. Several route announcements lead to the accumulation of information in these structures and is the main reason for the C-BGP simulator's memory consumption, especially when dealing with large topologies. It was observed that although these data structures are needed to compute the routers' routing tables required by traceroute, there is no further need to maintain them in memory once the forward path is returned. An additional *clear-ribs* command was therefore implemented in the C-BGP simulator and is called after each completed prefix propagation. The source code of this modification can be found in Appendix B.

# Chapter 5

# Test design and results

*This chapter elaborates on the objectives initially set out by the thesis and identifies ways to reach them using the now-available discovery simulation framework. A test plan is then accordingly laid out as well as the resulting choice of topologies on which to work. Finally, the test results are presented along with their interpretation.*

## 5.1 Test design

This part of the thesis has three main objectives: validate the previously described implementation, analyse both algorithms' (classic traceroute and Doubletree) performance in comparison with previously obtained results and optimise their performance benefiting from the points described in Section 4.1.1.

Although progressive testing has been carried throughout the development process, it is not sufficient. Validation of the implementation must be performed in order to ensure correct results. A formal approach being rather complicated to apply, the pragmatic approach of providing debugging features is provided to this regard and provides human-readable output of the many steps performed by a simulation. A detailed verification is thus performed on an example covering the most possible[1] input cases yet small enough to be hand-verifiable.

Once the validity of the process has been established, it could be interesting to see the results it yields using a large-scale topology, which hypothetically could be more "representative" of the Internet. These results could be used to provide a more detailed analysis of the discovery process than those available in the real-world, and thus enlighten previous work results or performance claims.

[1] Testing all potential topologies is not possible.

The final objective is to identify the means to optimise Internet topology discovery through the analysis of simulated results. These means including the development of heuristics, covered by Section 5.1.1, such as the placement of monitors and destinations according to their location in the network. A statistical approach to the problem is then described in Section 5.1.2 in an attempt to apply optimal experimental planning methods to it. A test plan is ultimately laid out according to the required features to be tested.

### 5.1.1 Identification of heuristics

Many heuristics have been elaborated and tested to improve the performance of router-level network topology discovery through the development of tools such as Mercator [54] and Skitter [12] which mainly concern the way an individual monitor independently probes the network. Other studies have proposed collaboration between such monitors, notably leading to the Doubletree algorithm [17] and its extensions, which may also be broadly considered as a form of heuristic. In practice, the performance of these heuristics can be relatively well evaluated according to the obtained results which is the main value sought to be maximized. However, heuristics regarding placement in the Internet are far more difficult to evaluate due to limited physical access reasons. Such heuristics can then realistically only be tested inside simulated environments. To our best knowledge, no study has yet put forward heuristics regarding the placement of active probing monitors and their destinations in the network.

Our network discovery simulation framework has a number of benefits, one of which is the opportunity to freely choose monitors and destinations amongst all potential nodes of a topology. The other is to know in advance the topology which is about to be discovered. This knowledge can partly help in guiding placement, for instance, with graph algorithms based on specific metrics. Such an example is Jamin's [113] which proposes heuristics and algorithms for mirror placement based on Jamin's own performance metrics and evaluate their results using the Inet topology generator [114]. In our case, graph metrics such as *joint degree distribution* (JDD) or betweenness which have been shown by Mahadevan et al. [10] to play a central role in determining a wide range of topological properties, could also be used as an informative way of placing nodes in the network. However, we concentrate on analysing the impact of monitor and destination location in the network based on the Subramanian 5-level AS classification.

### 5.1.2 Experimental planning approach

In contrast to a discovery's preparation and evaluation, its simulation within our framework is a costly issue from a time point of view. For instance, the C-BGP simulator takes in the region of an hour to propagate 1000 prefixes, in a large AS-level topology composed of approximately 17000 ASs and 38000 relationships between them. As a consequence, it is not feasible to test all combinatorial possibilities and choosing a subset of these possible experiments is therefore compulsory. A formalisation of the process can help in such a task.

A process can be considered as a black-box function of one or several input variables, i.e. *qualitative*[2] or *quantitative*[3] factors, resulting in one or several output variables, i.e. the processed results. In our case, the discovery process is determined by the number of sources and destinations for each level and router type if applicable[4] and it yields two main results, i.e. global router and link coverage defined in Section 4.2.3. Other results such as detailed coverage or total number of used probes could also be considered.
To optimise a given process, statistical design tools are available to generate flexible and optimal testing procedures. Their first step is to approximately sketch the process's behaviour in order to identify the most influential factors. Once these factors have been isolated, a statistical model, i.e. linear, quadratic or higher-order term, is chosen to represent the process and is often the result of a trade-off between:

- the **process's behaviour**: a complex behaviour requires a complex model to describe it precisely.
- the **study's needs**: a higher-order model may be needed, but it requires many more experiments for its analysis, On the other hand, complex models are not always easy to understand and simplifying the model sometimes helps interpretation.
- the **experiments' cost**: although both the process and study's needs would require a complete modelisation of the phenomenon, the cost of an individual experiment may be very high and as a consequence may not allow a sufficient number of such experiments to be carried out in order to meet model validation requirements.

[2] A qualitative variable is a non-numerically valued variable and can be placed into distinct categories.
[3] A quantitative variable is a numerically valued variable and can be ordered or ranked.
[4] If no underlying router-level topology is given, all routers are classified as border routers.

In these situations, statistical tools such as classic *composite* experimental plan designs are available to generate test procedures with good statistical properties. Other optimal plan designs such as the well-known *D-optimal* design provide the means to evaluate a plan's quality according to, in principle, any criteria. The D-criterium[5] is however by far the most used for its effectiveness and simplicity. It can notably take into account constraints that are imposed on the experimental input variables so that the problem of designing *mixture experiments*[6] can also be tackled.

Such constraints could thus be considered in the case of the discovery process, as for instance, taking into account the higher cost of deploying actively probing sources in contrast to tracing passive destinations. Another constraint could consist in attributing a lower cost, thus higher priority to discovering the core levels of the topology. Furthermore, discovery experiments could be given mixture restrictions by setting the number of sources we are prepared to deploy and obtain, as a result of the D-optimal design, a way to evaluate their optimised distribution in the network. Indeed, once the resulting experimental plan has been put in place and has been executed, standard regression analysis can be performed to determine, for instance, the model's empirical surface response and ultimately predict the input values to attain the best results.

This approach was deemed promising in the context of this thesis. Unfortunately, after concertation and several meetings with the internal service of the Institute of Statistics (UCL), the outcome was that an issue concerning the inputs' domains was unresolvable in the given amount of time; and according to the best knowledge of those involved, these statistical tools are indeed applicable to both qualitative and quantitative variables, but only *continuous*[7] variables as opposed to *discrete*[8] quantitative variables which is the case of our inputs. Nevertheless a proposal was made to generate an experimental plan by stating continuous variables and rounding the resulting input combinations in order to be fed into the simulator, but this was discarded as statistical validity would have been compromised.

[5] consists of maximizing Fisher's information matrix, which in turn maximizes the volume of the confidence ellipsoid of the regression estimates of the linear model parameters.

[6] The challenge in a mixture experiment is to recognize and handle the inherent restriction that the sum of the mixture components is always 100%.

[7] A quantitative variable whose possible values form some interval of numbers.

[8] A quantitative variable whose possible values form a finite (or countably infinite) set of numbers.

### 5.1.3 Effect tests layout

As a consequence of the unfruitful attempt to take up the powerful approach of experimental planning, the combination of test inputs was chosen in order to analyse the following specific effects:

1. the influence of the number of destinations
2. the influence of the number of sources
3. the influence of the location of sources
4. the influence of the location of destinations
5. the influence of clustering
6. the influence of capping
7. the influence of clustering with capping

### 5.1.4 Choice of topologies

Resulting from the network model requirements described in Section 4.1.2, topologies used for the simulations must at least comply to the provision of a policy-based AS-level topology. For further studies, an underlying router network can also specified in order to fully benefit from the discovery framework's potential.

As described in Section 2.1.1, the first available global network topologies were AS-level topologies inferred from publicly available BGP table dumps and made available at repositories such as Routeviews' [115]. Similar approaches of relying on existing network topologies were made by proposals such as Rocketfuel [58] which led to the first router-level topologies. However, Section 3.1.3 describes how such methods have their limitations as they are known to sometimes miss multiple paths or infer non-existing links and routers.

The next approach consisted of generating synthetic topologies through the observed graph properties of real networks. It was adopted by projects such as BRITE [116] and GT-ITM [117] which are both able to produce router-level topologies. A final approach taken by Quoitin's IGen [30] has been to build topologies by taking into account network design objectives such as latency minimization, link dimension and redundancy against possible failures.

Initial tests are performed on a small Internet-like topology[9] setup with the help of IGen and providing a small AS-level and router-level topology for exhaustive testing purposes.

The first larger-scale tests are performed on an AS-level topology inferred through Subramanian and Agarwal's previously described work [42] in Section 2.1.1, which is currently available from C-BGP's website[10]. If you wish to revisit this topology's distribution of nodes and edges, please respectively refer to Figures 4.1 and 4.2.

## 5.2 Tests and results

Validation tests covering a selection of setups and discoveries were performed and their details are available in the appended material. Furthermore, the following tests which were set out in different plans according to their objectives, also have their setups and results all available in the appended material[11].

Note that preparations are based on independent sources and their results are based on the Doubletree algorithm without the use of Bloom filters to avoid adding unnecessary unknowns.

The main results are available in the appended material and are focused on the following values:

- Global node and link coverage
- Per-level node coverage
- Number of "probes" received by all edges (equal to the sum of the following)
- Number of node probes received per-level
- Global Peering and Provider-Customer link coverage

[9] composed of 5 domains, one for each continent, whose routers were generated using a uniform distribution. Each domain is divided in 10 PoPs using k-medoids, each one containing 2 backbone routers, the others being access routers. Each access router is connected to the two backbone routers in its PoP and backbone routers in a PoP are connected in full-mesh. Finally, all the backbone routers are connected using a Delaunay triangulation.

[10] at http://cbgp.info.ucl.ac.be/

[11] Supplied on CD-ROM

### 5.2.1 Standard deviation between repeated experiments

Before interpreting the results, an initial observation of standard deviation errors is performed between repetitions of a same experiment. Since the outcome of a discovery process is deterministic as described in Section 4.1.1, several inputs of identical parameters were fed to the preparation module to generate similar yet randomly affected discovery setups of sources and destinations. Between one and five repetitions were deployed according to the experiment's resource usage.

When comparing most repetitions' standard deviation relative to the experiment's average, observed results are consistently close to each other with an average deviation of 3% is observed over all preceding values for preliminary experiments. A reasonable degree of confidence can therefore be given to interpreting expensive and thus singly observed experiments although an attentive eye must be kept on these deviations in the eventuality of unexpected issues.

### 5.2.2 Impact of source and destination placement

#### Influence of the location of a single source

The influence of the single source's location was analysed by individually placing one source node in the different network levels and probing towards the full range of stub destinations. No difference according to the level's source can be observed neither in the variation of global interface and link coverage, nor in the peering and provider-customer coverage.

A first phenomenon is observable in Figure 5.1 which represents the total number of probes imposed to the whole network when a source is respectively sited in one of the five bottom levels. The coloured layers represent the value obtained through simulations using the specified numbers of destinations. When a source is moved closer to the dense core, the total number of probes is significantly reduced irrespective of the number of destinations and without affecting the global coverage values. This is possibly due to the absence of an underlying router-level topology although, intuitively, it seems a logical result as the dense core can be viewed as the central point around which other layers evolve and are so more easily reachable. Interestingly enough, probing load also slightly drops when a source belongs to the customer level when compared to the ISP level.

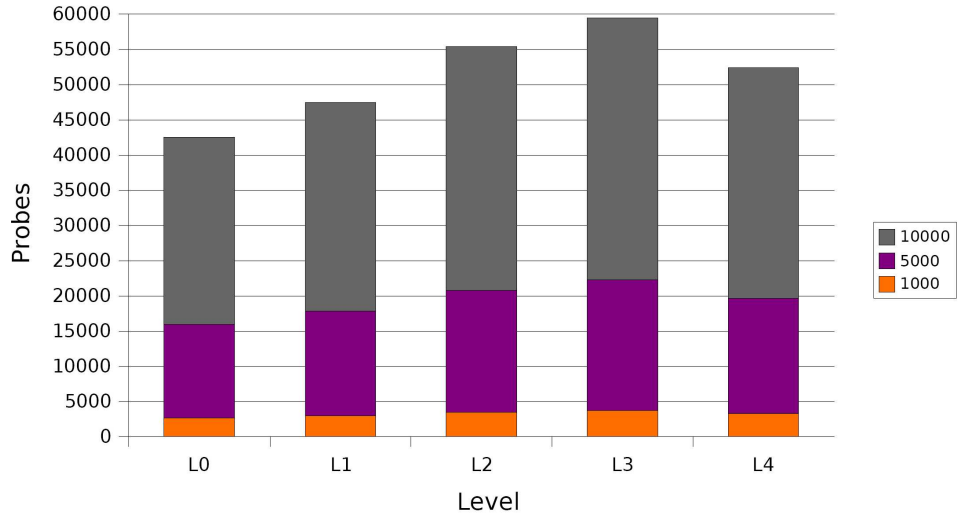


Figure 5.1: Influence of the location of a source on the total number of probes

As shown in Figure 5.2, another tendency can be observed when plotting the individual level coverage against an increasing number of destinations . When the set of destinations is small, in the region of a hundred or a thousand, the proportion of knowledge about the dense and transit core is far higher than with the other levels. This effect then tends to disappear with an increasing number of destinations although the level 2 outer core becomes the least discovered. Note that the global interface coverage closely follows the customer level's evolution since its proportion of nodes is far greater than others.

A more level-orientated view of the average number of probes received per-level is given in Figure 5.3 which represents the number of probes to each level and according to different destination setups. It can be observed that dense and transit core levels are almost as likely to have their nodes probed by a single source compared to the outer core and ISP levels which remain relatively unprobed. In the context of a single source discovery, this shows how biased an active probing approach can be to discovering the topology. Although many nodes may be discovered, they can represent a significant proportion of some levels and not others, thus not being representative of the real topology.

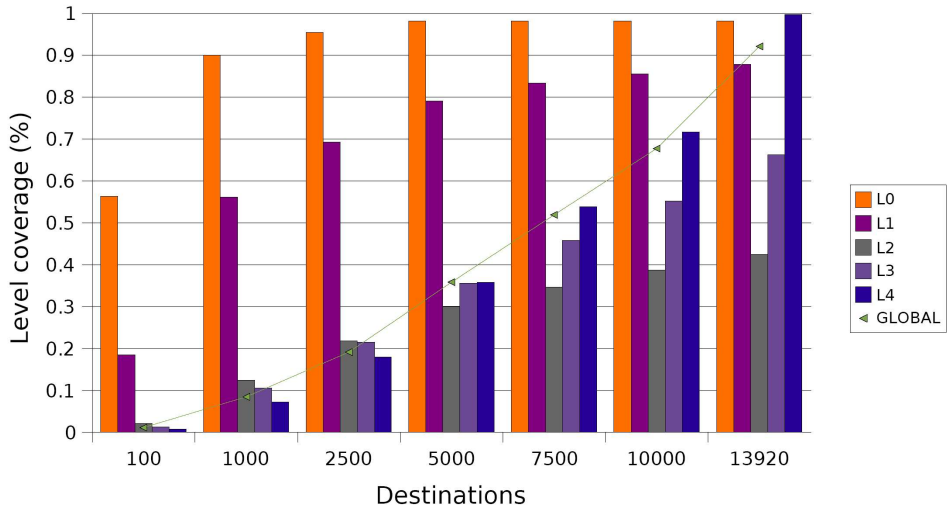


Figure 5.2: Evolution of individual levels' coverage with increasing number of destinations

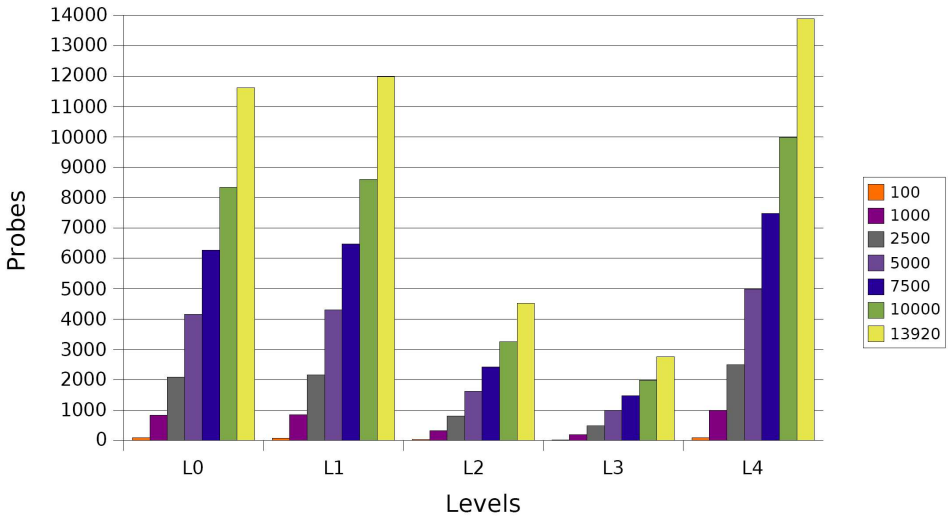


Figure 5.3: Average number of probes received at each topology level from a single source

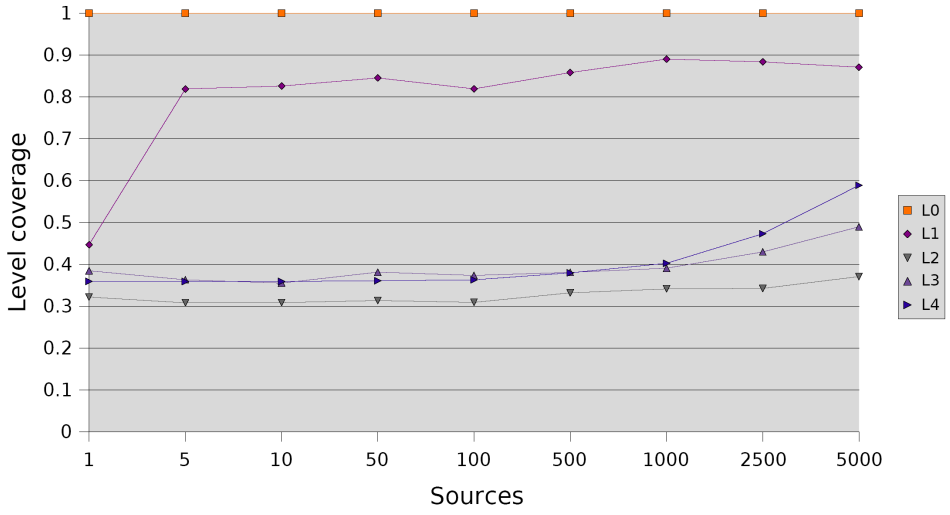


Figure 5.4: Evolution of levels' node coverage with an increasing number of sources

### Influence of the number of sources

The influence of the number of sources was analysed by increasing the number of sources localized in the stub networks, i.e. customer level, in the presence of two large destinations sets of respectively 5000 and 10000 destinations also located at the customer level.

An initial effect of increasing the number of sources can be seen on the evolution of individual levels' coverage in Figure 5.4. As expected they tend to go up, but in different manners. The core level's coverage is already at its maximum since the number of destinations is already quite high and as previously explained by the earlier discovery of its nodes in a single source context. A sharp increase of level 1 coverage happens when the first few sources are added whereas other levels' coverage stabilize at a similar level before increasing as the number of sources converges with the number of destinations. Figure 5.5 offers a different view that can be observed in the case of an initially higher destination count, i.e. respectively 5000 destinations for the former and 10000 for the latter. Indeed, each level seems to stabilize rapidly.

A second effect is the noticeable difference in behaviour between the discovery of peering links and provider-customer links. While the coverage in peering links seems to evolve more rapidly at first, it then stabilizes as provider-customer links coverage increase more sharply. Note that these coverage values may be deceptive, since peering links are far scarcer than provider-customer links. A higher coverage increase may hence correspond to an identical increase in absolute number or links.

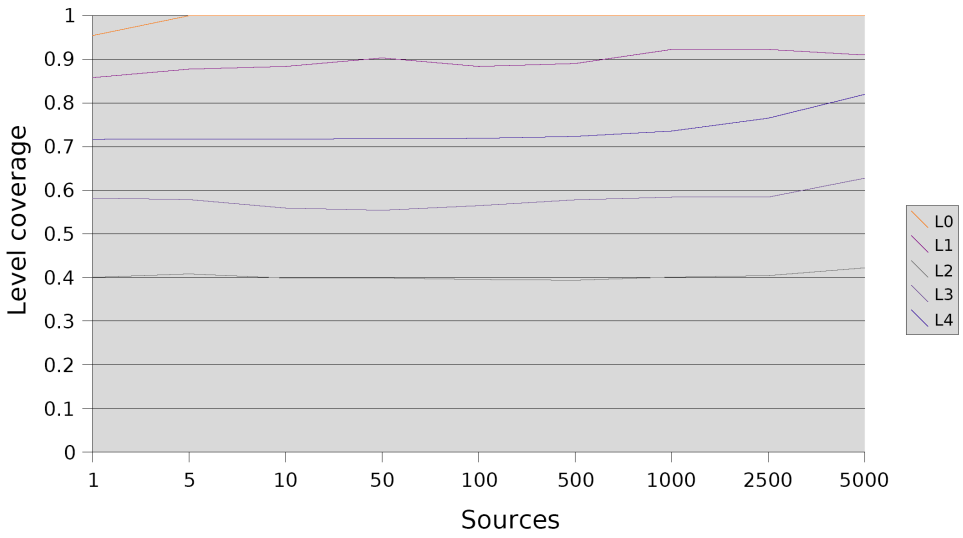


Figure 5.5: Evolution of levels' node coverage with an increasing number of sources

**Influence of the location of multiple sources**

The next step was to extend the single source case by analysing the results produced by the location of *multiple* sources grouped in a specific level. Once again, this does not have a major impact on the topology's node or edge coverage although a slight advantage is given to sources closer to the dense core as they discover 40% to 50% more peering links than the outer sources. This is probably a result of the high number of peering relationships characterizing the dense core. However, there is again confirmation of a significant decrease in the total number of probes performed when the sources get closer to the dense core while keeping the same coverage values. Additionally, higher variations appear in the probes' cross-level distribution making it less clearly draw a picture and thus complicating its interpretation.

If grouped sources do not have any influence, maybe a particular distribution of sources throughout the levels may have. Tests therefore tried multiple possibilities for distributing 100 and 1000 sources across all five levels and were compared to previous experiments using the same number of sources and destinations. The same result was observed for both: the best performing layout, by a small margin, is one where all sources regroup as close to the dense core as possible.

#### Influence of the number and location of destinations

Since the location of sources did not appear to be very influential, the analysis of both the number and location of destinations was carried out conjointly and deployed two reference sets of 100 and 1000 stub sources respectively.

As a first result, the maximum coverage able to be completed by both sets towards all destinations was 99.7%[12] although link discovery peaks at 51.5% comprising 16.1% of peering edges and 52.4% of provider-customer edges.

A second result was that for the two simulations (and all their repetitions) which had all their destinations located in either the ISP level or in the outer core level but with sources still in the customer level; no probes were found to hit other customer level nodes as implied by the AS classification algorithm implemented in the classification module described in 4.2.1.

### 5.2.3 Impact of clustering and capping

#### Influence of clustering

Table 5.1 shows the result of four simulations using 1000 sources probing to all possible destinations.

| number of clusters | interface coverage | PP coverage | PC coverage | probes |
|---|---|---|---|---|
| 5 | 100% | 31.6% | 90.8% | 5243527 |
| 10 | 100% | 28.1% | 89.8% | 2803478 |
| 50 | 100% | 25.3% | 84% | 705406 |
| 100 | 100% | 23.8% | 79.1% | 392802 |

Table 5.1: Performance results of 1000 clustered sources

Note that these clustered performances can not be compared to previous coverage results since they generate $s/n$ more destinations traces than from the single set, such as $s$ is the number of sources and $n$ the number of clusters. This could however give an approximation of the number of overall probes sent out. For instance, Simulation 175[13], using 1000 sources with all possible destinations distributed in their independent sets, should yield 1000/100 thus 10 times

[12] A few nodes have actually been missed despite the exhaustive approach and is supposedly due to incompatible policies of the used topology. The precise cause could be identified but either requires re-performing the simulation and activating the xml export option to identify the missing interfaces from the discovery state, or to display debugging information which would display an unreachable destination error.

[13] refer to CD-ROM material for additional information

less traces than the fourth simulation from Table 5.1. It actually results in 61224 probes instead of the estimated 40000 probes. The improvement made by inter-monitor collaboration would account for this, although as previously stated, probe values' interpretation must be made with caution in the case of these large-scale topology discoveries.

#### Influence of capping

As the capping limit increases, so do the performance figures since more sources are able to probe a given destination as shown by Table 5.2. The number of performed traces is implemented in the setup script described in Section 4.2.1 to be equal $c * d$, such as $c$ is the cap limit and $d$ the number of destinations. The first simulation of Table 5.2 is thus similar to the last entry of 5.1 as they both carry out a total of 10000 traces.

| **cap limit** | **interface coverage** | **PP coverage** | **PC coverage** | **probes** |
|---|---|---|---|---|
| 10 | 100% | 23.8% | 79.3% | 392660 |
| 5 | 100% | 22.7% | 72.6% | 221568 |
| 3 | 100% | 20.6% | 66.7% | 144136 |
| 1 | 100% | 17.3% | 51.9% | 61305 |

Table 5.2: Performance results of 1000 sources with 1000 capped destinations

Note that, unfortunately, the full clustering with capping results are not available for analysis because the needed simulations are still in progress at the time of writing.

### 5.2.4 Summary results

The placement of source and destination nodes in our simulated topology by using Subramanian's 5-level classification has not revealed the existence of a new level-based heuristic for drastically improving network topology coverage. However, access to the detailed proportions in which nodes and edges are discovered helps realize the fundamental difficulties in discovering a complete topology with peering links.

The inherent bias single sources display due to their tree-like explorations of the network needs to be counter-balanced by the deployment of a larger number of sources. As seen through the application of clustering and capping methods to existing collaborative methods such as Doubletree, a distributed discovery process can yield excellent results compared to traditional methods.

# Chapter 6

# Summary conclusions

A framework for active topology discovery analysis has been designed and operates within an advanced network simulator. The design objectives have been achieved, and the basic framework allows for further refinement of the process with the potential to adressing issues that have emerged during this phase and future enhancements. Its features allow it to characterize unknown AS-level topologies according to a relevant classification method, extend its realism by enabling an underlying router-level topology to be specified and apply them the implemented network discovery tools. The detailed router-level node and inter-level link discovery matrix analysis can also be used to further evaluate their performance.

Several ideas of further work have appeared from the readings leading to the description of the litterature and through the design of easily implementable extensions to the current framework.

From an implementation point of view, currently, tables of Section 4.2.3 give an evaluation of the interface/router-level topology discovery although this evaluation of the AS-level discovery could be performed in different ways. The first step would be to either infer the AS topology based on existing methods, such as those described in Section 2.1.3, or to more simply consider an AS as "discovered" if at least one of its routers has been discovered. The latter requires to make the hypothesis of being able to map every router to an AS which is not always the case in the real-world. The second step would to compute a discovered connectivity matrix based on the AS classification known *a priori*. The second way would be to compute that same connectivity matrix based on the classification of the *discovered* topology. This could lead to analysis of the induced bias described in Section 3.1.3 due to the small number of monitors. In this regard, an option for exporting the global discovery state to a specified file has been foreseen and thus allows the discovered router topology to be extracted for future use without requiring a new discovery process.

A second extensions could be testing heuristics based on identified important graph characteristics [10] such as *betweenness* which can be the object of an effective implementation based on Brandes' algorithm [118].

Having partly come to terms with the vastness of the Internet topology discovery topic, the development of a functional and original framework based on state-of-the-art research has been very enthusing especially when as application finally leads to coherent results, confirmation of previous work and further contributing relevant facts and means. Although time and resource constraints had an impact on some aspects of the original plan, the results demonstrate the potential of the approach.

# Bibliography


[1] E.C. Rosen. Vulnerabilities of network control protocols: An example. RFC 789, July 1981.

[2] Y. Rekhter. BGP Protocol Analysis. RFC 1265 (Informational), October 1991.

[3] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), August 2004.

[4] S. Murphy. BGP Security Vulnerabilities Analysis. RFC 4272 (Informational), January 2006.

[5] A. Retana, R. White, V. Fuller, and D. McPherson. Using 31-Bit Prefixes on IPv4 Point-to-Point Links. RFC 3021 (Proposed Standard), December 2000.

[6] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771 (Draft Standard), March 1995. Obsoleted by RFC 4271.

[7] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. On inferring as-level connectivity from bgp routing tables, 2001.

[8] David G. Andersen, Nick Feamster, and Hari Balakrishnan. Topology Inference from BGP Routing Dynamics. In *2nd ACM SIGCOMM Internet Measurement Workshop*, Boston, MA, November 2002.

[9] Internet routing registries, http://www.irr.net/.

[10] Priya Mahadevan, Dmitri Krioukov, Marina Fomenkov, Bradley Huffaker, Xenofontas Dimitropoulos, kc claffy, and Amin Vahdat. The internet as-level topology: Three data sources and one definitive metric. *ACM SIGCOMM COMPUTER COMMUNICATION REVIEW*, 36:2006, 2005.

[11] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Networks*, 9(6):733–745, 2001.

[12] B. Huffaker, D. Plummer, D. Moore, and k claffy. Topology discovery by active probing, 2002.

[13] A. Lakhina, J. Byers, M. Crovella, and P. Xie. Sampling biases in IP topology measurements. *Boston University Computer Science, Tech. Rep. BUCS-TR-2002-021*, July 2002.

[14] Dimitris Achlioptas, Aaron Clauset, David Kempe, and Cristopher Moore. On the bias of traceroute sampling; or, power-law degree distributions in regular graphs. Mar 2005.

[15] Luca Dall'Asta, Ignacio Alvarez-Hamelin, Alain Barrat, Alexei Vázquez, and Alessandro Vespignani. Exploring networks with traceroute-like probes: theory and simulations. *Theor. Comput. Sci.*, 355(1):6–24, 2006.

[16] M. Latapy L. Guillaume. Relevance of massively distributed explorations of the internet topology: simulation results. 2005.

[17] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *Proc. ACM SIGMETRICS*, Banff, Canada, Jun. 2005. See also the traceroute@home project: http://trhome.sourceforge.net.

[18] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network, vol. 19, no. 6, November 2005*, 2005.

[19] R. Atkinson, S. Floyd, and Internet Architecture Board. IAB Concerns and Recommendations Regarding Internet Research and Evolution. RFC 3869 (Informational), August 2004.

[20] D. MAGONI. Tearing down the internet. *IEEE Journal on Selected Areas in Communications*, 21:949–960, August 2003.

[21] J. Li, M. Sung, J. Xu, and L. Li. Large-scale IP traceback in high-speed internet: Practical techniques and theoretical foundation, 2004.

[22] W. Robertson C. Kruegel, D. Mutz and F. Valeur. Topology-based detection of anomalous BGP messages. *6th Symposium on Recent Advances in Intrusion Detection Proceedings, 2003.*, 2003.

[23] A. Jin, S. Bestavros. Small-world characteristics of the internet and multicast scaling. *11th IEEE/ACM International Symposium, 2003.*, 2003.

[24] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[25] J.-J. Pansiot D. Magoni. Analysis of the autonomous system network topology. *ACM SIGCOMM Computer Communication Review*, 31(3):26 – 37, July 2001.

[26] H. Jeong S. H. Yook and A. Barabasi. Modeling the internet's large-scale topology. *Proc. Nat. Acad. Sciences*, 99:13382–13386, Oct. 2002.

[27] BRITE, Boston University Representative Internet Topology Generator, http://www.cs.bu.edu/brite.

[28] GT-ITM, Georgia Tech Internetwork Topology Models, http://www-static.cc.gatech.edu/fac/ellen.zegura/graphs.html.

[29] TIERS, topology generator, http://www.isi.edu/nsnam/dist/topogen/tiers1.1.tar.gz.

[30] Bruno Quoitin. Topology generation based on network design heuristics. In *CoNEXT'05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 278–279, New York, NY, USA, 2005. ACM Press.

[31] C. de Launois. GHITLE, Generator of Hierarchical Internet Topologies using LEvels, 2003.

[32] Stefan Savage, David Wetherall, Anna R. Karlin, and Tom Anderson. Practical network support for IP traceback. In *SIGCOMM*, pages 295–306, 2000.

[33] Kihong Park and Heejo Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2001. ACM Press.

[34] C. Labovitz, R. Wattenhofer, S. Venkatachary, and A. Ahuja. The impact of internet policy and topology on delayed routing convergence. 2001.

[35] Hongsuda Tangmunarunkit, Ramesh Govindan, Scott Shenker, and Deborah Estrin. The impact of routing policy on internet paths. In *INFOCOM*, pages 736–742, 2001.

[36] L. Gao and F. Wang. The extent of AS path inflation by routing policies. *IEEE Global Internet Symposium*, 2002.

[37] S. Shenker H. Tangmunarunkit, R. Govindan. Internet path inflation due to policy routing. In *Proceeding of SPIE ITCom 2001, Denver 19-24 August 2001*, pages 188–195, Aug. 2001.

[38] D. Meyer. The RouteViews project, august 2004, http://www.routeviews.org/.

[39] RIPE NCC, RIPE WHOIS database.

[40] Routeserver, http://www.bgp4.net/.

[41] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. In *Measurement and Modeling of Computer Systems*, pages 307–317, 2000.

[42] Lakshminarayanan Subramanian, Sharad Agarwal, Jennifer Rexford, and Randy H. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proc. of IEEE INFOCOM 2002, New York, NY*, Jun 2002.

[43] G. Di Battista, M. Patrignani, and M. Pizzonia. Computing the types of the relationships between autonomous systems. *Technical Report RT-DIA-73-2002, Dipartimento di Informatica e Automazione, Universita di Roma Tre*, 2002.

[44] Jianhong Xia University. On the evaluation of AS relationship inferences.

[45] B. Quoitin and O. Bonaventure. A survey of the utilization of the BGP community attribute, 2002.

[46] Ramesh Govindan and Anoop Reddy. An analysis of internet inter-domain topology and route stability. pages 850–857, 1997.

[47] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[48] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998.

[49] P.V. Mockapetris. Domain names: Concepts and facilities. RFC 882, November 1983. Obsoleted by RFCs 1034, 1035, updated by RFC 973.

[50] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518 (Proposed Standard), September 1993.

[51] M. Gunes and K. Sarac. Analytical IP alias resolution. *IEEE International Conference on Communications (ICC)*, 2006.

[52] R. Teixeira, K. Marzullo, S. Savage, and G. Voelker. In search of path diversity in isp networks. *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, October 2003.

[53] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 2463 (Draft Standard), December 1998. Obsoleted by RFC 4443.

[54] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM 2000*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE.

[55] CAIDA, Iffinder tool.

[56] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. *4th USENIX Symposium on Internet Technologies and Systems*, 2002.

[57] Neil Spring Mira. How to resolve IP aliases.

[58] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. 32(4):133–145, October 2002.

[59] Vivek Pai Ming Zhang, Yaoping Ruan and Jennifer Rexford. How dns misnaming distorts internet topology mapping. In *Annual Technical Conference Abstract*, pages 369–374. USENIX, 2006.

[60] A. Broido and k claffy. Internet topology: connectivity of IP graphs. *Proceedings of SPIE ITCom*, August 2001.

[61] Xiaoliang Zhao, Dan Pei, Lan Wang, Dan Massey, Allison Mankin, Felix S. Wu, and Lixia Zhang. An analysis of bgp multiple origin as (moas) conflicts. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 31–35, New York, NY, USA, 2001. ACM Press.

[62] Zhuoqing Morley Mao, Jennifer Rexford, Jia Wang, and Randy H. Katz. Towards an accurate as-level traceroute tool. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 365–378, New York, NY, USA, 2003. ACM Press.

[63] Z. Mao, D. Johnson, J. Rexford, J. Wang, and R. Katz. Scalable and accurate identification of as-level forwarding paths. *Proceedings of the IEEE INFOCOM*, March 2004.

[64] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). RFC 1105 (Experimental), June 1989. Obsoleted by RFC 1163.

[65] Vern Paxson. End-to-end routing behavior in the internet. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 25–38, New York, NY, USA, 1996. ACM Press.

[66] Lisa Amini, Anees Shaikh, and Benning Schulzrinne. Issues with inferring internet topological attributes. *Computer Communications*, 27(6):557–567, 2004.

[67] S. V. Krishnamurthy Y. He, M. Faloutsos and B. Huffaker. On routing asymmetry in the internet. *IEEE GLOBECOM, Autonomic Internet, St. Louis*, November 2005.

[68] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (RPSL), 1999.

[69] RIPE NCC. routing registry consistency check reports.

[70] J. Moy. OSPF specification. RFC 1131 (Proposed Standard), October 1989. Obsoleted by RFC 1247.

[71] D. Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Informational), February 1990.

[72] Ramesh Govindan, Cengiz Alaettinog-lu, Kannan Varadhan, and Deborah Estrin. Route servers for inter-domain routing. *Computer Networks and ISDN Systems*, 30(12):1157–1174, 1998.

[73] T. Kernen. http://www.traceroute.org/.

[74] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFC 950.

[75] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006.

[76] J.C. Mogul and J. Postel. Internet Standard Subnetting Procedure. RFC 950 (Standard), August 1985.

[77] V. Jacobson and S. Deering. Traceroute tool, available for download at ftp://ftp.ee.lbl.gov/traceroute.tar.gz, 1989.

[78] N. McCarthy. Layer four traceroute. http://pwhois.org/lft/.

[79] V. Jacobson. Pathchar tool, available for download at ftp://ftp.ee.lbl.gov/pathchar/.

[80] A. Downey. Using pathchar to estimate internet link characteristics. *Proc.SIGCOMM 1999, Cambridge, MA*, pages 241–250, September 1999.

[81] M. Toren. tcptraceroute. http://michael.toren.net/code/tcptraceroute.

[82] Ehud Gavron. Nanog traceroute distribution, ftp://ftp.login.com/pub/software/traceroute/.

[83] Princeton University. Planetlab, http://www.planet-lab.org/.

[84] NLANR Measurement and Network Analysis Group. Amp, active measurement project, http://amp.nlanr.net/.

[85] Y. Shavitt. Dimes, distributed internet measurements and simulations, http://www.netdimes.org.

[86] Young Hyun, Andre Broido, and kc claffy. On third-party addresses in traceroute paths. In *Passive and Active Measurement Workshop 2003*, La Jolla, CA, Apr 2003.

[87] Yuval Shavitt and Eran Shir. Dimes: let the internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5):71–74, 2005.

[88] T. Moors. Streamlining traceroute by estimating path lengths. In *Proc. IEEE International Workshop on IP Operations and Management (IPOM)*, 2004.

[89] Renata Teixeira, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker. Characterizing and measuring path diversity of internet topologies. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 304–305, New York, NY, USA, 2003. ACM Press.

[90] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), January 2001.

[91] J. Byers P. Barford, A. Bestavros and M. Crovella. On the marginal utility of network topology measurements. *SIGCOMM Internet Measurement Workshop*, 2001.

[92] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. The origin of power laws in internet topologies revisited.

[93] Lun Li, David Alderson, Walter Willinger, and John Doyle. A first-principles approach to understanding the internet's router-level topology. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14, New York, NY, USA, 2004. ACM Press.

[94] Aaron Clauset and Cristopher Moore. Accuracy and scaling phenomena in internet mapping. *Physical Review Letters*, 94:018701, 2005.

[95] T. Friedman B. Donnet, P. Raoult and M. Crovella. Traceroute@home project, http://trhome.sourceforge.net/.

[96] B Donnet, T. Friedman, and M. Crovella. Improved algorithms for network topology discovery. In *Proc. Passive and Active Measurement Workshop (PAM)*, Boston, MA, USA, Mar. 2005.

[97] B. Donnet and T. Friedman. Topology discovery using an address prefix based stopping rule. In *Proc. EUNICE Workshop*, Madrid, Spain, Jul. 2005.

[98] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[99] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519 (Proposed Standard), September 1993.

[100] T. Friedman B. Donnet, B. Huffaker and Kc claffy. Increasing the coverage of a cooperative internet topology discovery algorithm. submitted to internet measurement conference (imc) 2006., 2006.

[101] B. Donnet. Chapter 8 - windowed doubletree - doctorate thesis. sept 2006.

[102] Benoit Donnet, Philippe Raoult, and Timur Friedman. Efficient route tracing from a single source, 2006.

[103] Benoit Donnet, Bruno Baynat, and Timur Friedman. Retouched bloom filters: Allowing networked applications to flexibly trade off false positives against false negatives, 2006.

[104] B. Donnet and T. Friedman. Topology discovery using an address prefix based stopping rule. *IFIP, Internation Federation for Information Processing*, 196:119–130, Mar. 2006.

[105] B. Donnet, B. Huffaker, T. Friedman, and kc claffy. Implementation and deployment of a distributed network topology discovery algorithm. cs.NI 0603062, arXiv, Mar. 2006. See also the traceroute@home project: http://trhome.sourceforge.net.

[106] T. Friedman et Kc claffy. B. Donnet, B. Huffaker. Evaluation of a large-scale topology discovery algorithm. in proc. 6th ieee international workshop on ip operations and management (ipom). oct. 2006, dublin, ireland.

[107] X. A. Dimitropoulos and G. F. Riley. Creating realistic bgp models. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 64–70, 2003.

[108] GNU Zebra Open-Source Routing Software, http://www.zebra.org/.

[109] UCB/LBNL/VINT. The NS-2 network simulator, Aug. 2003. http://www.isi.edu/nsnam/ns/.

[110] J. Cowie and H. Liu. Towards realistic million-node internet simulations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[111] Ahmed Sobeih, Wei P. Chen, Jennifer C. Hou, Lu C. Kung, Ning Li, Hyuk Lim, Hung Y. Tyan, and Honghai Zhang. J-sim: A simulation environment for wireless sensor networks. In *Annual Simulation Symposium*, pages 175–187, 2005.

[112] Timothy G. Griffin and Gordon Wilfong. An analysis of bgp convergence properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 277–288, New York, NY, USA, 1999. ACM Press.

[113] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *INFOCOM*, pages 31–40, 2001.

[114] C. Jin, Q. Chen, and S. Jamin. Inet: Internet topology generator, 2000.

[115] D. Meyer. University of oregon route views archive project, http://archive.routeviews.org.

[116] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John W. Byers. BRITE: An approach to universal topology generation. In *MASCOTS*, page 346. IEEE Computer Society, 2001.

[117] K. Calvert, J. Eagan, S. Merugu, A. Namjoshi, J. Stasko, and E. Zegura. Extending and enhancing gt-itm. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 23–27, New York, NY, USA, 2003. ACM Press.

[118] U. Brandes. A faster algorithm for betweenness centrality, 2001.

# Appendix A

# Interaction with a C-BGP instance

The c-bgp simulator provides an easy CISCO-like command-line interface, but simulations are generally configured through the use of scripts. A c-bgp script (*.cli) contains a sequence of c-bgp commands that are used to build the topology by adding nodes and links, and to setup BGP sessions.

```
net add node address1
net add node address2
net add link address1 address2 delay
```

Listing A.1: Example of commands contained in a c-bgp script

The simulator can either be launched in *interactive mode* which gives the user direct access to the command-line interface, or in *script mode* which executes the commands contained in a specified script file. However, for more advanced simulations, several interfaces are provided to facilitate communication between a c-bgp instance and various programming languages (Perl, Python, Java). The two latter experience memory management problems and are still under development. Perl was therefore chosen as the best alternative.

The perl interface is provided for easy communication between a perl script and a c-bgp instance. It comes as a perl module to be imported in the perl script. This module contains methods to establish the dialog and to send and receive messages to and from the c-bgp instance.

```
use CBGP 0.3;

# Create cbgp instance and establish communication
my $cbgp = CBGP->new("../bin/cbgp");
$cbgp->spawn;
```

```
$cbgp->send("set autoflush on\n");

# Interactions with cbgp instance
...

# Terminate communication
$cbgp->finalize;
```

Listing A.2: Communication establishment with the c-bgp simulator

A typical interaction scheme would consist of setting up the simulated environment (e.g. feeding the instance with c-bgp commands from a script), "running it" and then performing one or more requests (e.g. sending a traceroute command and retrieving the answer) :

```
# Simulation setup
$cbgp->send("net add node address1\n");
$cbgp->send("net add node address2\n");
$cbgp->send("net add link address1 address2 delay\n");

# Simulation run
$cbgp->send("sim run\n");

# Send traceroute request and wait for answer
$cbgp->send("net node address1 record-route address2\n");
my $response = $cbgp->expect(1);
```

Listing A.3: Interaction scheme with the c-bgp instance

# Appendix B

# C-BGP patch source code

## B.1 Added/modified functions to `bgp.c`

```
1
2  // ----- cli_bgp_topology_clearribs ----------------------------------
3  /**
4   * Clears all RIB tables for memory purposes; patch by G. Culpin
5   *
6   */
7  int cli_bgp_topology_clearribs(SCliContext * pContext, STokens * pTokens){
8
9          int iIndex;
10         SPtrArray * pRL;
11         SBGPRouter * pRouter;
12         pRL = (SPtrArray *) build_router_list();
13
14         // For all BGP instances...
15         for (iIndex= 0; iIndex < ptr_array_length(pRL); iIndex++) {
16                 pRouter = (SBGPRouter *) pRL->data[iIndex];
17                 if (bgp_router_del_rib(pRouter))
18                                 return CLI_ERROR_COMMAND_FAILED;
19                 }
20         return CLI_SUCCESS;
21 }
22
23 //
24 // ----- cli_bgp_router_del_rib ---------------------------------------
25 /**
26  * Remove RIB tables of router for memory purposes; patch by G. Culpin
27  *
28  */
29 int cli_bgp_router_del_rib(SCliContext * pContext, STokens * pTokens)
30 {
31   SBGPRouter * pRouter;
32
33   // Get BGP instance from context
34   pRouter = (SBGPRouter *) cli_context_get_item_at_top(pContext);
35
36   // Clear the RIB
37   if (bgp_router_del_rib(pRouter))
38     return CLI_ERROR_COMMAND_FAILED;
39
40   return CLI_SUCCESS;
41 }
42
43 // ----- cli_register_bgp_topology ------------------------------------
44 int cli_register_bgp_topology(SCliCmds * pCmds)
```

```
45  {
46    SCliCmds * pSubCmds;
47    SCliParams * pParams;
48
49    pSubCmds= cli_cmds_create();
50    pParams= cli_params_create();
51  #ifdef _FILENAME_COMPLETION_FUNCTION
52    cli_params_add2(pParams, "<file>", NULL,
53                    _FILENAME_COMPLETION_FUNCTION);
54  #else
55    cli_params_add(pParams, "<file>", NULL);
56  #endif
57    cli_cmds_add(pSubCmds, cli_cmd_create("load",
58                                          cli_bgp_topology_load,
59                                          NULL, pParams));
60    cli_cmds_add(pSubCmds, cli_cmd_create("policies",
61                                          cli_bgp_topology_policies,
62                                          NULL, NULL));
63  #ifdef HAVE_XML
64    pParams= cli_params_create();
65  #ifdef _FILENAME_COMPLETION_FUNCTION
66    cli_params_add2(pParams, "<file>", NULL,
67                    _FILENAME_COMPLETION_FUNCTION);
68  #else
69    cli_params_add(pParams, "<file>", NULL);
70  #endif
71    cli_cmds_add(pSubCmds, cli_cmd_create("xml-load",
72                                          cli_bgp_xml_topology_load,
73                                          NULL, pParams));
74    pParams= cli_params_create();
75  #ifdef _FILENAME_COMPLETION_FUNCTION
76    cli_params_add2(pParams, "<file>", NULL,
77                    _FILENAME_COMPLETION_FUNCTION);
78  #else
79    cli_params_add(pParams, "<file>", NULL);
80  #endif
81    cli_cmds_add(pSubCmds, cli_cmd_create("xml-dump",
82                                          cli_bgp_xml_topology_dump,
83                                          NULL, pParams));
84  #endif
85    pParams= cli_params_create();
86    cli_params_add(pParams, "<prefix>", NULL);
87    cli_params_add(pParams, "<input>", NULL);
88    cli_params_add(pParams, "<output>", NULL);
89    cli_cmds_add(pSubCmds, cli_cmd_create("record-route",
90                                          cli_bgp_topology_recordroute,
91                                          NULL, pParams));
92  #ifdef __EXPERIMENTAL__
93    pParams= cli_params_create();
94    cli_params_add(pParams, "<prefix>", NULL);
95    cli_params_add(pParams, "<bound>", NULL);
96    cli_params_add(pParams, "<input>", NULL);
97    cli_params_add(pParams, "<output>", NULL);
98    cli_cmds_add(pSubCmds, cli_cmd_create("record-route-bm",
99                                          cli_bgp_topology_recordroute_bm,
100                                         NULL, pParams));
101 #endif
102   cli_cmds_add(pSubCmds, cli_cmd_create("show-rib",
103                                         cli_bgp_topology_showrib,
104                                         NULL, NULL));
105   // RIB table patch for memory purposes by G. Culpin
106   cli_cmds_add(pSubCmds, cli_cmd_create("clear-ribs",
107                                         cli_bgp_topology_clearribs,
108                                         NULL, NULL));
109   /*
110   pParams= cli_params_create();
111   cli_params_add(pParams, "<prefix>", NULL);
112   cli_params_add(pParams, "<output>", NULL);
113   cli_cmds_add(pSubCmds, cli_cmd_create("route-dp-rule",
114                                         cli_bgp_topology_route_dp_rule,
115                                         NULL, pParams));
```

```
116     */
117     cli_cmds_add(pSubCmds, cli_cmd_create("run",
118                                           cli_bgp_topology_run,
119                                           NULL, NULL));
120     return cli_cmds_add(pCmds, cli_cmd_create("topology", NULL, pSubCmds, NULL));
121   }
122
123   // ----- cli_register_bgp_router_del ---------------------------------
124   int cli_register_bgp_router_del(SCliCmds * pCmds)
125   {
126     SCliCmds * pSubCmds;
127     SCliParams * pParams;
128
129     pSubCmds= cli_cmds_create();
130     pParams= cli_params_create();
131     cli_params_add(pParams, "<prefix>", NULL);
132     cli_cmds_add(pSubCmds, cli_cmd_create("network",
133                                           cli_bgp_router_del_network,
134                                           NULL, pParams));
135     // RIB table patch for memory purposes by G. Culpin
136     cli_cmds_add(pSubCmds, cli_cmd_create("rib",
137                                           cli_bgp_router_del_rib,
138                                           NULL, NULL));
139     return cli_cmds_add(pCmds, cli_cmd_create("del", NULL, pSubCmds, NULL));
140   }
```

## B.2 Added/modified functions to as.c

```
1   // ----- bgp_router_del_rib ----------------------------------------
2   /**
3    * Remove RIB contents of a router; patch by G. Culpin
4    */
5   int bgp_router_del_rib(SBGPRouter * pRouter)
6   {
7           // remove router's local RIB
8           rib_destroy(&pRouter->pLocRIB);
9           pRouter->pLocRIB = rib_create(0);
10
11          // remove router's adjacency RIBs (in and out)
12          int iIndex;
13          SPeer * pPeer;
14          for (iIndex= 0; iIndex < ptr_array_length(pRouter->pPeers); iIndex++) {
15                  pPeer = (SPeer *) pRouter->pPeers->data[iIndex];
16                  rib_destroy(&pPeer->pAdjRIBIn);
17                  rib_destroy(&pPeer->pAdjRIBOut);
18                  pPeer->pAdjRIBIn = rib_create(0);
19                  pPeer->pAdjRIBOut = rib_create(0);
20          }
21
22          return 0;
23  }
```

# Appendix C

# XML discovery files

## C.1 XML Schema – discovery.xsd

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.ucl.ac.be"
       xmlns="http://www.ucl.ac.be"
3   elementFormDefault="qualified">
4
5   <xs:element name="discovery">
6    <xs:complexType>
7     <xs:sequence>
8      <xs:element name="cluster" type="cluster_type" minOccur="1" maxOccur="*"/>
9     </xs:sequence>
10   </xs:complexType>
11  </xs:element>
12
13  <xs:complexType name="cluster_type">
14   <xs:attribute name="id" type="xs:integer" use="required"/>
15   <xs:sequence>
16    <xs:element name="cdest" type="xs:integer" minOccur="0" maxOccur="*"/>
17    <xs:element name="global_set" type="globalset_type" minOccur="0" maxOccur="1"/>
18    <xs:element name="src" type="src_type" minOccur="1" maxOccur="*"/>
19   </xs:sequence>
20  </xs:complexType>
21
22  <xs:complexType name="globalset_type">
23   <xs:anyAttribute/>
24  </xs:complexType>
25
26  <xs:complexType name="src_type">
27   <xs:attribute name="ip" type="xs:integer" use="required"/>
28   <xs:attribute name="hop" type="xs:integer"/>
29   <xs:sequence>
30    <xs:element name="d" type="xs:integer" minOccur="0" maxOccur="*"/>
31   </xs:sequence>
32  </xs:complexType>
33
34 </xs:schema>
```

## C.2 Example of generated discovery file – example1.xml

```
1  <discovery>
2    <cluster id="0">
```

```
 3       <src ip="131145">
 4         <d>196620</d>
 5         <d>196621</d>
 6       </src>
 7       <src ip="23">
 8         <d>196609</d>
 9         <d>262158</d>
10       </src>
11       <src ip="65565">
12         <d>196629</d>
13         <d>196613</d>
14         <d>262149</d>
15       </src>
16       <src ip="65576">
17         <d>196622</d>
18         <d>196632</d>
19       </src>
20     </cluster>
21   </discovery>
```

# Appendix D

# Guidance for script usage

## D.1 AS classification script usage

```
Usage: ./classifyAS.pm <as_topology_file>
```

## D.2 Discovery setup script usage

```
Usage: ./setupDiscovery [options] <as_topology_file>

By default, independent destination sets are generated for sources inside a single cluster

Required:
 --srcs (-s) <i0 i1 i2 i3 i4 b0 b1 b2 b3 b4>  specifies the number of sources per type/level
 --dests (-d) <i0 i1 i2 i3 i4 b0 b1 b2 b3 b4> specifies the number of destinations per type/level

Options:
 --rfile (-r) <file>        specifies the underlying cbgp router topology of given AS topology
 --cap (-c) <n>             specifies the maximum number of monitors (sources) per destination
 --clusters (-n) <n>        specifies the number of clusters in which to divide monitors
 --exclude_src_as_dest (-e) excludes the choice of source nodes as potential destinations

 --ip-string     exports the XML file with IP addresses in dotted format

 --help (-?)     this page
 --info (-i)     show information
 --debug         show debug information
```

## D.3 Discovery simulation script usage

```
Usage: ./performDiscovery [options] <as_topology_file> <xml_discovery_file>

Options:
 --rfile (-r) <file>          specifies the underlying cbgp router topology of given AS topology
 --hop-eval (-h) <n>          specifies the number of destinations used to evaluate initial hop
 --hit-probability (-p) <f>   specifies the probability p of hitting a destination on the first probe
 --max-propagations (-m) <n>  specifies the maximum number of prefixes to propagate in cbgp before a restart
```

```
--bloom-capacity (-c) <n>    specifies the capacity of bloom filters used for encoding global sets
--bloom-error (-e) <f>       specifies the error rate of bloom filters used for encoding global sets
--nobloom (-n)               don't use bloom filters for encoding global sets

--traceroute (-t)            don't use doubletree discovery but simple traceroutes

--help (-?)                  this page
--show-info (-i)             show information
--show-detailed-results (-d) show detailed result information
--debug                      show debug information
```

# Appendix E

# Perl source code

All the following source code is also available on the provided CD-ROM.

## E.1 AS-level topology classification script – `classifyAS`

```
1  #!/usr/bin/perl
2  # ===================================================================
3  # Classifies an AS topology according to "Characterizing the
4  # Internet Hierarchy from Multiple Vantage Points" article
5  # Sibling relationships are relaxed to p2p
6  #
7  # Usage: ./classifyAS.pm in_file
8  #
9  # @author Gregory Culpin
10 # @date 20/02/2006
11 # @lastdate 5/03/2006
12 # ===================================================================
13
14 use strict;
15 use lib ".";
16 use Graph::Directed;
17 use constant {
18         # script options
19         DEBUG => 0, # debug step-by-step algorithm details
20         INFO => 1, # generic information
21         # AS classification
22         AS_CORE => 0,
23         AS_TRANSIT => 1,
24         AS_OUTER => 2,
25         AS_ISP => 3,
26         AS_CUSTOMER => 4,
27 };
28
29 # various utils
30 use Data::Dumper;
31
32 # MAIN
33
34 my $infile = $ARGV[0];
35 my $outfile = $infile.".classified";
36 if (@ARGV!=1) {
37         die "\nUsage: $0 <as_topology_file>\n"
38 }
39 my $astopo = read_file($infile);
40 my $asgraph = Graph::Directed->new();
41
```

```
42  build_asgraph($asgraph,$astopo);
43  classify_and_export($asgraph,$outfile,$infile);
44
45  # SUBS
46
47  sub read_file
48      {
49          my $filename = shift;
50          my @lines;
51
52          open( FILE, "< $filename" ) or die "Can't open $filename : $!";
53          while( <FILE> ) {
54              s/#.*//;            # ignore comments by erasing them
55              next if /^(\s)*$/;  # skip blank lines
56              chomp;              # remove trailing newline characters
57              push @lines, $_;    # push the data line onto the array
58          }
59          close FILE;
60
61          print "- Imported ".(@lines)." lines from $filename\n";
62          return \@lines;  # array reference
63
64  }
65
66  sub build_asgraph {
67
68          my ($g,$topo) = @_;
69
70          foreach (@$topo) {
71                  my @line= split /\s+/;
72                  # add each AS as vertex and relationship as edge
73                  if (scalar(@line) == 3) { # input format [as1 as2 relationship] (0:PP,-1:CP,1:PC
                        ,2:S)
74                      if ($line[2]==0) { # peer-to-peer relationship is represented by two
                            directional edges
75                          $g->add_edge($line[1],$line[0]);
76                          $g->add_edge($line[0],$line[1]);
77                      }
78                      elsif ($line[2]==-1) { # customer-to-provider relationship represented by
                            a directional edge
79                          $g->add_edge($line[1],$line[0]);
80                      }
81                      elsif ($line[2]==1) { # provider-to-customer relationship represented by a
                             directional edge
82                          $g->add_edge($line[0],$line[1]);
83                      }
84                      elsif ($line[2]==2) { # sibling relationship currently dropped (or
                            previously converted back to p2p
85                      }
86                      else {
87                          die "relationship file has syntax errors (wrong value for 3rd
                                argument)\n";
88                      }
89                  }
90                  else {
91                          die "relationship file has syntax errors (wrong number of arguments per
                                line)\n";
92                  }
93          }
94
95          print "- Found ".($g->unique_vertices)." ASes and ".($g->unique_edges)." interactions\n"
                if INFO; # interactions contain double P-P edges
96
97  }
98
99  sub classify_and_export {
100
101         my ($g,$filename,$infile) = @_;
102         my $tmp = $g->copy_graph;
103
104         # FIRST PASS : classify customers (LEVEL 4)
```

```
105
106         my @customers = $tmp->sink_vertices;
107         foreach (@customers){ # sink has predecessors but no successors
108                 $g->set_vertex_weight($_,AS_CUSTOMER); # label customers in AS graph
109                 $tmp->delete_vertex($_); # remove from temp graph
110         }
111         my $num_CUSTOMERS = @customers;
112
113         print "Classified ".$num_CUSTOMERS." customer ASes\n" if INFO;
114
115         # SECOND PASS : reverse pruning to classify small ISPs (LEVEL 3)
116         my $num_ISPs =0;
117         my @leaves = $tmp->sink_vertices;
118         while (@leaves){
119                 my $v = pop(@leaves);
120                 $num_ISPs++;
121                 $g->set_vertex_weight($v,AS_ISP);
122                 $tmp->delete_vertex($v);
123                 if (@leaves==1){ # last leaf was removed from previous leaves
124                         @leaves = $tmp->sink_vertices;
125                 }
126         }
127
128         print "Classified ".$num_ISPs." small ISP ASes\n" if INFO;
129
130         # THIRD PASS (2): greedily classify dense core ASes (LEVEL 0)
131
132         my $num_globalCORE = $tmp->unique_vertices;
133         my $core_edges = $tmp->unique_edges;
134         my $undirected_core = $tmp->undirected_copy_graph;
135
136         my @core = $tmp->unique_vertices; # contains dense core, transit core and outer core
137         my $core_size = scalar @core;
138         my @dense_core;
139         my $temp_transit = $tmp->copy_graph; # graph containing nodes excluding current (for
              greedy search)
140         my $dense = $tmp->undirected_copy_graph; # keep copy to perform dense core stats
141
142         my $z = max_outdegree($tmp); # z is max out_degree node
143         print "AS ".$z." with max connectivity is added to dense core\n" if DEBUG;
144         push (@dense_core,$z);
145         $temp_transit->delete_vertex($z);
146
147         # greedy algorithm
148         while (scalar @dense_core != $core_size){ # while X doesn't contain all core vertices
149                 my ($y,$y_c) = max_connectivity($temp_transit,\@dense_core,$tmp); # get vertex
                      with max connectivity
150                 push (@dense_core,$y); # add to X
151                 print "AS ".$y." with connectivity ".$y_c." is added to dense core\n" if DEBUG;
152                 print "testing if ".$y_c." < ".(scalar @dense_core/2)."\n" if DEBUG;
153                 if ($y_c < (scalar @dense_core/2)) { # conn(k+1) < (k+1)/2 -> previous set was
                      dense, but not anymore
154                     pop (@dense_core);
155                     print "determined greedily dense core of ".(scalar @dense_core)." ASes\n"
                            if DEBUG;
156                     last; # stop greedy search
157                 }
158                 $temp_transit->delete_vertex($y); # update current transit core
159         }
160         my $num_DENSE = scalar @dense_core;
161
162         # label in graph
163         foreach (@dense_core){
164                 $g->set_vertex_weight($_,AS_CORE);
165         }
166         # delete all non-dense nodes from graph to perform dense stats
167         foreach my $v ($dense->unique_vertices){
168                 my $is_dense=0;
169                 foreach (@dense_core) {
170                         if ($_==$v){
171                                 $is_dense=1;
```

```
172                         }
173                 }
174                 if ($is_dense==0){
175                         $dense->delete_vertex($v);
176                 }
177         }
178
179         print "Core contains ".$num_globalCORE." ASes and ".$undirected_core->unique_edges." links
                with average degree of ".$undirected_core->average_degree."\n" if INFO;
180         print "Classified ".$dense->unique_vertices." dense core ASes having ".$dense->
              unique_edges." links with average degree of ".$dense->average_degree."\n" if INFO;
181
182         # FOURTH PASS : greedily classify transit core ASes (LEVEL 1)
183
184         print $temp_transit->unique_vertices." ASes remain to be classified\n";
185
186         my @transit_core;
187         my $temp_outer = $tmp->copy_graph; # graph containing nodes excluding current (for greedy
              search)
188
189         my $z = max_outdegree($tmp); # z is max out_degree node
190         print "AS ".$z." with max connectivity is added to transit core\n" if DEBUG;
191         push (@transit_core,$z);
192         $temp_outer->delete_vertex($z);
193
194         while (scalar @transit_core != $core_size){ # while X doesn't contain all core vertices
195                 my ($y,$y_c) = max_connectivity($temp_outer,\@transit_core,$tmp); # get vertex
                      with max connectivity
196                 push (@transit_core,$y); # add to X
197                 print "AS ".$y." with connectivity ".$y_c." is added to transit core\n" if DEBUG;
198
199                 print "testing if ".in_way_cut($temp_outer,\@transit_core,$tmp)." < ".((scalar
                      @transit_core)/2)."\n" if DEBUG;
200                 if (in_way_cut($temp_outer,\@transit_core,$tmp) < (scalar @transit_core)/2) { #
                      conn(k+1) < (k+1)/2
201                     pop (@transit_core);
202                     print "determined greedily transit core of ".(scalar @transit_core)." ASes
                            \n" if DEBUG;
203                     last; # stop greedy search
204                 }
205                 $temp_outer->delete_vertex($y); # update current transit core
206         }
207
208         # remove included dense core
209         my @final_transit;
210         foreach my $t (@transit_core){
211                 my $is_core=0;
212                 foreach (@dense_core) {
213                         if ($t==$_) {
214                                 $is_core=1;
215                         }
216                 }
217                 if ($is_core==0){
218                         push (@final_transit,$t);
219                 }
220         }
221
222         print "Classified ".scalar @final_transit." transit core ASes\n" if INFO;
223         # label in graph
224         foreach (@final_transit){
225                 $g->set_vertex_weight($_,AS_TRANSIT);
226         }
227         print "Classified ".$temp_outer->unique_vertices." remaining as outer core ASes\n";
228         foreach ($temp_outer->unique_vertices){
229                 $g->set_vertex_weight($_,AS_OUTER);
230         }
231
232         # Write results to file
233         open CLASSIFIED, ">$filename" or
234         die "Error: unable to create CLASSIFIED file \"$filename\": $!";
235
```

```
236         # ---| Header |---
237         print CLASSIFIED "# AS classification performed by ClassifyAS\n";
238         print CLASSIFIED "# on ".localtime(time())." and based on $infile\n";
239         print CLASSIFIED "# Format: as_num level\n";
240         print CLASSIFIED "# Levels : 0=CORE 1=TRANSIT 2=OUTER 3=ISP 4=CUSTOMER\n";
241         print CLASSIFIED "# Topology contains ".$g->unique_vertices." ASes and ".$g->unique_edges.
                " unidirectional relationships\n";
242         #print CLASSIFIED "#                    $num_DENSE COREs, $num_TRANSIT TRANSITs, $num_ISPs
                ISPs, $num_CUSTOMERS CUSTOMERs\n";
243
244         # ---| Classification |---
245         foreach ($g->unique_vertices){
246                 print CLASSIFIED $_." ".$g->get_vertex_weight($_)."\n";
247         }
248
249         close CLASSIFIED;
250 }
251
252 # compute maximum out-degree node of graph g
253 sub max_outdegree {
254         my ($g) = @_;
255         my $max=0;
256         my $max_c=0;
257         foreach ($g->unique_vertices){
258                 if ($g->out_degree($_)>$max_c){
259                         $max=$_;
260                         $max_c=$g->out_degree($_);
261                 }
262         }
263         return $max;
264 }
265
266 # compute node of graph g-X having maximum out-degree in g with a specified set of vertices
267 sub max_connectivity {
268         my ($g_minus_X,$set,$g) = @_;
269         my $max;
270         my $max_c=0;
271         foreach ($g_minus_X->unique_vertices){ # compute c for each vertex in g-X
272                 my $c=get_connectivity($_,$set,$g);
273                 if ($c>$max_c){ # store vertex with max c
274                         $max=$_;
275                         $max_c=$c;
276                 }
277         }
278         return ($max,$max_c);
279 }
280
281 # compute number of outgoing edges from node v to a set of vertices, all in graph G
282 sub get_connectivity {
283         my ($v,$set,$g) =@_;
284         my $count=0;
285         foreach (@{$set}) { # foreach set member, check if exists directed edge from v
286                 if ($g->has_edge($v,$_)){
287                         $count++;
288                 }
289         }
290         return $count;
291 }
292
293 # compute number of incoming edges to the set members from its complement set (g-X)
294 sub in_way_cut {
295         my ($g_minus_X,$set,$g) =@_;
296         my $count=0;
297         foreach my $m (@{$set}) { # foreach set member, count incoming edges from outside (g-X)
298                 foreach ($g_minus_X->unique_vertices){
299                         if ($g->has_edge($_,$m)){
300                                 $count++;
301                         }
302                 }
303         }
304         return $count;
```

```
305 }
```

## E.2 Discovery setup script – `setupDiscovery`

```
1  #!/usr/bin/perl
2  # ==================================================================
3  # Discovery setup script
4  # @(#)setupDiscovery
5  # @author Gregory Culpin
6  # @date 08/04/2006
7  # @lastdate 12/05/2006
8  # ==================================================================
9
10 use topology;
11 use tools;
12 use strict;
13 use lib ".";
14 use Math::BigInt;
15 use Data::Dumper;
16 use Graph::Directed;
17 use Graph::Undirected;
18 use Getopt::Long;
19 use constant {
20         # router-level node labels
21         NLABELS_RNODES => 2,
22         ROUTER_INTERNAL => 0,
23         ROUTER_BORDER => 1,
24         # router-level edge labels
25         NLABELS_RLINKS => 3,
26         RLINK_INTERNAL => 0,
27         RLINK_PP => 1,
28         RLINK_PC => 2,
29         # AS-level node labels
30         NLEVELS => 5,
31         AS_CORE => 0,
32         AS_TRANSIT => 1,
33         AS_OUTER => 2,
34         AS_ISP => 3,
35         AS_CUSTOMER => 4,
36         # router-level edge labels
37         NLABELS_ASLINKS => 2,
38         ASLINK_PP => 0,
39         ASLINK_PC => 1,
40         # result parameters
41         NPARAMETERS => 3,
42         COVERAGE => 0,
43         PROBED => 1,
44         PROBES => 2,
45 };
46 use XML::Simple;
47
48 #-- manage command-line options
49
50 my $asfile; # file containing AS relationships
51 my $rfile; # file containing cbgp router level script
52
53 my @nb_sources; # number of sources to choose randomly at each level$routers_by_level_and_type
54 my @nb_dests; # number of destinations to choose randomly at each level
55 my $nb_clusters; # number of src clusters having individual dest sets rem: if specified, implies
       capping
56 my $cap_limit; # max number of sources per destination
57 my $exclude_s_as_dest; # disable sources from being chosen as destinations
58
59 my $ipstring; # outputs string IP address in XML file
60
61 my $debug; # debug information
62 my $show_info; # generic information
```

```
63  my $help; # show usage
64  my $export=''; # file to export discovery
65
66  GetOptions (
67                  # topology attributes
68                  'rfile|r=s' => \$rfile,
69
70                  # discovery attributes
71                  'srcs|s=i{10}' => \@nb_sources,
72                  'dests|d=i{10}' => \@nb_dests,
73                  'clusters|n=i' => \$nb_clusters,
74                  'cap|c=i' => \$cap_limit,
75                  'exclude_src_as_dest|e=i' => \$exclude_s_as_dest,
76                  #encoding options
77                  'ip-string' => \$ipstring,
78
79                  # display related options
80                  'debug' => \$debug,
81                  'show-info|i' => \$show_info,
82                  'help|?' => \$help,
83                  'export|x=s' => \$export
84  );
85
86  if (@ARGV!=1 or $help) {
87          die "\nUsage: $0 [options] <as_topology_file>
88
89  By default, independent destination sets are generated for sources inside a single cluster
90
91  Required:
92   --srcs  (-s) <i0 i1 i2 i3 i4 b0 b1 b2 b3 b4>\t specifies the number of sources per type/level
93   --dests (-d) <i0 i1 i2 i3 i4 b0 b1 b2 b3 b4>\t specifies the number of destinations per type/
         level
94
95  Options:
96   --rfile (-r) <file>\t\t specifies the underlying cbgp router topology of given AS topology
97   --cap (-c) <n>\t\t\t specifies the maximum number of monitors (sources) per destination
98   --clusters (-n) <n>\t\t specifies the number of clusters in which to divide monitors
99   --exclude_src_as_dest (-e)\t excludes the choice of source nodes as potential destinations
100
101  --ip-string \t exports the XML file with IP addresses in dotted format
102
103  --help (-?)\t this page
104  --info (-i)\t show information
105  --debug\t show debug information\n";
106 }
107 if ($nb_clusters ne undef && $nb_clusters<=0){
108         die "Number of clusters must be >= 1\n";
109 }
110 if ($cap_limit ne undef && $cap_limit<=0){
111         die "Capping limit must be >= 1\n";
112 }
113 if (@nb_sources!=10){
114         die "Sources need to be specified with following option : --srcs <i0 i1 i2 i3 i4 b0 b1 b2
              b3 b4>\n";
115 }
116 if (@nb_dests!=10){
117         die "Destinations need to be specified with following option : --dests <i0 i1 i2 i3 i4 b0
              b1 b2 b3 b4>\n";
118 }
119
120 #-- import router and AS topology information
121 print "(1) Importing topology information from files\n" if $show_info;
122 my $asfile = $ARGV[0]; # file containing AS relationships
123 my $rtopo = file_to_array($rfile,$show_info);
124 my $astopo = file_to_array($asfile,$show_info);
125 my $aslevels = file_to_array($asfile.".classified",$show_info);
126
127 #-- initialize AS and router graphs (if no rfile specified uses 1 router per AS)
128 print "\n(2) Initializing internal topology representation\n" if $show_info;
129 my $asgraph = Graph::Directed->new();
130 my $rgraph = Graph::Undirected->new(countvertexed=>1,countedged=>1);
```

```
131  init_asgraph($astopo,$asgraph,$aslevels,$show_info);
132  init_rgraph($rfile,$rtopo,$asgraph,$rgraph,$show_info);
133
134  #-- initialize discovery data (sources, destinations,..)
135  print "\n(3) Initializing discovery data\n" if $show_info;
136  my %discovery;
137  if ($cap_limit ne undef){ # CAPPED DISCOVERY (limited number of monitors per destination)
138          print "- Capping with $cap_limit monitors per destination\n" if $show_info;
139          generate_capped_discovery();
140  }
141  elsif ($nb_clusters ne undef){ # CLUSTERED DISCOVERY (common destination sets per cluster, without
         capping)
142          print "- Clustering into $nb_clusters clusters without capping\n" if $show_info;
143          generate_clustered_discovery();
144  }
145  else { # INDEPENDENT DISCOVERY (independent destination set per source, without clustering)
146          print "- Generating independent destinations sets for each source\n" if $show_info;
147          generate_independent_discovery();
148  }
149
150  #-- export discovery data to XML file
151  print "\n(4) Generating XML discovery data\n" if $show_info;
152  my $xml = XML::Simple->new(RootName=>'discovery', KeyAttr=>{cluster=>'id',src=>'ip'});
153  if ($export){
154          open (EXPORT, "> $export");
155          print EXPORT $xml->XMLout(\%discovery);
156          close EXPORT;
157  }
158  else {
159          print $xml->XMLout(\%discovery);
160  }
161
162  # -----[ generate_independent_discovery ]---------------------------------------------
163  # ------------------------------------------------------------------------------------
164
165  sub generate_independent_discovery {
166
167          # initialize arrays containing routers for random source and random destination selection
168          my @sources_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; # ref
                 to array of sorted routers
169          my @dests_by_level_and_type;
170          if ($exclude_s_as_dest eq undef){ # both arrays are independant, sources can be chosen as
                 destinations
171                  @dests_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; #
                         ref to array of sorted routers
172          }
173          else { # once sources are chosen, they get removed from both arrays, which makes their
                 destination selection impossible
174                  @dests_by_level_and_type = @sources_by_level_and_type;
175          }
176
177          #-- SOURCE SELECTION
178
179          # add sources to source set according to specified levels
180          my @selected_sources;
181          foreach my $t (0..NLABELS_RNODES-1){
182                  foreach my $l (0..NLEVELS-1){
183                          if ($sources_by_level_and_type[$l][$t] ne undef) { # check that current
                                 level isn't empty
184                                  if ($nb_sources[$t*NLEVELS+$l] <= scalar @{
                                         $sources_by_level_and_type[$l][$t]}){ # check sufficient
                                         sources are available
185                                          for (my $i=0;$i<$nb_sources[$t*NLEVELS+$l];$i++){ # add
                                                 number of routers specified at command-line
186                                                  my $random_index = rand(@{
                                                         $sources_by_level_and_type[$l][$t]}); #
                                                         random selection from router set
187                                                  push (@selected_sources,@{
                                                         $sources_by_level_and_type[$l][$t]}[
                                                         $random_index]); # add it to source set
```

```
188                                             splice(@{$sources_by_level_and_type[$l][$t]},
                                                    $random_index,1); # remove it from router set
189                                     }
190                                 }
191                                 else {
192                                     die "Error : maximum of ".scalar @{
                                            $sources_by_level_and_type[$l][$t]}. " sources
                                            available at type $t level $l\n";
193                                 }
194                             }
195                             elsif ($nb_sources[$t*NLEVELS+$l]!=0){
196                                 die "Error : no sources available at type $t level $l\n";
197                             }
198                         }
199                 }
200                 my $unique_source;
201                 if (scalar @selected_sources == 1){
202                         $unique_source=$selected_sources[0];
203                 }
204                 # distribute sources into single cluster, start by random cluster then modular loop
205                 while (@selected_sources) {
206                         if ($ipstring){
207                             $discovery{cluster}{0}{src}{int2address(pop(@selected_sources))}={}; # put
                                    data in main hash, leave value undefined for now
208                         }
209                         else {
210                             $discovery{cluster}{0}{src}{pop(@selected_sources)}={}; # put data in main
                                    hash, leave value undefined for now
211                         }
212                 }
213
214                 #-- DESTINATION SELECTION
215
216                 # add destination to destination set according to specified levels
217                 my @selected_dests;
218                 foreach my $t (0..NLABELS_RNODES-1){
219                         foreach my $l (0..NLEVELS-1){
220                             if ($dests_by_level_and_type[$l][$t] ne undef) { # check that current
                                    level isn't empty
221                                 if ($nb_dests[$t*NLEVELS+$l] <= scalar @{$dests_by_level_and_type[
                                        $l][$t]}){ # check sufficient dests are available
222                                     for (my $i=0;$i<$nb_dests[$t*NLEVELS+$l];$i++){ # add
                                            number of routers specified at command-line
223                                         my $random_index = rand(@{$dests_by_level_and_type
                                                [$l][$t]}); # random selection from router
                                                set
224                                         # if there is a single source and random dest is
                                                same as source, choose next destination in
                                                set (if exists)
225                                         if ($unique_source ne undef && (@{
                                                $dests_by_level_and_type[$l][$t]}[
                                                $random_index]==$unique_source)){
226                                             if (scalar @{$dests_by_level_and_type[$l][
                                                    $t]} > 1){
227                                                 $random_index=($random_index+1)%@{
                                                        $dests_by_level_and_type[$l][
                                                        $t]}; # add next destination
228                                             }
229                                             else {
230                                                 die "Error : too many destinations
                                                        specified at type $t level
                                                        $l for unique source\n";
231                                             }
232                                         }
233                                         push (@selected_dests,@{$dests_by_level_and_type[
                                                $l][$t]}[$random_index]); # add it to dest
                                                set
234                                         splice(@{$dests_by_level_and_type[$l][$t]},
                                                $random_index,1); # remove it from router set
235                                     }
236                                 }
```

```
237                                     else {
238                                             die "Error : maximum of ".scalar @{
                                                    $dests_by_level_and_type[$l][$t]}. " destinations
                                                    available at type $t level $l\n";
239                                     }
240                             }
241                             elsif ($nb_dests[$t*NLEVELS+$l]!=0){
242                                     die "Error : no destinations avilable at type $t level $l\n";
243                             }
244                     }
245             }
246
247             #-- DESTINATION-TO-SOURCE ASSIGNEMENT
248
249             # assign destinations to sources
250             while (@selected_dests) {
251                     foreach (keys(%{$discovery{cluster}{0}{src}})){
252                             if (@selected_dests) {
253                                     my $random_index = rand(@selected_dests); # select random
                                            destination from set
254                                     if ($selected_dests[$random_index]!=$_){ # don't add dest if same
                                            ip as current source (no self probe)
255                                             if ($ipstring){
256                                                     push (@{$discovery{cluster}{0}{src}{$_}{d}},
                                                            int2address($selected_dests[$random_index]));
                                                             # put data in main hash
257                                             }
258                                             else {
259                                                     push (@{$discovery{cluster}{0}{src}{$_}{d}},
                                                            $selected_dests[$random_index]); # put data
                                                            in main hash
260                                             }
261                                             splice(@selected_dests,$random_index,1); # remove it from
                                                    router set
262                                     }
263                             }
264                     }
265             }
266     }
267
268     # -----[ generate_clustered_discovery ]-----------------------------------------
269     # ------------------------------------------------------------------------------
270     sub generate_clustered_discovery {
271
272             # initialize arrays containing routers for random source and random destination selection
273             my @sources_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; # ref
                    to array of sorted routers
274             my @dests_by_level_and_type;
275             if ($exclude_s_as_dest eq undef){ # both arrays are independant, sources can be chosen as
                    destinations
276                     @dests_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; #
                            ref to array of sorted routers
277             }
278             else {
279                     @dests_by_level_and_type = @sources_by_level_and_type;
280             }
281
282             #-- SOURCE SELECTION
283
284             # add sources to source set according to specified levels
285             my @selected_sources;
286             foreach my $t (0..NLABELS_RNODES-1){
287                     foreach my $l (0..NLEVELS-1){
288                             if ($sources_by_level_and_type[$l][$t] ne undef) { # check that current
                                    level isn't empty
289                                     if ($nb_sources[$t*NLEVELS+$l] <= scalar @{
                                            $sources_by_level_and_type[$l][$t]}){ # check sufficient
                                            sources are available
290                                             for (my $i=0;$i<$nb_sources[$t*NLEVELS+$l];$i++){ # add
                                                    number of routers specified at command-line
```

```
291                                             my $random_index = rand(@{
                                                    $sources_by_level_and_type[$l][$t]}); #
                                                    random selection from router set
292                                             push (@selected_sources,@{
                                                    $sources_by_level_and_type[$l][$t]}[
                                                    $random_index]); # add it to source set
293                                             splice(@{$sources_by_level_and_type[$l][$t]},
                                                    $random_index,1); # remove it from router set
294                                     }
295                             }
296                             else {
297                                     die "Error : maximum of ".scalar @{
                                            $sources_by_level_and_type[$l][$t]}. " sources
                                            available at type $t level $l\n";
298                             }
299                     }
300                     elsif ($nb_sources[$t*NLEVELS+$l]!=0){
301                             die "Error : sources specified at empty type $t level $l\n";
302                     }
303             }
304     }
305
306
307     #-- DESTINATION SELECTION
308
309     # add destination to destination set according to specified levels
310     my @selected_dests;
311     foreach my $t (0..NLABELS_RNODES-1){
312             foreach my $l (0..NLEVELS-1){
313                     if ($dests_by_level_and_type[$l][$t] ne undef) { # check that current
                            level isn't empty
314                             if ($nb_dests[$t*NLEVELS+$l] <= scalar @{$dests_by_level_and_type[
                                    $l][$t]}){ # check sufficient dests are available
315                                     for (my $i=0;$i<$nb_dests[$t*NLEVELS+$l];$i++){ # add
                                            number of routers specified at command-line
316                                             my $random_index = rand(@{$dests_by_level_and_type
                                                    [$l][$t]}); # random selection from router
                                                    set
317                                             push (@selected_dests,@{$dests_by_level_and_type[
                                                    $l][$t]}[$random_index]); # add it to dest
                                                    set
318                                             splice(@{$dests_by_level_and_type[$l][$t]},
                                                    $random_index,1); # remove it from router set
319                                     }
320                             }
321                             else {
322                                     die "Error : maximum of ".scalar @{
                                            $dests_by_level_and_type[$l][$t]}. " destinations
                                            available at type $t level $l\n";
323                             }
324                     }
325                     elsif ($nb_dests[$t*NLEVELS+$l]!=0){
326                             die "Error : destinations specified at empty type $t level $l\n";
327                     }
328             }
329     }
330
331     #-- SOURCE-TO-CLUSTER AND DESTINATION-TO-CLUSTER ASSIGNEMENTS
332
333     # distribute sources into clusters, start by random cluster then modular loop
334     my $c = int rand($nb_clusters-1);
335     while (@selected_sources) {
336             my $random_index = rand(@selected_sources);
337             if ($ipstring){
338                     $discovery{cluster}{$c}{src}{int2address($selected_sources[$random_index])
                            }={}; # put data in main hash
339             }
340             else {
341                     $discovery{cluster}{$c}{src}{$selected_sources[$random_index]}={}; # put
                            data in main hash
342             }
```

```
343                     splice(@selected_sources,$random_index,1); # remove it from router set
344                     $c = ($c+1)%$nb_clusters;
345             }
346
347             # assign destinations to clusters, start by random cluster then modular loop
348             my $c = int rand($nb_clusters-1);
349             while (@selected_dests) {
350                     my $random_index = rand(@selected_dests);
351                     if ($ipstring){
352                             push (@{$discovery{cluster}{$c}{cdest}},int2address($selected_dests[
                                    $random_index])); # put data in main hash
353                     }
354                     else {
355                             push (@{$discovery{cluster}{$c}{cdest}},$selected_dests[$random_index]); #
                                    put data in main hash
356                     }
357                     splice(@selected_dests,$random_index,1); # remove it from router set
358                     $c = ($c+1)%$nb_clusters;
359             }
360     }
361
362     # -----[ generate_capped_discovery ]-------------------------------------------
363     # ---------------------------------------------------------------------------------
364     sub generate_capped_discovery {
365
366             # initialize arrays containing routers for random source and random destination selection
367             my @sources_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; # ref
                    to array of sorted routers
368             my @dests_by_level_and_type;
369             if ($exclude_s_as_dest eq undef){ # both arrays are independant, sources can be chosen as
                    destinations
370                     @dests_by_level_and_type = @{sort_routers_by_level_and_type($rgraph,$asgraph)}; #
                            ref to array of sorted routers
371             }
372             else {
373                     @dests_by_level_and_type = @sources_by_level_and_type;
374             }
375
376             #-- SOURCE SELECTION
377
378             # add sources to source set according to specified levels
379             my @selected_sources;
380             foreach my $t (0..NLABELS_RNODES-1){
381                     foreach my $l (0..NLEVELS-1){
382                             if ($sources_by_level_and_type[$l][$t] ne undef) { # check that current
                                    level isn't empty
383                                     if ($nb_sources[$t*NLEVELS+$l] <= scalar @{
                                            $sources_by_level_and_type[$l][$t]}){ # check sufficient
                                            sources are available
384                                             for (my $i=0;$i<$nb_sources[$t*NLEVELS+$l];$i++){ # add
                                                    number of routers specified at command-line
385                                                     my $random_index = rand(@{
                                                            $sources_by_level_and_type[$l][$t]}); #
                                                            random selection from router set
386                                                     push (@selected_sources,@{
                                                            $sources_by_level_and_type[$l][$t]}[
                                                            $random_index]); # add it to source set
387                                                     splice(@{$sources_by_level_and_type[$l][$t]},
                                                            $random_index,1); # remove it from router set
388                                             }
389                                     }
390                                     else {
391                                             die "Error : maximum of ".scalar @{
                                                    $sources_by_level_and_type[$l][$t]}." sources
                                                    available at type $t level $l\n";
392                                     }
393                             }
394                             elsif ($nb_sources[$t*NLEVELS+$l]!=0){
395                                     die "Error : sources specified at empty type $t level $l\n";
396                             }
397                     }
```

```
398         }
399
400         #-- DESTINATION SELECTION
401
402         # add destinations to destination set according to specified levels
403         my @selected_dests;
404         foreach my $t (0..NLABELS_RNODES-1){
405                 foreach my $l (0..NLEVELS-1){
406                         if ($dests_by_level_and_type[$l][$t] ne undef) { # check that current
                                level isn't empty
407                             if ($nb_dests[$t*NLEVELS+$l] <= scalar @{$dests_by_level_and_type[
                                    $l][$t]}){ # check sufficient dests are available
408                                 for (my $i=0;$i<$nb_dests[$t*NLEVELS+$l];$i++){ # add
                                        number of routers specified at command-line
409                                     my $random_index = rand(@{$dests_by_level_and_type
                                            [$l][$t]}); # random selection from router
                                            set
410                                     push (@selected_dests,@{$dests_by_level_and_type[
                                            $l][$t]}[$random_index]); # add it to dest
                                            set
411                                     splice(@{$dests_by_level_and_type[$l][$t]},
                                            $random_index,1); # remove it from router set
412                                 }
413                             }
414                             else {
415                                     die "Error : maximum of ".scalar @{
                                            $dests_by_level_and_type[$l][$t]}." destinations
                                            available at type $t level $l\n";
416                             }
417                         }
418                         elsif ($nb_dests[$t*NLEVELS+$l]!=0){
419                                 die "Error : destinations specified at empty type $t level $l\n";
420                         }
421                 }
422         }
423
424         #-- SOURCE-TO-CLUSTER, DESTINATION-TO-CLUSTER AND DESTINATION-TO-SOURCE ASSIGNEMENTS
425
426         # distribute sources into clusters, start by random cluster then modular loop
427         my $c = int rand($nb_clusters-1);
428         if ($nb_clusters eq undef){
429                 $nb_clusters=1;
430         }
431         while (@selected_sources) {
432                 my $random_index = rand(@selected_sources);
433                 if ($ipstring){
434                         $discovery{cluster}{$c}{src}{int2address($selected_sources[$random_index])
                                }={}; # put data in main hash
435                 }
436                 else {
437                         $discovery{cluster}{$c}{src}{$selected_sources[$random_index]}={}; # put
                                data in main hash
438                 }
439                 splice(@selected_sources,$random_index,1); # remove it from router set
440                 $c = ($c+1)%$nb_clusters;
441         }
442
443         # assign each randomly chosen destination to each cluster, then assign it to $cap_limit
                randomly chosen sources from this cluster
444         my $c = int rand($nb_clusters-1);
445         while (@selected_dests) {
446
447                 my @sources= keys %{$discovery{cluster}{$c}{src}};
448                 my @chosen_sources;
449
450                 if (scalar @sources > $cap_limit){
451                         @chosen_sources = random_n_from_array($cap_limit,\@sources);
452                 }
453                 else { # available sources <= cap limit (cap limit becomes number of monitors in
                        cluster)
454                         @chosen_sources = @sources;
```

```
455                     }
456
457                     my $random_dest_index = rand(@selected_dests); # choose dest randomly
458                     foreach (@chosen_sources){ # add it to each chosen source
459                             if ($ipstring){
460                                     push (@{$discovery{cluster}{$c}{src}{$_}{d}},int2address(
                                            $selected_dests[$random_dest_index])); # put data in main
                                            hash
461                             }
462                             else {
463                                     push (@{$discovery{cluster}{$c}{src}{$_}{d}},$selected_dests[
                                            $random_dest_index]); # put data in main hash
464                             }
465                     }
466                     splice(@selected_dests,$random_dest_index,1); # remove dest from set
467
468                     $c = ($c+1)%$nb_clusters; # goto next cluster
469             }
470     }
```

## E.3 Discovery simulation script – performDiscovery

```
1   #!/usr/bin/perl
2   # ==================================================================
3   # Discovery simulation script
4   # @(#)performDiscovery
5   # @author Gregory Culpin
6   # @date 08/12/2005
7   # @lastdate 12/05/2006
8   # ==================================================================
9
10  use strict;
11  use lib ".";
12  use CBGP 0.4;
13  use Math::BigInt;
14  use Data::Dumper;
15  use Graph::Directed;
16  use Graph::Undirected;
17  use Getopt::Long;
18  use constant {
19          # cbgp path
20          CBGP_PATH => 'cbgp',
21          # router-level node labels
22          NLABELS_RNODES => 2,
23          ROUTER_INTERNAL => 0,
24          ROUTER_BORDER => 1,
25          # router-level edge labels
26          NLABELS_RLINKS => 3,
27          RLINK_INTERNAL => 0,
28          RLINK_PP => 1,
29          RLINK_PC => 2,
30          # AS-level node labels
31          NLEVELS => 5,
32          AS_CORE => 0,
33          AS_TRANSIT => 1,
34          AS_OUTER => 2,
35          AS_ISP => 3,
36          AS_CUSTOMER => 4,
37          # router-level edge labels
38          NLABELS_ASLINKS => 2,
39          ASLINK_PP => 0,
40          ASLINK_PC => 1,
41          # result parameters
42          NPARAMETERS => 3,
43          COVERAGE => 0,
44          PROBED => 1,
45          PROBES => 2
```

```
46  };
47  use topology;
48  use tools;
49  use doubletree;
50  use XML::Simple;
51
52  # ------------------------------------------------------------------
53  # Main program
54  # ------------------------------------------------------------------
55
56  #-- manage command-line options
57
58  my $rfile=''; # file containing cbgp router level script
59
60  my $use_traceroute=''; # if option specified, use traceroutes algorithm
61  my $hop_eval='10'; # specifies number of destinations used to evaluate source hop
62  my $p='.05'; # specifies probability (%) of hitting a destination on first probe
63
64  my $max_propagations='50'; # max number of propagations before reloading cbgp process
65
66  my $bloom_capacity='1000';
67  my $bloom_error='.01';
68  my $nobloom='';
69
70  my $help=''; # show help
71  my $debug=''; # debug information
72  my $show_info=''; # generic information
73  my $show_detailed_results=''; # show detailed results
74
75  my $xml_export='';
76
77  GetOptions (
78                  # topology attributes
79                  'rfile|r=s' => \$rfile,
80
81                  # discovery attributes
82                  'traceroute|t' => \$use_traceroute,
83                  'hop-eval|h=i' => \$hop_eval,
84                  'p=f' => \$p,
85                  'bloom-capacity|c=i' => \$bloom_capacity,
86                  'bloom-error|e=f' => \$bloom_error,
87                  'nobloom|n' => \$nobloom,
88
89                  # cbgp related options
90                  'max-prop|m=i' => \$max_propagations,
91
92                  # display related options
93                  'help|?' => \$help,
94                  'debug' => \$debug,
95                  'show-info|info|i' => \$show_info,
96                  'show-detailed-results|d' => \$show_detailed_results,
97                  'export|x=s' => \$xml_export
98  );
99
100 if (@ARGV!=2 or $help) {
101         die "\nUsage: $0 [options] <as_topology_file> <xml_discovery_file>
102
103 Options:
104  --rfile (-r) <file>\t\t specifies the underlying cbgp router topology of given AS topology
105  --hop-eval (-h) <n>\t\t specifies the number of destinations used to evaluate initial hop (
         default:$hop_eval)
106  --hit-probability (-p) <f>\t specifies the probability p of hitting a destination on the first
         probe (default:$p)
107  --max-propagations (-m) <n>\t specifies the maximum number of prefixes to be propagated in cbgp
         before restarting it (default:$max_propagations)
108
109  --bloom-capacity (-c) <n>\t specifies the capacity of bloom filters used for encoding global sets
        (default:$bloom_capacity)
110  --bloom-error (-e) <f> \t specifies the error rate of bloom filters used for encoding global sets
        (default:$bloom_error)
111  --nobloom (-n)\t\t\t don't use bloom filters for encoding global sets
```

```
112
113   --traceroute (-t)\t\t don't use doubletree discovery but simple traceroutes
114   --export (-x <file>)\t\t exports the updated discovery state
115
116   --help (-?)\t\t\t this page
117   --show-info (-i)\t\t show information
118   --show-detailed-results (-d)\t show detailed result information
119   --debug\t\t\t show debug information\n"
120  }
121
122  my $n=1;
123  print "\n(".$n++.") Importing discovery data\n" if $show_info;
124  my $asfile = $ARGV[0]; # file containing AS relationships
125  my $dfile = $ARGV[1]; # XML file containing discovery plan
126  my $rtopo = file_to_array($rfile,$show_info);
127  my $astopo = file_to_array($asfile,$show_info);
128  my $aslevels = file_to_array($asfile.".classified");
129
130  #-- initialize AS and router graphs (if no rfile specified uses 1 router per AS)
131  print "\n(".$n++.") Initializing internal topology representation\n" if $show_info;
132  my $asgraph = Graph::Directed->new();
133  my $rgraph = Graph::Undirected->new(countvertexed=>1,countedged=>1);
134
135  init_asgraph($astopo,$asgraph,$aslevels,$show_info);
136  init_rgraph($rfile,$rtopo,$asgraph,$rgraph,$show_info);
137
138  #-- initialize discovery data (sources, destinations,..)
139  print "\n(".$n++.") Initializing discovery data\n" if $show_info;
140  my $xs1 = XML::Simple->new(forcearray=>1,suppressempty=>1,KeyAttr=>{cluster=>'id',src=>'ip'});
141  my $discovery = $xs1->XMLin($dfile);
142
143  #-- feed configuration and start simulation
144  print "\n(".$n++.") Setting up C-BGP simulator\n" if $show_info;
145  my $cbgp_ref = \CBGP->new(CBGP_PATH);
146  $$cbgp_ref->spawn;
147  die if $$cbgp_ref->send("set autoflush on\n");
148  cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
149
150  #-- run baby run
151  print "\n(".$n++.") Discovery process\n" if $show_info;
152  if (!$use_traceroute) {
153          print "- Evaluating initial hops for each source\n" if $show_info;
154          compute_hops($cbgp_ref,$discovery,$p,$hop_eval,$max_propagations,$asfile,$rfile,$rtopo,
                 $debug);
155          print "- Performing doubletree algorithm\n" if $show_info;
156          doubletree($cbgp_ref,$rgraph,$discovery,$rfile,$asfile,$rtopo,$max_propagations,
                 $bloom_capacity,$bloom_error,$nobloom,$debug);
157  }
158  else {
159          print "- Performing traceroutes\n" if $show_info;
160          traceroute($cbgp_ref,$rgraph,$discovery,$rfile,$asfile,$rtopo,$max_propagations,$debug);
161  }
162  $$cbgp_ref->finalize;
163  while (my $res= $$cbgp_ref->expect(0)){
164          die "Error: expect \"$res\"\n";
165  }
166
167  if ($xml_export){
168          print "\n(".$n++.") Exporting discovery state to $xml_export\n";
169          open (EXPORT, "> $xml_export");
170          print EXPORT $xs1->XMLout($discovery,RootName=>'discovery',KeyAttr=>{cluster=>'id',src=>'
                 ip'});
171          close EXPORT;
172  }
173  #-- compute discovery performance
174  print "\n(".$n++.") Computing discovery performance values\n" if $show_info;
175  compute_performance($rgraph,$discovery,$asgraph);
176  print "\n(".$n++.") Finished\n" if $show_info;
177
178  # -----[ compute_performance ]-----------------------------------------
179  # ---------------------------------------------------------------------
```

```
180 sub compute_performance
181     {
182         my ($rgraph,$sd,$asgraph) = @_;
183 
184         # GLOBAL INTERFACE AND LINK RESULTS
185         # compute number of unique interfaces and links probed during discovery (needed for
                coverage)
186         my $probed_ifs = 0;
187         my $probed_links = 0;
188         foreach ($rgraph->unique_vertices){
189                 if ($rgraph->get_vertex_count($_) > 1){
190                         $probed_ifs++;
191                 }
192         }
193         foreach ($rgraph->unique_edges){
194                 if ($rgraph->get_edge_count(@{$_}[0],@{$_}[1]) > 1){
195                         $probed_links++;
196                 }
197         }
198 
199         # count redundant sources in probed interfaces for coverage computation (are otherwise
                counted multiple times)
200         my $scount=0;
201         my $rcount=0;
202         foreach my $c (keys %{$sd->{cluster}}){
203                 foreach my $s (keys %{$sd->{cluster}{$c}{src}}){
204                         $scount++;
205                         # if current source has been probed at least once, do not consider it as
                                new discovery (=redundant)
206                         if ($rgraph->get_vertex_count($_)>1){
207                                 $rcount++;
208                         }
209                 }
210         }
211         # compute coverages such as known interfaces = src U disc = src+disc-(disc/\src), and
                store in data hash
212         # rem: number of interfaces exclude known sources and destinations
213         my $icov = ($scount + $probed_ifs - $rcount) / $rgraph->unique_vertices;
214         my $lcov= $probed_links / $rgraph->unique_edges;
215 
216         # compute total number of probes on vertices and edges, and store in data hash
217         # rem: number of probes excludes self-source probes, but includes destination probes
218         my $iprobes = $rgraph->vertices - $rgraph->unique_vertices;
219         my $lprobes = $rgraph->edges - $rgraph->unique_edges;
220 
221         # DETAILED ROUTER-LEVEL INTERFACE RESULTS
222         my @interface_results;
223         my @discovered_interfaces_per_level;
224 
225         # first pass : per-level computation of "probed" and "probes" values
226         foreach ($rgraph->unique_vertices){
227                 my $l = $asgraph->get_vertex_weight($_>>16);
228                 my $nb_probes = $rgraph->get_vertex_count($_)-1;
229                 my $c = $rgraph->get_vertex_weight($_); # internal or border
230                 if ($nb_probes>0){ # probed at least once
231                         $interface_results[PROBED][$c][$l]++; # increment "probed" counter by 1
232                         $interface_results[PROBES][$c][$l]+=$nb_probes; # increment "probes"
                                counter by nb_probes
233                 }
234                 $discovered_interfaces_per_level[$c][$l]++; # update level counter for coverage
                        computation
235         }
236 
237         # compute number of redundant sources in probed interfaces (needed for coverage)
238         my @scount_per_level_and_class;
239         my @rcount_per_level_and_class;
240 
241         foreach my $cl (keys %{$sd->{cluster}}){
242                 foreach my $s (keys %{$sd->{cluster}{$cl}{src}}){
243                         my $c = $rgraph->get_vertex_weight($s);
244                         my $l = $asgraph->get_vertex_weight($s>>16);
```

```
245                         my $nb_probes = $rgraph->get_vertex_count($s)-1;
246                         $scount_per_level_and_class[$c][$l]++;
247                         # if current source has been probed at least once, do not consider it as
                                new discovery (=redundant)
248                         if ($nb_probes>0){
249                                 $rcount_per_level_and_class[$c][$l]++;
250                         }
251                 }
252         }
253
254         # second pass : per-level and per-class "coverage" computation
255         foreach my $c (0..NLABELS_RNODES-1){ # for each router class
256                 if (exists $discovered_interfaces_per_level[$c]) { # check non empty class
257                         foreach my $l (0.. (scalar @{$discovered_interfaces_per_level[$c]}-1)) { #
                                for each AS level
258                                 if (!exists $interface_results[PROBED][$c][$l]){
259                                         $interface_results[PROBED][$c][$l]=0; # assign zero values
                                                to undefined "probed" param
260                                 }
261                                 if (!exists $interface_results[PROBES][$c][$l]){
262                                         $interface_results[PROBES][$c][$l]=0; # assign zero values
                                                to undefined "probes" param
263                                 }
264                                 if (exists $discovered_interfaces_per_level[$c][$l] &&
                                      $discovered_interfaces_per_level[$c][$l]!=0){ # compute
                                      coverage if current level contains interfaces
265                                     $interface_results[COVERAGE][$c][$l]=($interface_results[
                                          PROBED][$c][$l]+$scount_per_level_and_class[$c][$l]-
                                          $rcount_per_level_and_class[$c][$l]) /
                                          $discovered_interfaces_per_level[$c][$l];
266                                 }
267                                 else { # no routers in level (irrelevant)
268                                         $interface_results[COVERAGE][$c][$l]=undef;
269                                 }
270                         }
271                 }
272         }
273         # DETAILED ROUTER-LEVEL LINK RESULTS
274
275         #-- compute inter-level router links discovered
276         my @inter_level_discovery;
277
278         # init to all zeroes
279         foreach my $l1 (0..NLEVELS-1){
280                 foreach my $l2 (0..NLEVELS-1){
281                         foreach my $p (0..NLABELS_RLINKS-1){
282                                 $inter_level_discovery[$l1][$l2][$p]=0;
283                         }
284                 }
285         }
286         # compute level values
287         foreach ($rgraph->unique_edges){
288                 if ($rgraph->get_edge_count(@{$_}[0],@{$_}[1])>1){
289                         my $as1 = @{$_}[0]>>16;
290                         my $as2 = @{$_}[1]>>16;
291                         if ($as1==$as2){ # internal link
292                                 my $l = $asgraph->get_vertex_weight($as1);
293                                 $inter_level_discovery[$l][$l][RLINK_INTERNAL]++;
294                         }
295                         else { # inter-as link
296                                 # get level to which belongs each router
297                                 my $l1 = $asgraph->get_vertex_weight($as1);
298                                 my $l2 = $asgraph->get_vertex_weight($as2);
299                                 # rem : consider individual routers of an AS obeys its policies
300                                 if ($asgraph->has_edge($as1,$as2)){
301                                         if ($asgraph->has_edge($as2,$as1)){ # edge in AS graph is
                                              bidirectional as1<->as2
302                                             # add peer link twice (once for each router's
                                                  level)
303                                             $inter_level_discovery[$l1][$l2][RLINK_PP]+=0.5;
304                                             $inter_level_discovery[$l2][$l1][RLINK_PP]+=0.5;
```

```
305                                         }
306                                         else { # edge direction in AS graph is as1->as2
307                                                 $inter_level_discovery[$l1][$l2][RLINK_PC]++;
308                                         }
309                                 }
310                                 elsif ($asgraph->has_edge($as2,$as1)){ # edge direction in AS
                                        graph is as1<-as2
311                                         $inter_level_discovery[$l2][$l1][RLINK_PC]++;
312                                 }
313                         }
314                 }
315         }
316         # compute totals
317         foreach my $l1 (0..NLEVELS-1){ # per line totals, stored in extra column
318                 foreach my $l2 (0..NLEVELS-1){
319                         foreach my $p (0..NLABELS_RLINKS-1){
320                                 $inter_level_discovery[$l1][NLEVELS][$p]+=$inter_level_discovery[
                                        $l1][$l2][$p];
321                         }
322                 }
323         }
324         foreach my $l2 (0..NLEVELS){ # per column totals (including line sum), stored in extra
                line
325                 foreach my $l1 (0..NLEVELS-1){
326                         foreach my $p (0..NLABELS_RLINKS-1){
327                                 $inter_level_discovery[NLEVELS][$l2][$p]+=$inter_level_discovery[
                                        $l1][$l2][$p];
328                         }
329                 }
330         }
331
332         # DETAILED AS-LEVEL RESULTS
333         # DETAILED AS-LEVEL LINK RESULTS
334
335         print "- Global router-level results\n  icov:".$icov." lcov:".$lcov."\n";
336
337         if ($show_detailed_results) {
338
339                 print "- INTERFACE-LEVEL : interface discovery\n";
340                 foreach my $p (0..NPARAMETERS-1) { # for each parameter
341                         foreach my $c (0..NLABELS_RNODES-1) { # for each router class
342                                 if ($interface_results[$p][$c] != undef){
343                                         if ($p==COVERAGE){
344                                                 print "  coverage";
345                                         }
346                                         if ($p==PROBED){
347                                                 print "  probed  ";
348                                         }
349                                         if ($p==PROBES){
350                                                 print "  probes  ";
351                                         }
352                                         if ($c==ROUTER_INTERNAL){
353                                                 print "(i) ";
354                                         }
355                                         if ($c==ROUTER_BORDER){
356                                                 print "(b) ";
357                                         }
358                                         if ($discovered_interfaces_per_level[$c] != undef) { #
                                                check non empty class
359                                                 print "  ";
360                                                 foreach my $l (0.. (scalar @{
                                                        $discovered_interfaces_per_level[$c]}-1)){ #
                                                        for each AS level
361                                                    my $result = $interface_results[$p][$c][$l
                                                           ];
362                                                    if ($interface_results[COVERAGE][$c][$l]
                                                           eq undef) {
363                                                        print "/ ";
364                                                    }
365                                                    else {
366                                                        print "$result ";
```

```
367                                                     }
368                                                 }
369                                             }
370                                             print "\n";
371                                         }
372                                     }
373                                 }
374                                 print "- INTERFACE-LEVEL : inter-level link discovery (internal,pp,pc) performed
                                        with $lprobes lprobes\n";
375                                 foreach my $l1 (0..$#inter_level_discovery) { # for each level1
376                                     print "  ";
377                                     foreach my $l2 (0..$#{$inter_level_discovery[$l1]}) { # for each level2
378                                         foreach my $t (0..$#{$inter_level_discovery[$l1][$l2]}) { # for
                                                each link type (internal,PP,PC)
379                                             print $inter_level_discovery[$l1][$l2][$t];
380                                             if ($t!=$#{$inter_level_discovery[$l1][$l2]}){
381                                                 print ",";
382                                             }
383                                         }
384                                         print " \t";
385                                     }
386                                     print "\n";
387                                 }
388             }
389     }
```

## E.4 Doubletree algorithms module – `doubletree.pm`

```
1   #!/usr/bin/perl
2   # ==================================================================
3   # @(#)doubletree.pm
4   #
5   # @author Gregory Culpin
6   # @date 27/05/2006
7   # @lastdate 28/05/2006
8   #
9   # @version 0.1
10
11  package doubletree;
12
13  require Exporter;
14  @ISA= qw(Exporter);
15  @EXPORT= qw(traceroute
16              execute_traceroute
17              compute_hops
18              doubletree
19              execute_doubletree
20              );
21
22  use strict;
23  use warnings;
24  use lib ".";
25  use CBGP 0.4;
26  use topology;
27  use tools;
28  use Graph::Directed;
29  use Graph::Undirected;
30  use Data::Dumper;
31  use Math::BigInt;
32  use Bloom::Filter;
33
34  sub compute_hops
35      {
36          my ($cbgp_ref,$discovery,$p,$hop_eval,$max_propagations,$asfile,$rfile,$rtopo,$debug) = @_
                ;
37
38          my $memcount = 1;
```

```
39
40          # foreach cluster
41          foreach my $c (keys %{$discovery->{cluster}}){
42
43                  # foreach source
44                  foreach my $s (keys %{$discovery->{cluster}{$c}{src}}){
45
46                          # select random destinations
47                          my @random_dests;
48                          if (exists $discovery->{cluster}{$c}{cdest}) { # from common dest set
49                                  @random_dests = random_n_from_array($hop_eval,$discovery->{cluster
                                        }{$c}{cdest});
50                          }
51                          else { # from individual dest set
52                                  if (exists $discovery->{cluster}{$c}{src}{$s}{d}){ # test for
                                        sources with empty dest sets
53                                          @random_dests = random_n_from_array($hop_eval,$discovery
                                                ->{cluster}{$c}{src}{$s}{d});
54                                  }
55                          }
56
57                          my $dcount = 1; # destination counter for responding destinations
58                          my @distribution; # hop distribution array containing hops of responding
                                destinations
59
60                          # update hop distribution array with the path lengths from current source
                                to each destination
61                          foreach my $d (@random_dests){
62                                  # don't self probe
63                                  if ($s!=$d) {
64                                          # propagate current destination prefix in network
65                                          print "      (".$dcount.") adding prefix ".int2address($d)
                                                ."/16 and propagating into network\n" if $debug;
66                                          $$cbgp_ref->send("bgp router ".int2address($d)." add
                                                network ".int2address($d)."/16\n");
67                                          $$cbgp_ref->send("sim run\n");
68                                          $$cbgp_ref->send("bgp topology clear-ribs\n"); # free up
                                                memory by clearing out ribs
69                                          cbgp_check($cbgp_ref);
70
71                                          # send traceroute
72                                          $$cbgp_ref->send("net node ".int2address($s)." record-
                                                route ".int2address($d)."\n");
73                                          # wait for answer and process
74                                          my $response= $$cbgp_ref->expect(1);
75
76                                          # answer format : src, dest, status, hop0, hop1, hop2, ...
77                                          my @answer= split /\s+/, $response;
78                                          # only take replying destinations into account
79                                          print "       traceroute : ".$response."\n" if $debug;
80                                          print "       path length : ".(scalar @answer - 4)."\n" if
                                                $debug;
81                                          if ($answer[2] eq "SUCCESS") {
82                                                  splice(@answer, 0, 4); # contains hop1, hop2, ...
83                                                  $distribution[scalar @answer]++; # update
                                                        distribution array
84                                                  $dcount++;
85                                          }
86
87                                          # if maximum memory count is reached finalize and
                                                reinitialize current cbgp process
88                                          if ($memcount==$max_propagations) {
89                                                  print "           --- memcount reached, reloading
                                                        cbgp simulator ---\n" if $debug;
90
91                                                  cbgp_reset($cbgp_ref);
92                                                  cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
93                                                  $memcount=1;
94                                          }
95                                          else {
96                                                  $memcount++;
```

```
97                                      }
98                              }
99                      }
100
101                     # compute cumulative array of hops
102                     my @cumulative;
103                     foreach my $i (1..(scalar @distribution -1)){
104                             foreach (1..$i) {
105                                     if (exists $distribution[$_]){
106                                             $cumulative[$i]+=$distribution[$_];
107                                             }
108                             }
109                             if (exists $cumulative[$i]){
110                                     $cumulative[$i]/=($dcount-1);
111                             }
112                     }
113
114                     # choose hop such as there is a probability p% of hitting a destination on
                              the first hit
115                     my $hop=1;
116                     for (my $i=1;$i<scalar @cumulative;$i++){ # for each hop i
117                             if (exists $cumulative[$i] && $cumulative[$i] <= $p){ # if the
                                      cumulative proportion of destinations at hop i is smaller
                                      than p
118                                     $hop = $i; # update hop
119                                     print "       ".$cumulative[$i]." <= ".($p).", ok\n" if
                                              $debug;
120                             }
121                             else { # starting at hop i would hit a larger proportion of
                                      destinations than p, keep previous hop
122                                     print "       ".$cumulative[$i]." >= ".($p).", breaking and
                                              choosing hop ".($hop)."\n" if $debug;
123                                     last;
124                             }
125                     }
126                     # set initial hop for current source
127                     $discovery->{cluster}{$c}{src}{$s}{hop}=$hop;
128             }
129     }
130     # reset cbgp simulator before discovery (or keep track of nb of propagations)
131     cbgp_reset($cbgp_ref);
132     cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
133 }
134
135 # -----[ traceroute ]-----------------------------------------------
136 # ------------------------------------------------------------------
137
138 sub traceroute
139     {
140         my ($cbgp_ref,$rgraph,$discovery,$rfile,$asfile,$rtopo,$max_propagations,$debug) = @_;
141
142         my $memcount = 1; # memory counter to limit number of propagations before cbgp reset
143
144         foreach my $c (keys %{$discovery->{cluster}}){
145
146                 # if current cluster has a common destination set : cycle on destinations then on
                          sources
147                 if (exists $discovery->{cluster}{$c}{cdest}) {
148
149                         print "   <CLUSTER ".$c."> performing common destination set discovery\n"
                                  if $debug;
150                         my $dcount = 1; # destination counter for display
151
152                         #-- FOR EACH DESTINATION
153                         foreach my $d (@{$discovery->{cluster}{$c}{cdest}}){
154
155                                 my $success = 0; # number of successful traceroutes
156                                 my $failure = 0; # number of failed traceroutes
157
158                                 # propagate current destination prefix in network
```

```
159                         print "      (".$dcount.") adding prefix ".int2address($d)."/16
                                and propagating into network\n" if $debug;
160                         $$cbgp_ref->send("bgp router ".int2address($d)." add network ".
                                int2address($d)."/16\n");
161                         $$cbgp_ref->send("sim run\n");
162                         $$cbgp_ref->send("bgp topology clear-ribs\n"); # free up memory by
                                clearing out ribs
163                         cbgp_check($cbgp_ref);
164
165                         #-- FROM EACH SOURCE
166                         foreach my $s (keys %{$discovery->{cluster}{$c}{src}}){
167                                 # perform single traceroute call (1 source to 1
                                        destination), router graph updated with probed
                                        interfaces
168                                 my ($curr_success,$curr_failure) = execute_traceroute(
                                        $cbgp_ref,$rgraph,$s,$d,$debug);
169                                 $success+=$curr_success;
170                                 $failure+=$curr_failure;
171                         }
172
173                         # display success statistics for traceroutes towards current
                                destination
174                         print "      (".$dcount++.") performed $success out of ".($success
                                +$failure)." probing scheme(s) successfully towards ".
                                int2address($d)."\n" if $debug;
175                         # if maximum memory count is reached finalize and reinitialize
                                current cbgp process
176                         if ($memcount==$max_propagations) {
177                                 print "          --- memcount reached, reloading cbgp
                                        simulator ---\n" if $debug;
178                                 cbgp_reset($cbgp_ref);
179                                 cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
180
181                                 $memcount=1;
182                         }
183                         else {
184                                 $memcount++;
185                         }
186                 }
187         }
188
189         # destination sets are individual : cycle on sources, then on destinations
190         else {
191
192                 print "   <CLUSTER ".$c."> performing individual destination set discovery
                        \n" if $debug;
193                 my $scount = 1; # source counter for display
194
195                 #-- FROM EACH SOURCE
196                 foreach my $s (keys %{$discovery->{cluster}{$c}{src}}){
197
198                         my $dcount = 1; # destination count for print
199                         my $success = 0; # number of successful traceroutes
200                         my $failure = 0; # number of failed traceroutes
201
202                         #-- EXECUTE TRACEROUTE TO EACH DESTINATION
203                         foreach my $d (@{$discovery->{cluster}{$c}{src}{$s}{d}}){
204
205                                 # propagate current destination prefix in network
206                                 print "      (".$dcount++.") adding prefix ".int2address(
                                        $d)."/16 and propagating into network\n" if $debug;
207                                 $$cbgp_ref->send("bgp router ".int2address($d)." add
                                        network ".int2address($d)."/16\n");
208                                 $$cbgp_ref->send("sim run\n");
209                                 $$cbgp_ref->send("bgp topology clear-ribs\n"); # free up
                                        memory by clearing out ribs
210                                 cbgp_check($cbgp_ref);
211
212                                 # perform single traceroute call (1 source to 1
                                        destination), router graph updated with probed
                                        interfaces
```

```
213                                     my ($curr_success,$curr_failure) = execute_traceroute(
                                            $cbgp_ref,$rgraph,$s,$d,$debug);
214                                     $success+=$curr_success;
215                                     $failure+=$curr_failure;
216
217                                     # if maximum memory count is reached finalize and
                                            reinitialize current cbgp process
218                                     if ($memcount==$max_propagations) {
219                                             print "          --- memcount reached, reloading
                                                    cbgp simulator ---\n" if $debug;
220                                             cbgp_reset($cbgp_ref);
221                                             cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
222                                             $memcount=1;
223                                     }
224                                     else {
225                                             $memcount++;
226                                     }
227                             }
228
229                             # display success statistics for current source
230                             print "      (".$scount++.") Performed $success out of ".($success
                                    +$failure)." probing scheme(s) successfully from ".
                                    int2address($s)."\n" if $debug;
231                     }
232             }
233     }
234 }
235
236 sub execute_traceroute # from a specific source to a specific destination
237     {
238         my ($cbgp_ref,$rgraph,$s,$d,$debug) = @_;
239
240         my $success = 0;
241         my $failure = 0;
242
243         if ($d ne $s){ # don't self probe
244
245                 # send traceroute command to simulator
246                 $$cbgp_ref->send("net node ".int2address($s)." record-route ".int2address($d)."\n"
                        );
247                 # wait for answer and process
248                 my $response= $$cbgp_ref->expect(1);
249                 # answer format : src, dest, status, hop0, hop1, hop2, ...
250                 my @answer= split /\s+/, $response;
251
252                 print "          * traceroute : ".$response."\n" if $debug;
253
254                 # check traceroute status
255                 if ($answer[2] eq "SUCCESS") {
256                         # if successful, add to array each ip appearing after initial hop (from
                                index h+3 onwards)
257                         splice(@answer, 0, 3); # array contains hop0, hop1, ...
258
259                         for (my $i=1; $i <= $#answer; $i++){
260                                 # each interface from traceroute is added to graph by incrementing
                                        vertex count
261                                 # current source should not be considered as a probed interface
262                                 # as it would false the number of probes that may come from other
                                        sources
263                                 $rgraph->add_vertex(address2int($answer[$i])); # probe interface
                                        at hop i
264
265                                 # each link from traceroute, from (hop0,hop1) to (hopN-1,hopN),
                                        has its edge count incremented
266                                 $rgraph->add_edge(address2int($answer[$i-1]),address2int($answer[
                                        $i])); # probe interface at hop i
267
268                         }
269                         $success++;
270                 }
271                 # if unreachable or else, add partial traceroutes?
```

```
272                 else {
273                         $failure++;
274                 }
275         }
276         else {
277                 # ignore self probe, don't account as failure
278         }
279         return ($success,$failure);
280 }
281
282 # -----[ doubletree ]-----------------------------------------------
283 # ------------------------------------------------------------------
284
285 sub doubletree
286     {
287         my ($cbgp_ref,$rgraph,$discovery,$rfile,$asfile,$rtopo,$max_propagations,$bloom_capacity,
                $bloom_error,$nobloom,$debug) = @_;
288
289         my $memcount = 1; # memory counter to limit number of propagations before cbgp reset
290
291         foreach my $c (keys %{$discovery->{cluster}}){
292
293                 my $bf;
294
295                 # initialize bloom filter if used
296                 if (!$nobloom){
297                         $discovery->{cluster}{$c}{bloom}=Bloom::Filter->new(capacity=>
                                $bloom_capacity,error_rate=>$bloom_error);
298                         print "BLOOM filter length : ".$discovery->{cluster}{$c}{bloom}->length."\
                                n" if $debug;
299                 }
300
301                 # if current cluster has a common destination set : cycle on destinations then on
                        sources
302                 if (exists $discovery->{cluster}{$c}{cdest}) {
303
304                         print "   <CLUSTER ".$c."> performing common destination set discovery\n"
                                if $debug;
305                         my $dcount = 1; # destination counter for display
306
307                         #-- FOR EACH DESTINATION
308                         foreach my $d (@{$discovery->{cluster}{$c}{cdest}}){
309
310                                 my $success = 0; # number of successful doubletree calls
311                                 my $failure = 0; # number of failed doubletree calls (unsuccessful
                                        traceroute or hop beyond destination)
312
313                                 # propagate current destination prefix in network
314                                 print "      (".$dcount.") adding prefix ".int2address($d)."/16
                                        and propagating into network\n" if $debug;
315                                 $$cbgp_ref->send("bgp router ".int2address($d)." add network ".
                                        int2address($d)."/16\n");
316                                 $$cbgp_ref->send("sim run\n");
317                                 $$cbgp_ref->send("bgp topology clear-ribs\n"); # free up memory by
                                        clearing out ribs
318                                 cbgp_check($cbgp_ref);
319
320                                 #-- FROM EACH SOURCE
321                                 foreach my $s (keys %{$discovery->{cluster}{$c}{src}}){
322                                         # perform single doubletree call (1 source to 1
                                                destination), router graph updated with probed
                                                interfaces
323                                         my ($curr_success,$curr_failure) =  execute_doubletree(
                                                $cbgp_ref,$discovery,$rgraph,$c,$s,$d,$discovery->{
                                                cluster}{$c}{src}{$s}{hop},$nobloom,$debug);
324                                         $success+=$curr_success;
325                                         $failure+=$curr_failure;
326                                 }
327
328                                 # display success statistics for traceroutes towards current
                                        destination
```

```
329                             print "      (".$dcount++.") performed $success out of ".($success
                                    +$failure)." probing scheme(s) successfully towards ".
                                    int2address($d)."\n" if $debug;
330                             # if maximum memory count is reached finalize and reinitialize
                                    current cbgp process
331                             if ($memcount==$max_propagations) {
332                                     print "          --- memcount reached, reloading cbgp
                                            simulator ---\n" if $debug;
333                                     cbgp_reset($cbgp_ref);
334                                     cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
335
336                                     $memcount=1;
337                             }
338                             else {
339                                     $memcount++;
340                             }
341                     }
342             }
343
344             # destination sets are individual : cycle on sources, then on destinations
345             else {
346
347                     print "   <CLUSTER ".$c."> performing individual destination set discovery
                            \n" if $debug;
348                     my $scount = 1; # source counter for display
349
350                     #-- FROM EACH SOURCE
351                     foreach my $s (keys %{$discovery->{cluster}{$c}{src}}){
352
353                             my $dcount = 1; # destination count for print
354                             my $success = 0; # number of successful traceroutes
355                             my $failure = 0; # number of failed traceroutes
356
357                             #-- EXECUTE TRACEROUTE TO EACH DESTINATION
358                             foreach my $d (@{$discovery->{cluster}{$c}{src}{$s}{d}}){
359
360                                     # propagate current destination prefix in network
361                                     print "      (".$dcount++.") adding prefix ".int2address(
                                            $d)."/16 and propagating into network\n" if $debug;
362                                     $$cbgp_ref->send("bgp router ".int2address($d)." add
                                            network ".int2address($d)."/16\n");
363                                     $$cbgp_ref->send("sim run\n");
364                                     $$cbgp_ref->send("bgp topology clear-ribs\n"); # free up
                                            memory by clearing out ribs
365                                     cbgp_check($cbgp_ref);
366
367                                     # perform single traceroute call (1 source to 1
                                            destination), router graph updated with probed
                                            interfaces
368                                     my ($curr_success,$curr_failure) = execute_doubletree(
                                            $cbgp_ref,$discovery,$rgraph,$c,$s,$d,$discovery->{
                                            cluster}{$c}{src}{$s}{hop},$nobloom,$debug);
369                                     $success+=$curr_success;
370                                     $failure+=$curr_failure;
371
372                                     # if maximum memory count is reached finalize and
                                            reinitialize current cbgp process
373                                     if ($memcount==$max_propagations) {
374                                             print "          --- memcount reached, reloading
                                                    cbgp simulator ---\n" if $debug;
375                                             cbgp_reset($cbgp_ref);
376                                             cbgp_setup($cbgp_ref,$asfile,$rfile,$rtopo);
377                                             $memcount=1;
378                                     }
379                                     else {
380                                             $memcount++;
381                                     }
382                             }
383
384                             # display success statistics for current source
```

```
385                                     print "      (".$scount++.") Performed $success out of ".($success
                                            +$failure)." probing scheme(s) successfully from ".
                                            int2address($s)."\n" if $debug;
386                             }
387                     }
388             }
389     }
390
391     sub execute_doubletree # from a specific source to a specific destination
392     {
393             my ($cbgp_ref,$discovery,$rgraph,$c,$s,$d,$hop,$nobloom,$debug) = @_;
394
395             my $success = 0;
396             my $failure = 0;
397
398             if ($d ne $s){ # don't self probe
399
400                     # send traceroute command to simulator
401                     $$cbgp_ref->send("net node ".int2address($s)." record-route ".int2address($d)."\n"
                            );
402                     # wait for answer and process
403                     my $response = $$cbgp_ref->expect(1);
404                     print "          * traceroute : ".$response."\n" if $debug;
405                     # answer format : src, dest, status, hop0, hop1, hop2, ...
406                     my @answer = split /\s+/, $response;
407
408                     # check if hop h (at index h+3) is not beyond destination (at last index)
409                     if (3+$hop <= $#answer){
410                             if ($answer[2] eq "SUCCESS") {
411                                     # remove first three fields which are useless
412                                     splice(@answer, 0, 3);
413
414                                     # forward probing starting from hop to destination
415                                     foreach ($hop..$#answer) {
416
417                                             # probe interface and link at current hop
418                                             $rgraph->add_vertex(address2int($answer[$_])); #
                                                    discovered interface
419                                             print "FORWARD PROBE :   probed ".$answer[$_]." at h=".$_.
                                                    "\n" if $debug;
420                                             $rgraph->add_edge(address2int($answer[$_-1]),address2int(
                                                    $answer[$_])); # discovered link (preceding interface
                                                    )
421                                             print "                  probed (".$answer[$_-1].",".
                                                    $answer[$_].") between h=".($_-1)." and h=".$_."\n"
                                                    if $debug;
422
423                                             # if current interface != destination and not yet
                                                    contained in global set, add it
424                                             if (address2int($answer[$_])!=$d) {
425
426                                                     if ($nobloom){ # don't use bloom filter for global
                                                            set
427                                                             if (!exists $discovery->{cluster}{$c}{
                                                                    global_set}{link2bigint($answer[$_],
                                                                    int2address($d),0)}) {
428                                                                     $discovery->{cluster}{$c}{
                                                                            global_set}{link2bigint(
                                                                            $answer[$_],int2address($d)
                                                                            ,0)}++; # ip couple treated
                                                                            as asymmetrical link
429                                                                     print "                  adding ("
                                                                            .$answer[$_].",".int2address(
                                                                            $d).") to global set\n" if
                                                                            $debug;
430                                                             }
431                                                             else { # is already part of global set
432                                                                     print "                  global
                                                                            set member hit, not added.\n"
                                                                             if $debug;
433                                                                     last;
```

```
434                                         }
435                                     }
436                                     else { # use bloom filter for global set
437                                         if (!$discovery->{cluster}{$c}{bloom}->
                                                check(link2bigint($answer[$_],
                                                int2address($d),0))) {
438                                             $discovery->{cluster}{$c}{bloom}->
                                                    add(link2bigint($answer[$_],
                                                    int2address($d),0)); # ip
                                                    couple treated as
                                                    asymmetrical link
439                                             print "                  adding ("
                                                    .$answer[$_].",".int2address(
                                                    $d).") to global set\n" if
                                                    $debug;
440                                         }
441                                         else { # is already part of global set
442                                             print "                  global
                                                    set member hit, not added.\n"
                                                     if $debug;
443                                             last;
444                                         }
445
446                                     }
447                                 }
448                                 else { # is destination hit
449                                     print "                  destination hit, not
                                            added.\n" if $debug;
450                                     last;
451                                 }
452
453                             }
454
455                             # trace backwards from hop to source+1
456                             foreach (reverse (1..$hop-1)) {
457
458                                 # probe interface and link at current hop
459                                 $rgraph->add_vertex(address2int($answer[$_])); #
                                        discovered interface
460                                 print "BACKWARD PROBE :  probed ".$answer[$_]." at h=".$_.
                                        "\n" if $debug;
461                                 $rgraph->add_edge(address2int($answer[$_-1]),address2int(
                                        $answer[$_]));
462                                 print "                  probed (".$answer[$_-1].",".
                                        $answer[$_].") between h=".($_-1)." and h=".$_."\n"
                                        if $debug;
463                                 # if current interface not yet contained in local set, add
                                        it to both local and global sets
464                                 if (!exists $discovery->{cluster}{$c}{src}{$s}{local_set}{
                                        address2int($answer[$_])}) {
465                                     $discovery->{cluster}{$c}{src}{$s}{local_set}{
                                            address2int($answer[$_])}++;
466                                     print "                  adding (".$answer[$_].")
                                            to local set\n" if $debug;
467
468                                     if ($nobloom){ # don't use bloom filter for global
                                            set
469                                         $discovery->{cluster}{$c}{global_set}{
                                                link2bigint($answer[$_],int2address(
                                                $d),0)}++;
470                                         print "                  adding (".$answer
                                                [$_].",".int2address($d).") to global
                                                 set\n" if $debug;
471                                     }
472                                     else { # use bloom filter for global set
473                                         $discovery->{cluster}{$c}{bloom}->add(
                                                link2bigint($answer[$_],int2address(
                                                $d),0)); # ip couple treated as
                                                asymmetrical link
474                                         print "                  adding (".$answer
                                                [$_].",".int2address($d).") to global
```

```
                                                                    set\n" if $debug;
475                                                         }
476                                                 }
477                                                 else { # else is already part of local set
478                                                         print "                 local set member hit, not
                                                                added.\n" if $debug;
479                                                         last;
480                                                 }
481                                                 if (address2int($answer[$_-1])==$s) { # is interface at
                                                        hop 1
482                                                         print "                 source next-hop hit.\n"
                                                                if $debug;
483                                                 }
484                                         }
485
486                                         $success++;
487                                 }
488                                 # if traceroute unsuccessful for any reason (TODO add partial traceroutes
                                        ?)
489                                 else {
490                                         print "Failure : traceroute unsuccessful\n" if $debug;
491                                         $failure++;
492                                 }
493                         }
494                         # hop h is beyond destination, adapt value by halving it
495                         elsif ($hop!=1) {
496                                 print "Failure : hop beyond destination. re-executing with hop = ".int((
                                        $hop/2)+0.5)."\n" if $debug;
497                                 my ($inc_success,$inc_failure) = execute_doubletree($cbgp_ref,$discovery,
                                        $rgraph,$c,$s,$d,int(($hop/2)+0.5),$nobloom,$debug);
498
499                                 $success+=$inc_success;
500                                 $failure+=$inc_failure+1;
501                         }
502                         # god knows what hop problem
503                         else {
504                                 print "Failure : unknown\n" if $debug;
505                                 $failure++;
506                         }
507                 }
508                 else {
509                         # ignore self probe, don't account as failure
510                 }
511                 return ($success,$failure);
512         }
```

## E.5 Topology model module – topology.pm

```
1   #!/usr/bin/perl
2   # ==================================================================
3   # Topology module providing initialization and sorting subroutines
4   # @(#)topology.pm
5   # @author Gregory Culpin
6   # @date 27/05/2006
7   # @lastdate 28/05/2006
8
9   package topology;
10
11  require Exporter;
12  @ISA= qw(Exporter);
13  @EXPORT= qw(init_asgraph
14              init_rgraph
15              sort_ases_by_level
16              sort_routers_by_level_and_type
17              );
18  use strict;
19  use tools;
```

```
20  use lib ".";
21  use Math::BigInt;
22  use Data::Dumper;
23  use Graph::Directed;
24  use Graph::Undirected;
25  use constant {
26          # router-level node labels
27          NLABELS_RNODES => 2,
28          ROUTER_INTERNAL => 0,
29          ROUTER_BORDER => 1,
30          # router-level edge labels
31          NLABELS_RLINKS => 3,
32          RLINK_INTERNAL => 0,
33          RLINK_PP => 1,
34          RLINK_PC => 2,
35          # AS-level node labels
36          NLEVELS => 5,
37          AS_CORE => 0,
38          AS_TRANSIT => 1,
39          AS_OUTER => 2,
40          AS_ISP => 3,
41          AS_CUSTOMER => 4,
42          # router-level edge labels
43          NLABELS_ASLINKS => 2,
44          ASLINK_PP => 0,
45          ASLINK_PC => 1,
46          # result parameters
47          NPARAMETERS => 3,
48          COVERAGE => 0,
49          PROBED => 1,
50          PROBES => 2,
51  };
52
53  # -----[ init_asgraph ]---------------------------------------------
54  # ------------------------------------------------------------------
55  sub init_asgraph {
56
57          my ($astopo,$asgraph,$aslevels,$show_info) = @_;
58
59          my @level_counter; # for display
60
61          # initialize edges
62          foreach (@$astopo) {
63                  my @line= split /\s+/;
64                  # add each AS as vertex and relationship as edge
65                  if (scalar(@line) == 3) { # input format [as1 as2 relationship] (0:PP,-1:CP,1:PC)
66                          if ($line[2]==0) { # peer-to-peer relationship is represented by two
                                directional edges
67                                  $asgraph->add_edge($line[1],$line[0]);
68                                  $asgraph->add_edge($line[0],$line[1]);
69                          }
70                          elsif ($line[2]==-1) { # customer-to-provider relationship represented by
                                a directional edge
71                                  $asgraph->add_edge($line[1],$line[0]);
72                          }
73                          elsif ($line[2]==1) { # provider-to-customer relationship represented by a
                                directional edge
74                                  $asgraph->add_edge($line[0],$line[1]);
75                          }
76                          else {
77                                  die "relationship file has syntax errors (wrong value for 3rd
                                        argument)\n";
78                          }
79                  }
80                  else {
81                          die "relationship file has syntax errors (wrong number of arguments per
                                line)\n";
82                  }
83          }
84
85          # set weight of each AS according to its classification
```

```
86          foreach (@$aslevels) {
87                  my @line= split /\s+/;
88                  if (scalar(@line) == 2) { # input format [as level]
89                          $asgraph->set_vertex_weight($line[0],$line[1]);
90                          $level_counter[$line[1]]++;
91                  }
92          }
93
94          # display information (and don't compute if no display)
95          if ($show_info){
96
97                  print "- AS-LEVEL : per-level distribution\n  ";
98                  foreach my $l (0..NLEVELS-1){ # for each level
99                          if ($level_counter[$l] != undef) {
100                                 print $level_counter[$l]."\t";
101                         }
102                         else {
103                                 print "0\t";
104                         }
105                 }
106                 print "\n";
107
108                 #-- inter-level AS connectivity
109                 my @inter_level_as_connectivity;
110
111                 # init array to all zeroes
112                 foreach my $l1 (0..NLEVELS-1){
113                         foreach my $l2 (0..NLEVELS-1){
114                                 foreach my $p (0..1){
115                                         $inter_level_as_connectivity[$l1][$l2][$p]=0;
116                                 }
117                         }
118                 }
119                 # compute values
120                 foreach ($asgraph->unique_edges){
121                         my $as1 = @{$_}[0];
122                         my $as2 = @{$_}[1];
123
124                         # get level to which belongs each AS
125                         my $l1 = $asgraph->get_vertex_weight($as1);
126                         my $l2 = $asgraph->get_vertex_weight($as2);
127
128                         if ($asgraph->has_edge($as1,$as2)){
129                                 if ($asgraph->has_edge($as2,$as1)){ # edge in AS graph is
                                        bidirectional as1<->as2
130                                         # add peer link twice 1/2 (once for each router's level)
131                                         $inter_level_as_connectivity[$l1][$l2][ASLINK_PP]+=0.5;
132                                         $inter_level_as_connectivity[$l2][$l1][ASLINK_PP]+=0.5;
133                                 }
134                                 else { # edge direction in AS graph is as1->as2
135                                         $inter_level_as_connectivity[$l1][$l2][ASLINK_PC]++;
136                                 }
137                         }
138                         elsif ($asgraph->has_edge($as2,$as1)){ # edge direction in AS graph is as1
                                <-as2
139                                 $inter_level_as_connectivity[$l2][$l1][ASLINK_PC]++;
140                         }
141                 }
142                 # compute totals
143                 foreach my $l1 (0..NLEVELS-1){ # per line totals, stored in extra column
144                         foreach my $l2 (0..NLEVELS-1){
145                                 foreach my $p (0..1){
146                                         $inter_level_as_connectivity[$l1][NLEVELS][$p]+=
                                                $inter_level_as_connectivity[$l1][$l2][$p];
147                                 }
148                         }
149                 }
150                 foreach my $l2 (0..NLEVELS){ # per column totals (including line sum), stored in
                        extra line
151                         foreach my $l1 (0..NLEVELS-1){
152                                 foreach my $p (0..1){
```

```
153                                         $inter_level_as_connectivity[NLEVELS][$l2][$p]+=
                                                $inter_level_as_connectivity[$l1][$l2][$p];
154                                 }
155                         }
156                 }
157                 # display (rem: peering relationships are double edges so are counted twice)
158                 print "- AS-LEVEL : inter-level connectivity (pp,pc)\n";
159                 foreach my $l1 (0..$#inter_level_as_connectivity) { # for each level1
160                         print "  ";
161                         foreach my $l2 (0..$#{$inter_level_as_connectivity[$l1]}) { # for each
                                level2
162                                 foreach my $t (0..$#{$inter_level_as_connectivity[$l1][$l2]}) { #
                                        for each link type (PP,PC)
163                                         print $inter_level_as_connectivity[$l1][$l2][$t];
164                                         if ($t!=$#{$inter_level_as_connectivity[$l1][$l2]}){
165                                                 print ",";
166                                         }
167                                 }
168                                 print " \t";
169                         }
170                         print "\n";
171                 }
172         }
173 }
174
175 # -----[ init_rgraph ]-----------------------------------------------
176 # -------------------------------------------------------------------
177 sub init_rgraph {
178
179         my ($rfile,$rtopo,$asgraph,$rgraph,$show_info) = @_;
180
181         # extract routers and links from topology
182         my $interfaces = get_interfaces($rtopo);
183         my $links = get_links($rtopo);
184
185         # initialize undirected graph of routers
186
187         if ($rfile ne '') { # extract from router topology if available
188                 # add each unique ip address to graph
189                 foreach (@$interfaces){
190                         $rgraph->add_vertex($_);
191                 }
192
193                 # add each unique link to graph
194                 foreach (@$links){
195                         my ($ip1,$ip2) = bigint2ints($_);
196                         $rgraph->add_edge($ip1,$ip2);
197                 }
198         }
199         else { # extract interfaces and edges from AS graph
200                 # rem: cbgp assigns a unique router of ip address (AS_num*65536) based on an AS
                        topology import
201                 my $undirected = $asgraph->undirected_copy_graph;
202                 foreach ($asgraph->unique_vertices){
203                         $rgraph->add_vertex(($_<<16));
204
205                 }
206                 foreach ($undirected->unique_edges){
207                         $rgraph->add_edge(@{$_}[0]<<16,@{$_}[1]<<16);
208                 }
209         }
210
211         # classify router-level edges (internal or inter-AS)
212         foreach ($rgraph->unique_edges){
213                 if (@{$_}[0]>>16 == @{$_}[1]>>16){ # intra-AS link
214                         $rgraph->set_edge_weight(@{$_}[0],@{$_}[1],RLINK_INTERNAL);
215                 }
216                 else {  # inter-AS link
217                         $rgraph->set_vertex_weight(@{$_}[0],ROUTER_BORDER); # label both routers
                                as border
218                         $rgraph->set_vertex_weight(@{$_}[1],ROUTER_BORDER);
```

```
219                 }
220         }
221         foreach ($rgraph->unique_vertices){
222                 # if no label, classify as internal router
223                 if (not $rgraph->has_vertex_weight($_)){
224                         $rgraph->set_vertex_weight($_,ROUTER_INTERNAL);
225                 }
226         }
227
228         # display information
229         if ($show_info){
230
231                 #-- display per-level distribution of router interfaces
232                 my @interfaces_per_level_counter;
233
234                 # init to zeroes
235                 foreach my $c (0..NLABELS_RNODES-1){
236                         foreach my $l (0..NLEVELS-1){
237                                 $interfaces_per_level_counter[$c][$l]=0;
238                         }
239                 }
240                 # compute values
241                 foreach ($rgraph->unique_vertices){
242                         my $l = $asgraph->get_vertex_weight($_>>16); # get level
243                         my $c = $rgraph->get_vertex_weight($_); # internal or border
244                         $interfaces_per_level_counter[$c][$l]++;
245                 }
246                 # compute per line totals, stored in extra column
247                 foreach my $c (0..NLABELS_RNODES-1){
248                         foreach my $l (0..NLEVELS-1){
249                                 $interfaces_per_level_counter[$c][NLEVELS]+=
                                        $interfaces_per_level_counter[$c][$l];
250                         }
251                 }
252                 # compute per column totals (including line sum), stored in extra line
253                 foreach my $l (0..NLEVELS){
254                         foreach my $c (0..NLABELS_RNODES-1){
255                                 $interfaces_per_level_counter[NLABELS_RNODES][$l]+=
                                        $interfaces_per_level_counter[$c][$l];
256                         }
257                 }
258                 # display
259                 print "- INTERFACE-LEVEL : per-level distribution\n";
260                 foreach my $c (0..NLABELS_RNODES){ # for each router type
261                         print "  ";
262                         foreach my $l (0..NLEVELS){ # for each level
263                                 print $interfaces_per_level_counter[$c][$l]."\t";
264                         }
265                         print "\n";
266                 }
267
268                 #-- compute inter-level connectivity of routers (INTERNAL,PP,PC)
269                 my @inter_level_router_connectivity;
270
271                 # init to all zeroes
272                 foreach my $l1 (0..NLEVELS-1){
273                         foreach my $l2 (0..NLEVELS-1){
274                                 foreach my $p (0..NLABELS_RLINKS-1){
275                                         $inter_level_router_connectivity[$l1][$l2][$p]=0;
276                                 }
277                         }
278                 }
279                 # compute values
280                 foreach ($rgraph->unique_edges){
281                         my $as1 = @{$_}[0]>>16;
282                         my $as2 = @{$_}[1]>>16;
283                         if ($as1==$as2){ # internal link
284                                 my $l = $asgraph->get_vertex_weight($as1);
285                                 $inter_level_router_connectivity[$l][$l][RLINK_INTERNAL]++;
286                         }
287                         else { # inter-as link
```

```
288                                     # get level to which belongs each router
289                                     my $l1 = $asgraph->get_vertex_weight($as1);
290                                     my $l2 = $asgraph->get_vertex_weight($as2);
291                                     # rem : consider individual routers of an AS obeys its policies
292                                     if ($asgraph->has_edge($as1,$as2)){
293                                             if ($asgraph->has_edge($as2,$as1)){ # edge in AS graph is
                                                    bidirectional as1<->as2
294                                                     # add peer link twice 1/2 (once for each router's
                                                            level)
295                                                     $inter_level_router_connectivity[$l1][$l2][
                                                            RLINK_PP]+=0.5;
296                                                     $inter_level_router_connectivity[$l2][$l1][
                                                            RLINK_PP]+=0.5;
297                                             }
298                                             else { # edge direction in AS graph is as1->as2
299                                                     $inter_level_router_connectivity[$l1][$l2][
                                                            RLINK_PC]++;
300                                             }
301                                     }
302                                     elsif ($asgraph->has_edge($as2,$as1)){ # edge direction in AS
                                            graph is as1<-as2
303                                             $inter_level_router_connectivity[$l2][$l1][RLINK_PC]++;
304                                     }
305                             }
306                     }
307                     # compute totals
308                     foreach my $l1 (0..NLEVELS-1){ # per line totals, stored in extra column
309                             foreach my $l2 (0..NLEVELS-1){
310                                     foreach my $p (0..NLABELS_RLINKS-1){
311                                             $inter_level_router_connectivity[$l1][NLEVELS][$p]+=
                                                    $inter_level_router_connectivity[$l1][$l2][$p];
312                                     }
313                             }
314                     }
315                     foreach my $l2 (0..NLEVELS){ # per column totals (including line sum), stored in
                            extra line
316                             foreach my $l1 (0..NLEVELS-1){
317                                     foreach my $p (0..NLABELS_RLINKS-1){
318                                             $inter_level_router_connectivity[NLEVELS][$l2][$p]+=
                                                    $inter_level_router_connectivity[$l1][$l2][$p];
319                                     }
320                             }
321                     }
322                     # display
323                     print "- INTERFACE-LEVEL : inter-level connectivity (internal,pp,pc)\n";
324                     foreach my $l1 (0..$#inter_level_router_connectivity) { # for each level1
325                             print "  ";
326                             foreach my $l2 (0..$#{$inter_level_router_connectivity[$l1]}) { # for each
                                    level2
327                                     foreach my $t (0..$#{$inter_level_router_connectivity[$l1][$l2]})
                                            { # for each link type (internal,PP,PC)
328                                             print $inter_level_router_connectivity[$l1][$l2][$t];
329                                             if ($t!=$#{$inter_level_router_connectivity[$l1][$l2]}){
330                                                     print ",";
331                                             }
332                                     }
333                                     print " \t";
334                             }
335                             print "\n";
336                     }
337             }
338     }
339
340     # -----[ sort_routers_by_level_and_type ]-----------------------------------------
341     # return array containing sorted routers
342     # ------------------------------------------------------------------------------
343     sub sort_routers_by_level_and_type {
344
345             my ($rgraph,$asgraph) = @_;
346
347             my @routers_by_level_and_type;
```

```
348         foreach ($rgraph->unique_vertices){
349                 push (@{$routers_by_level_and_type[$asgraph->get_vertex_weight($_>>16)][$rgraph->
                        get_vertex_weight($_)]},$_);
350         }
351         return \@routers_by_level_and_type;
352 }
353
354 # -----[ sort_ases_by_level ]---------------------------------------------
355 # ------------------------------------------------------------------------
356 sub sort_ases_by_level {
357         my $asgraph = shift;
358         my @aslevels;
359         foreach ($asgraph->unique_vertices){
360                 push (@{$aslevels[$asgraph->get_vertex_weight($_)]},$_);
361         }
362         return \@aslevels;
363 }
364
365 # -----[ get_interfaces ]-------------------------------------------------
366 # ------------------------------------------------------------------------
367 sub get_interfaces
368     {
369         my $topo = shift;
370
371         # hash storing ip addresses
372         my %unique;
373
374         # global match all ip addresses contained in each string
375         foreach (@$topo) {
376                 while (/(\d+)\.(\d+)\.(\d+)\.(\d+)/g){
377                         # remove duplicates by hashing on ip
378                         $unique{address2int("$1.$2.$3.$4")}++;
379                 }
380         }
381
382         # return reference to ip addresses array
383         my @ips = keys(%unique);
384
385         return \@ips;
386 }
387
388 # -----[ get_links ]------------------------------------------------------
389 # ------------------------------------------------------------------------
390 sub get_links
391     {
392         my $topo = shift;
393
394         # hash storing ip addresses
395         my %unique;
396
397         # global match all links contained in each string
398         foreach (@$topo) {
399                 if ($_ =~ m/net add link/){ # TODO optimize
400                         my $ip1;
401                         my $ip2;
402
403                         # match first ip
404                         if (/(\d+)\.(\d+)\.(\d+)\.(\d+)/g){
405                                 $ip1 = "$1.$2.$3.$4";
406                                 # match second ip
407                                 if (/(\d+)\.(\d+)\.(\d+)\.(\d+)/g){
408                                         $ip2 = "$1.$2.$3.$4";
409                                 }
410                         }
411
412                         # remove duplicates by hashing on big int representing symmetrical link
413                         $unique{link2bigint($ip1,$ip2,1)}++;
414                 }
415         }
416
417         # return reference to links array
```

```
418         my @links = keys(%unique);
419
420         return (\@links);
421 }
```

## E.6 Tool module – `tools.pm`

```
1  #!/usr/bin/perl
2  # ==================================================================
3  # Conversion and communication tools' module
4  # @(#)tools.pm
5  # @author Gregory Culpin
6  # @date 08/04/2006
7  # @lastdate 12/05/2006
8  # ==================================================================
9
10 package tools;
11 require Exporter;
12 @ISA= qw(Exporter);
13 @EXPORT= qw(random_n_from_array
14             cbgp_reset
15             cbgp_setup
16             cbgp_check
17             feed
18             link2bigint
19             bigint2link
20             bigint2ints
21             int2address
22             address2int
23             file_to_array
24             array_of_zeros
25             );
26 use constant {
27         # cbgp path
28         CBGP_PATH => 'cbgp',
29 };
30 use strict;
31 use Data::Dumper;
32
33 # ==================================================================
34 # IP link conversion
35 # input : two IP addresses in dotted format, symetrical boolean
36 # output : 64-bit big int representing both adresses
37 # ==================================================================
38 sub link2bigint
39 {
40         my ($IP1,$IP2,$sym) = @_;
41
42         my $ip1 = address2int($IP1); # convert to number format
43         my $ip2 = address2int($IP2);
44
45         my $big; # use big int to generate 64-bit link identifier
46
47         # Link discovery considered symetrical by storing links by "smallest" ip address
48         # No symetry involved in converting a couple of IP addresses for doubletree algorithm
49         if ($ip1<$ip2 or $sym==0){
50                 $big = Math::BigInt->new($ip1);
51                 $big->blsft(32); # high bits are first IP address (shifted 32 bits to the left)
52                 $big->badd($ip2);
53         } else {
54                 $big = Math::BigInt->new($ip2);
55                 $big->blsft(32); # high bits are second IP address (shifted 32 bits to the left)
56                 $big->badd($ip1);
57         }
58
59         return $big;
60 }
```

```
61
62  # ====================================================================
63  # IP link conversion
64  # input : 64-bit big int representing both adresses
65  # output : link of two IP addresses in dotted format
66  # ====================================================================
67  sub bigint2link
68  {
69          my $VALUE= Math::BigInt->new(shift);
70
71          my $IP1 = $VALUE->copy()->brsft(32); # = get 32 high bits, use copy to not modify $VALUE
72          my $IP2 = $VALUE->copy()->band(4294967295); # get 32 low bits, zero out 32 high bits
73
74          return (int2address($IP1),int2address($IP2));
75  }
76
77  # ====================================================================
78  # IP link conversion
79  # input : 64-bit big int representing both adresses
80  # output : two 32-bit ints representing both adresses
81  # ====================================================================
82  sub bigint2ints
83  {
84          my $VALUE= Math::BigInt->new(shift);
85
86          my $IP1 = $VALUE->copy()->brsft(32); # = get 32 high bits, use copy to not modify $VALUE
87          my $IP2 = $VALUE->copy()->band(4294967295); # get 32 low bits, zero out 32 high bits
88
89          return ($IP1,$IP2);
90  }
91
92  # ====================================================================
93  # IP address conversion
94  # input : 32-bit IP address in dotted format
95  # output : 32-bit int representing IP address
96  # ====================================================================
97  sub address2int($)
98  {
99    my $ADDRESS= shift;
100   my @FIELDS= split /\./, $ADDRESS;
101   (scalar(@FIELDS) == 4) or die "Error: incorrect address format \"$ADDRESS\" !";
102
103   return (($FIELDS[0]*256+$FIELDS[1])*256+$FIELDS[2])*256+$FIELDS[3];
104 }
105
106 # ====================================================================
107 # IP address conversion
108 # input : 32-bit int representing IP address (number format)
109 # output : 32-bit IP address in dotted format
110 # ====================================================================
111 sub int2address($)
112 {
113   my $VALUE= shift;
114
115   my $ADDRESS= "".($VALUE % 256);
116   $VALUE= int($VALUE / 256);
117   $ADDRESS= ($VALUE % 256).".$ADDRESS";
118   $VALUE= int($VALUE / 256);
119   $ADDRESS= ($VALUE % 256).".$ADDRESS";
120   $VALUE= int($VALUE / 256);
121
122   return "$VALUE.$ADDRESS";
123 }
124
125 # ====================================================================
126 # Read file
127 # ====================================================================
128 sub file_to_array
129     {
130         my ($filename,$show_info) = @_;
131         my @lines;
```

```
132
133         if ($filename eq '') {
134                 # no file provided
135         }
136         else {
137                 open( FILE, "< $filename" ) or die "Can't open $filename : $!";
138                 while( <FILE> ) {
139                         s/#.*//;            # ignore comments by erasing them
140                         next if /^(\s)*$/;  # skip blank lines
141                         chomp;              # remove trailing newline characters
142                         push @lines, $_;    # push the data line onto the array
143                 }
144                 close FILE;
145                 print "- Imported ".(@lines)." lines from $filename\n" if $show_info;
146         }
147
148         return \@lines;  # array reference
149 }
150
151 # get n random elements from array
152 sub random_n_from_array {
153
154         my ($n,$array) = @_;
155
156         if (scalar @{$array}<=$n){ # send back full array if not enough elements
157                 return @{$array};
158         }
159         else { # add randomly elements to hash until it contains sufficient elements
160                 my %unique;
161                 while (keys(%unique)<$n){
162                         $unique{@{$array}[rand(@{$array})]}++;
163                 }
164                 return (keys(%unique));
165         }
166 }
167
168 sub cbgp_reset
169     {
170         my $cbgp_ref = shift;
171         # reset and finalize current cbgp process
172         $$cbgp_ref->finalize;
173         while (my $res= $$cbgp_ref->expect(0)) {
174                 die "Error: expect \"$res\"\n";
175         }
176         # spawn new process
177         $$cbgp_ref = CBGP->new(CBGP_PATH);
178         $$cbgp_ref->spawn;
179         die if $$cbgp_ref->send("set autoflush on\n");
180         cbgp_check($cbgp_ref);
181 }
182
183 sub cbgp_setup
184     {
185         my ($cbgp_ref,$asfile,$rfile,$rtopo) = @_;
186         if ($rfile ne '') {
187                 $$cbgp_ref->send("include $rfile\n");
188         }
189         else {
190                 $$cbgp_ref->send("bgp topology load $asfile\n");
191                 $$cbgp_ref->send("bgp topology policies\n");
192                 $$cbgp_ref->send("bgp topology run\n");
193         }
194         cbgp_check($cbgp_ref);
195 }
196
197 sub cbgp_check
198     {
199         my $cbgp_ref = shift;
200         $$cbgp_ref->send("print \"CHECK\\n\"\n");
201         $_= $$cbgp_ref->expect(1);
202         chomp;
```

```
203             if ($_ ne "CHECK") {
204                     die "Error : problem with cbgp simulator\n";
205             }
206     }
207
208     sub feed
209         {
210             my ($cbgp,$commands) = @_;
211
212             foreach (@$commands) { # send each command to cbgp
213                     $cbgp->send("$_\n");
214             }
215     }
```