Seventh FRAMEWORK PROGRAMME
FP7-ICT-2007-2 - ICT-2007-1.6
New Paradigms and Experimental Facilities

SPECIFIC TARGETED RESEARCH OR INNOVATION PROJECT

Deliverable D2.3

"Low-level design specification of the machine learning engine"

Project description
Project acronym: **ECODE** Project full title: **Experimental Cognitive Distributed Engine** Grant Agreement no.: **223936**
Document Properties
Number: **TBD** Title: **Low-level design specification of the machine learning engine** Responsible: **TBD** Editor(s): **Damien Saucez** Contributor(s): **Chadi Barakat, Olivier Bonaventure, François Cantin, Pedro Casas Hernandez, Didier Colle, Benoit Donnet, Pierre Geurts, Amir Krifa, Guy Leduc, Pierre Lepropre, Yongjun Liao, Johan Mazel, Philippe Owezarski, Dimitri Papadimitriou, Bart Puype and Damien Saucez** Dissemination level: **Public (PU)** Date of preparation: **20th Sept. 2011** **Version: 1.0**

Deliverable D2.3 - Executive Summary

This deliverable is part of WP2 (Cognitive network & system architecture and design).

The feasibility, benefits and applicability of introducing a cognitive engine in the ECODE architecture are decomposed by using a number of use cases covering different problem areas identified as Internet architectural and design challenges. The Delivrable D2.1 describes the adaptive traffic sampling use case that allow one to efficiently monitor the traffic by adapting the rate at which the traffic is sampled by the monitoring tool. The cooperative intrusion and attack (or anomaly) detection systems forms the second use case. The anomaly detection use case allows the network to detect anomalies which can trigger a reaction from the network to solve or protect against the anomaly. A third use case is for path availability. This use case relies on the so-called IDIPS server. IDIPS ranks Internet paths based on their characteristics, such as delays, or available bandwidth. Finally, a use case that provides efficient network recovery and resiliency is provided in deliverable D2.1. This use case allows the network to effectively recover from an anomaly.

Delivrable D2.1 specifies an architecture that supports the cognitive routing system. Delivrable D2.2 specifies the software architecture of the machine learning engine. This architecture is called the ECODE Unified Architecture (EUA) and is implemented on the XORP routing platform. This delivrable D2.3 consolidates the EUA specification and presents the implementation of the different use cases studied in deliverables D3.3, D3.5, and D3.7. The quality of the implementation of these use cases in the EUA is studied in this deliverable.

Based on the implementation of the different use cases and their evaluation, we can conclude of the success of the ECODE Unified Architecture proposed in deliverable D2.2.

List of Authors

ALB	Dimitri Papadimitriou
IBBT	Didier Colle and Bart Puype
INRIA	Chadi Barakat and Amir Krifa
LAAS	Pedro Casas Hernandez, Johan Mazel and Philippe Owezarski
UCL	Olivier Bonaventure, Benoit Donnet and Damien Saucez
ULg	François Cantin, Pierre Geurts, Guy Leduc, Pierre Lepropre, Yongjun Liao

List of Figures

2.1 Adaptive sampling system design. 8

3.1 High-level description of NEWNADA. Module 1 is responsible for the Multi-Resolution Change-Detection algorithm of NEWNADA. Module 2 performs the Unsupervised Machine-Learning based Analysis of the traffic flows highlighted by Module 1. Finally, Module 3 Characterizes the detected anomalies. 15

3.2 Low-intensity anomalies might be hidden inside highly aggregated traffic, but are visible at finer-grained aggregations. The DDoS attack is evident at the victim's network. 18

3.3 Multi-Resolution Change-Detection (MRCD) module functionalities and interactions. 18

3.4 MRCD traffic capture sub-module API. 19

3.5 MRCD abrupt change detection sub-module API. 19

3.6 MRCD features computation sub-module API. 20

3.7 Sub-Space Clustering: 2-dimensional sub-spaces $\mathbf{X}_1$, $\mathbf{X}_2$, and $\mathbf{X}_3$ are obtain from a 3-dimensional feature space $\mathbf{X}$ by simple projection. Units in the graph are irrelevant. 21

3.8 Unsupervised Analysis (UA) module functionalities and interactions. . . 23

3.9 UA sub-spaces computation sub-module API. 24

3.10 UA DBSCAN clustering sub-module API. 24

3.11 UA Evidence Accumulation EA4C and EA4O sub-modules API. . . . 25

3.12 UA Anomaly Characterization sub-module API. 25

3.13 NEWNADA XORP processes within the EUA. 26

3.14 MRCD Monitoring Point XORP process. 27

3.15 MRCD Monitoring Point process XRL interface. 28

3.16 UAD Machine Learning XORP process. 28

3.17 UAD Machine Learning Process XRL interface. 29

4.1 IDIPS within the EUA 32

4.2 IDIPS server API for synchronous mode clients 34

4.3 IDIPS server API for asynchronous mode clients 35
4.4 One-by-one path ranking retrieval algorithm 37
4.5 Measurement module API 37
4.6 Prediction module API 38
4.7 Example of modules interactions in IDIPS 40
4.8 Querying module XRL interface 43
4.9 Measurement module XRL interface 44
4.10 Prediction module XRL interface 45
4.11 Measurement module `loop` method pseudo-code 50
4.12 UDP ICMP port unreachable management 50

5.1 High-level flowchart for normal OSPF LSA processing 54
5.2 High-level flowchart for SRG inference 55
5.3 Information model 56
5.4 Outline of SRG table 57
5.5 OSPF process flow 58
5.6 Correlating incoming LSAs with old link-state database to find failing links 59
5.7 Link failure detection and failing links update 59
5.8 Correlating update link-state database and set of SRG links to find list of failing links of interest 60
5.9 Use of set of failing links in pruning the shortest-path table 60
5.10 Use of set of failing links in pruning the shortest-path table 61
5.11 Implementation of state in OSPF module 62
5.12 Structure of router LSA and contained router links 63
5.13 Example of OSPF (left) and MLP (right) interaction in time 64
5.14 XRL interface for receiving link failure reports 64
5.15 XRL interface for the SRG monitoring point 65
5.16 Changes to the OSPF interface (partial) 66
5.17 High-level comparison of Xorp 1.6 (left) and EUA/Xorp 1.8 (right) implementation 66
5.18 Distributed/centralized SRG inference scenarios 67

Table of contents

1 Introduction **1**

1.1 Scope of Deliverable 1

1.1.1 Use cases 3

1.1.1.1 a1) Adaptive traffic sampling 3

1.1.1.2 a3) Cooperative intrusion and attack / anomaly detection 3

1.1.1.3 b1) Path availability and IDIPS 4

1.1.1.4 b2) Network recovery & resiliency / OSPF SRG inference 5

1.2 Structure of Document 5

2 Adaptive traffic sampling **7**

2.1 Introduction 7

2.2 System design 8

2.3 Implementation 9

2.4 Conclusion 11

3 Cooperative intrusion and attack / anomaly detection **13**

3.1 Introduction 13

3.2 System design 14

3.2.1 Multi-Resolution Change-Detection (MRCD) Module 16

3.2.2 Unsupervised Analysis Module 20

3.2.3 Characterization Module 24

3.3 Implementation 27

3.4 Conclusion 29

4 Path availability and IDIPS **31**

4.1 Introduction 31

4.2 System design 31

4.2.1 Querying module 33

4.2.2 Measurement Module 37
4.2.3 Prediction Module 38
4.3 Implementation 39
4.3.1 High Level Cost Functions Implementation 45
4.3.2 Examples of IDIPS module implementation 48
4.4 Conclusion 51

5 Network recovery & resiliency / OSPF SRG inference 53
5.1 Introduction 53
5.2 System design 54
5.3 Implementation 57
5.4 Conclusion 67

6 Conclusion 69

References 69

Chapter 1

Introduction

1.1 Scope of Deliverable

This deliverable is part of WP2 (Cognitive network & system architecture and design), which is a research and technological development activity. The overall objective of this work package is to design a cognitive routing system and engine by combining machine learning and networking techniques in order to efficiently address the future Internet challenges. This cognitive routing engine will enhance the existing routing system by combining machine learning methods that allow it to derive a number of observations from the data collected by its routing and forwarding engines and its interactions with other cognitive routing engines.

The overall objective of this work package is drawn (i) from a set of networking use cases representative of the Internet challenges, and referred to as technical objectives, and (ii) from applying novel machine-learning mechanisms (by designing a cognitive engine) to these use cases so as to address these challenges.

The first technical objective requires to develop adaptive methods for traffic sampling in core networks in order to take the appropriate actions either at the router level or the network-wide level (based on the sampling and processing results). It is also necessary to monitor the path performances (e.g., delay, loss rate, etc.) Monitoring is performed by means of a collaborative measurement tool that can be used to determine the best suited routing strategies. Finally, a cooperative intrusion and attack / anomaly detection system is required. It consists on a distributed tool enabling anomaly, attack, and intrusion detection by monitoring traffic and detecting changes in the measurements. The second technical objective aims at determining the availability of the paths and their qualities. If a path is not reachable anymore, a resiliency mechanism ensures that the network recovers from the failure. Finally, the third technical objective aims at enforcing the quality requirements in a scalable way. To achieve these objectives, the architecture is composed of four parts:

Data collection provides the appropriate interfaces for packet capture and conversion, for fast reaction by means of on-line processing (e.g., through adaptive packet sampling) as well as validation of the decisions by means of off-line processing. It also determines the relevant alternate source of information (e.g., routing table entries, routing information updates, daemon logs, active measurement tools) and designs interface to export meaningful events for further processing. Finally, it proposes flexible mechanisms to extract relevant information from captured packet or events and builds corresponding information tuples that will serve as input to the machine learning algorithms (processing function).

Interpretation provides mechanisms for online verification and notification of machine learning output accuracy and determines whether prior knowledge can be used or if the knowledge must be updated with new measurements.

Control determines the suitable hook points in the routing engine/forwarding engine or interfaces for passing decision(s) from the machine learning algorithms. It also determines the set of actions to take to meet the expected behavior.

Cooperation and Distribution determines the techniques cooperation between the different engines and the distribution of the processing between the "peers" and determines how to exchange the "knowledge", e.g., learned rules, between peers.

In a first phase (Task_2.1), WP2 has provided the network and system architecture framework that realizes the networking technical objectives listed here above by means of novel machine learning mechanisms techniques. This architecture defined the interaction between the routing platform components and the machine learning (cognitive) components. This architecture has been incrementally reviewed as results from experimentation phase 1 (WP3) were obtained. In a second phase (Task_2.2), at the middle of the project timeline, a low-level cognitive engine system has been designed. This cognitive engine has been experimented during the experimentation phase 2 of the project as part of WP4. In a third phase (Task_2.3), at the end of the project, a consolidated cognitive engine design has been proposed (resulting from experimentation) that enables knowledge exchange and synchronization with other cognitive engines. The architecture supporting this cognitive routing system is specified in deliverable D2.1. An experimental prototype is implemented on the XORP routing platform as detailed deliverable D2.2 and consolidated by the present document. XORP is an open source routing platform. XORP provides a fully featured control-plane platform that implements routing protocols and a unified platform to configure them. XORP's modular architecture allows rapid introduction of new protocols, features, and functionalities. Our prototypes are implemented on top of the ECODE Unified Architecture (EUA) defined in deliverable D2.2. The EUA is a distributed-capable XORP extension aiming at realizing technical experiments.

This deliverable D2.3 documents the experimentally validated software architecture and its companion toolbox library of methods and components as well as a clear description of the interfaces and components that would allow implementation of interoperable parts by third-party developers. This deliverable describes the design of the learning

modules (together with their companion toolbox library of learning methods and components) implemented to run as part of ECODE XORP-based platform documented in deliverable D2.2. The present document provides a detailed description of the interfaces and components that would allow implementation of interoperable parts by third-party developers. The experimental validation of the software architecture presented in deliverable D2.2 together with the experimental results that can be obtained by means of the modules, interfaces and components presented in this deliverable are included as part of deliverable D4.3.

1.1.1 Use cases

Delivrable D2.1 describes the following use cases necessary to meet the technical objectives:

1.1.1.1 a1) Adaptive traffic sampling

Traffic measurement and analysis are crucial management activities for network operators. With the increase in traffic volume, operators resort to sampling primitives to reduce the measurement load. Unfortunately, existing systems use sampling primitives separately and configure them statically to achieve some performance objective. It becomes then important to design a new system that combines different existing sampling primitives together to support a large spectrum of monitoring tasks while providing the best possible accuracy by spatially correlating measurements and adapting the configuration to traffic variability. In this use-case, we introduce a new adaptive system that combines two sampling primitives, packet sampling and flow sampling, and that is able to satisfy multiple monitoring tasks. Our system is general enough to account for other sampling primitives and for a diversity of monitoring tasks, either separately or jointly (accounting, large flow detection, flow counting, etc). It consists of two main functions: (i) a global estimator that investigates measurements done by the different sampling primitives inside routers in order to deal with multiple monitoring tasks and to construct a more reliable global estimator while providing visibility over the entire network; (ii) an optimization method based on overhead prediction that allows to reconfigure monitors according to accuracy requirements and monitoring constraints.

1.1.1.2 a3) Cooperative intrusion and attack / anomaly detection

The *Unsupervised Network Anomaly Detection Algorithm* (NEWNADA) is proposed to meet the objective of automatic detection and characterization of intrusions and attacks/anomalies. NEWNADA [MCO11] is a completely unsupervised approach to detect and characterize network attacks, intrusions, and anomalies, without relying on signatures or labeled traffic of any kind. The proposed approach permits to detect both well-known as well as completely unknown attacks, and to automatically produce easy-to-interpret signatures that characterize them. Unsupervised detection is accom-

plished by means of robust data-clustering techniques, combining Sub-Space Clustering [PHL04], Density-based Clustering [EKSX96], and multiple Evidence Accumulation [FJ05] algorithms to blindly identify anomalous traffic flows. Based on the observation that network attacks, and particularly the most difficult ones to detect, are contained in a small fraction of **aggregated flows** with respect to normal-operation traffic [ACP09], their unsupervised detection consists in the identification of outlying traffic flows, i.e. flows that are remarkably different from the majority. Unsupervised characterization is achieved by exploring inter-flows structure from multiple outlooks, building filtering rules to describe a detected anomaly.

NEWNADA works in a completely unsupervised fashion, which means that it can be directly plugged-in to any monitoring system and start to detect anomalies from scratch, without any kind of calibration. The algorithm analyzes traffic captured at a single-link, producing easy-to-interpret signatures that characterize a detected anomalous traffic event. This permits to reduce the time spent by the network operator to understand the nature of a detected anomaly. In addition, the automatically produced signatures can be directly exported towards standard signature-based security devices like IDSs, IPSs, and/or Firewalls to rapidly detect the same anomaly in the future. NEWNADA is designed to work in an on-line basis, analyzing traffic captured at consecutive time slots of fixed duration.

1.1.1.3 b1) Path availability and IDIPS

ISP-Driven Informed Path Selection (IDIPS) is proposed to meet the path availability and performance objectives. IDIPS is generic as it can be used in many networking contexts without changing anything to its behavior. IDIPS is scalable, lightweight, and designed to be easily deployed.

IDIPS is designed as a request/response service. The network operators deploy servers that are configured with policies and that collect routing information (e.g., OSPF, BGP) and measurements towards popular destinations. The clients that need to select a path send requests to an IDIPS server. A request contains a list of sources, a list of destinations, and a traffic qualification that determines the rule for ranking the paths to use. The client already knows the different paths it needs to rank. The server replies with an ordered list of $< source, destination, rank >$ tuples to the client. The reply gives an indication of the ranking lifetime. This ranking is based on the current network state and policies. The client will then use the first pairs of the list and potentially switch to the next one(s) in case of problems or if it wants to use several paths in parallel.

1.1.1.4 b2) Network recovery & resiliency / OSPF SRG inference

SRG inference is used to identify shared risk groups from network element failure history. Through clustering and data-mining of failure occurences, a predictive model is built which allows inferring the failure of an SRG upon the detection of a first (second ...) network element failure. Since failure detection in routing protocols such as OSPF requires time in the order of seconds and recover network element one-by-one, SRG inference allows for faster recovery by rerouting around the inferred failing elements pre-emptively.

The OSPF SRG inference system works by setting up a two-directional communications path between the OSPF module and the machine learning inference module. This interaction requires two changes to the standard OSPF module. The link-state advertisement algorithm is changed such that link failures are detected and reported to the inference module. Also, the OSFP module interface is changed such that it can accept inference information; the rerouting process is adapted to take this information into account, routing around links when they are part of an inferred SRG during initial detection of failure(s).

1.2 Structure of Document

Chapter 2 addresses the problem of monitoring traffic at high rate and for different sampling rates. To do so, an adaptive traffic sampling is applied. The sampling rate is adapted dynamically to optimally monitor the flows. Monitoring at optimal sampling rate ensure a minimum usage of resources even when several independent monitoring tasks are run in parallel.

Chapter 3 addresses the problem of automatically detecting and characterizing intrusions and attacks. The strength of the proposed solution is that the detection and characterization is performed without prior knowledge and does not require any supervision. To to do so, the architecture uses data clustering. The characterization can be use to automatically generate signatures that are useful for intrusion detection systems.

Chapter 4 presents the architecture of the ISP-Driven Informed Path Selection (IDIPS). IDIPS is a service that aim at ranking paths based on their performances. IDIPS monitors the networks to predict the future path behavior. The prediction can be used to determine the quality of the paths and thus rank them for the clients to select the paths that will provide the best performance.

Chapter 5 presents novel technique for fast network failure recovery and improvement of routing path resiliency. The proposed system uses advanced machine learning techniques to infer the shared risk groups (SRG) inside a network. The implemented system improves re-routing time in case of failure and thus network resiliency. Indeed, with current link state protocols, simultaneous link failures resulting from an SRG failure can trigger multiple successive routing table entries re-computation, one to address each of the link failure. Failing to account SRG failures during routing table entries re-

computation leads to longer recovery time and thus, higher magnitude of packet losses compared to the situation where the set of links (associated to the SRG failure) results in a single re-computation of all routing tables entries affected by the failure. Instead, if the router learns about the existence of SRGs from the arriving link state updates, then decisions regarding SRG failure can be taken promptly to avoid successive re-computations of alternate shortest paths across the updated topology.

Finally, Chapter 6 concludes this deliverable. It summarizes the main contributions and describes some additional work that can be realized thanks to the achievements of the ECODE project.

Chapter 2

Adaptive traffic sampling

2.1 Introduction

The importance of passive traffic measurements for the understanding and diagnosis of core IP networks has led to a considerable evolution in the number and quality of monitoring tools and techniques. Recently, numerous monitoring primitives have been proposed in order to achieve a large number of network management tasks. The spectrum is broad covering among others flow sampling [HV03], sample and hold [EV02] and packet sampling [CIB+06]. Currently, NetFlow [Cis00] is the most widely deployed measurement solution by ISPs. However, this solution still presents some shortcomings, namely the problem of accurately configuring sampling rates according to network conditions (and in particular, in the increasing trend in line speed) and the requirements of monitoring applications.

Numerous solutions exists that provide a balance between scalability (respecting the resource consumption constraints) and accuracy, many works have investigated the existing sampling primitives and have used them to build network-wide monitoring systems that coordinate responsibilities between the different monitors. These solutions rely on systems that use single sampling primitives to achieve specific management applications. However, none of these systems is optimized to achieve a general class of monitoring tasks and to combine different sampling primitives. In order to solve these limitations, some proposals have presented simple combination of existing sampling primitives in order to achieve a larger class of tasks. For instance, the authors in [VS10] combine a small number of simple and generic router primitives that collect flow-level data to estimate traffic metrics, while the authors in [KME05] use a combination of flow sampling and sample-and-hold to provide traffic summaries and detect resource hogs. The novel monitoring system we propose in ECODE is able to integrate various existing monitoring primitives (namely, packet sampling and flow sampling) in order to support multiple monitoring tasks, namely flow counting, flow size estimation and heavy-hitter detection. This system is not only able to combine different sampling primitives, but more importantly can adapt their contribution in a way to maximize the global measurement accuracy at limited overhead. Different monitoring applications will automatically

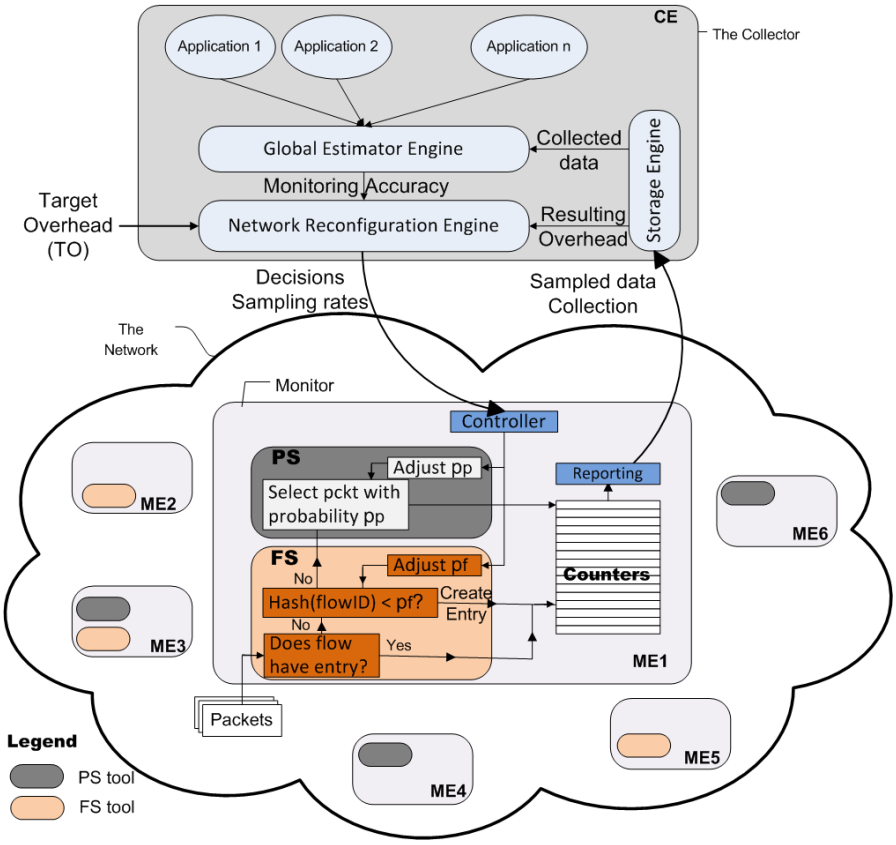

Figure 2.1: Adaptive sampling system design.

lead to different tuning of the sampling primitives.

2.2 System design

Figure 2.1 depicts the basic functional components of the proposed monitoring system together with the interactions among them. The system relies on existing NetFlow-like local measurement tools (Monitoring Engine (ME)) deployed in network routers. We chose to use two complementary sampling primitives: *(i)* Flow Sampling (FS) which is well suited for security and anomaly detection applications that require analyzing the flow communication structure, and *(ii)* Packet Sampling (PS) which is well suited for traffic engineering and accounting applications based on the traffic volume structure, e.g., heavy-hitter detection and traffic engineering that require an understanding of the number of packets/bytes per-port or per-prefix [VS10].[1]

Our system extends these local existing monitoring tools (MEs) with a centralized network-wide cognitive engine (CE) that drives its own deployment by automatically

[1]While packet sampling consists in capturing a subset of packets independently of each other, flow sampling consists in capturing flows independently of each other. Once a flow is captured by flow sampling, all its packets are captured and analyzed. The decision to capture a flow or not is done at the beginning of the flow.

and periodically reconfiguring the different monitors in a way to improve the overall accuracy (according to monitoring application requirements) and reduces the resulting overhead (respecting some resource consumption constraints, typically the volume of measurements).

The cognitive engine of our network-wide system comprises two main modules (i) the global estimator (GE) engine that combines measurements and estimates network traffic to provide a global more accurate estimation, and (ii) the reconfiguration engine (RE) that dynamically adjusts the sampling rates in routers. The GE engine extends local existing monitoring tools (MEs) with a network-wide inference engine that combines their measurements to support a large spectrum of applications and provide more accurate results. Given a set of measurement tasks T to realize, this inference engine investigates the local measurements made by the different routers to obtain a global and more reliable view. The RE, given a list of measurement tasks T and an overhead constraint measured in terms of reported NetFlow records (Target Overhead TO), adaptively adjusts its configuration of the sampling primitives according to the requirements of the multiple tasks while tracking short-term and long-term variations in the traffic. A configuration is a selection of sampling rates of the different primitives on the different interfaces of network routers (or monitors). This configuration is periodically updated as a function of the overhead and in a way to optimize the accuracy of the considered measurement tasks. Further details concerning these computational procedures executed by these modules can be found in Deliverable D3.3.

2.3 Implementation

For efficiency and compatibility reasons (e.g., NetFlow can potently run on switches or dedicated devices), the adaptive traffic sampling is not implemented directly in the EUA. However, the different cognitive elements that one could implement in the EUA can interact with our adaptive traffic sampling module natively within the EUA. To do so, we have implemented a wrapper that makes the link between the EUA and the adaptive traffic sampling module. The wrapper is implemented directly in the EUA and communicates with the adaptive module with over UDP. The role of the wrapper is to translate the XRL requests received in the EUA into primitives understood by the adaptive traffic sampling module. The rest of this section presents the XRLs that can be used by an EUA element to interact with the adaptive traffic sampling. The XRL directly correspond to the primitives of our sampling module.

The following two functions can be used to retrieve the optimally sample NetFlow reports. The returned NetFlow reports are filtered according to the parameters provided at the function call.

```
get_5tuple_flows_report
         ? sourceip & destinationip & sourceport
         & destinationport & protocol
         -> report

get_ipsource_ipdest_flows_report
        ? source & destination
        -> report
```

`get_5tuple_flows_report` returns reports filtered to only provide information about the flows that match the 5 tuples (i.e., source IP, destination IP, source port, destination port and protocol).

`get_ipsource_ipdest_flows_report` filters the reported flow on their <source IP, destination IP> address pair.

The following two functions allow one to interact with the sampling rate.

```
get_current_sampling_rate
        ? interface
        -> rate

change_current_sampling_rate
        ? interface & sa & sb
        -> ack
```

`get_current_sampling_rate` returns the current sampling rate used at a particular interface (softflowd should be running on this interface).

`change_current_sampling_rate` changes the sampling rate used at an interface. The new sampling rate is set to $\frac{sa}{sb}$.

The following two functions aggregate information based on NetFlow reports.

```
get_aggregated_sent_bytes_for_ipdest
         ? destination
         -> sentbytes

get_ipsource_ipdest_flows_report_filter
         ? source & destination & srcfilter & dstfilter
         -> report
```

`get_aggregated_sent_bytes_for_ipdest` provides the total number of bytes monitored for a given destination.

`get_ipsource_ipdest_flows_report_filter` aggregates into one single NetFlow record the information that correspond to all the flows matching the source and destination addresses. This function has the particularity to accept source and destination filters. These filters are applied on the source and destination address. The filters are used to implement prefix exact matching instead of IP address matching. The report is formatted as follow: "srcAdr dstAdr totalNbrPackets totalNbrBytes minStartTime maxStartTime".

2.4 Conclusion

We have presented an adaptive system that combines different existing sampling primitives in order to support a large spectrum of monitoring tasks while providing the best possible accuracy. Our system coordinates responsibilities between the different monitors and shares resources between the different sampling primitives. Our system is practical and provides a flexible optimization method based on overhead prediction that reconfigures monitors according to monitoring applications requirements and network conditions.

Chapter 3

Cooperative intrusion and attack / anomaly detection

3.1 Introduction

The *Unsupervised Network Anomaly Detection Algorithm* (NEWNADA) is an unsupervised machine-learning based system conceived to meet the objective of automatic detection and characterization of intrusions and attacks/anomalies within ECODE. NEWNADA is composed of three different modules. First, a monitoring Multi-Resolution Change-Detection module captures traffic in real-time and looks for anomalous changes in basic traffic descriptors. Second, an Unsupervised Machine-Learning based Analysis module uses a robust multi-clustering algorithm to identify the set of responsible traffic flows without relying on signatures or calibration. Finally, a Characterization module automatically produces a set of filtering rules to correctly isolate and characterize the identified anomalous flows.

NEWNADA is a traffic analysis system that permits to identify previously unknown anomalous traffic behaviors without relying on signatures or calibration. The system permits to rank the degree of abnormality of a set of traffic flows going through a monitored network link. In addition, NEWNADA provides a summary of the most relevant traffic descriptors that characterize the top-ranked flows in the form of anomalous traffic signatures. Such signatures permit to automatically separate the interesting traffic events from the normal-operation traffic, dramatically simplifying network monitoring tasks. The information provided by NEWNADA permits not only to pinpoint anomalous traffic flows, but also to rapidly understand the nature of the anomaly and therefore, to rapidly apply accurate and adapted countermeasures.

This chapter is decomposed in two parts. On the one hand, Sec. 3.2 describes the modules, their design and interactions. On the other hand, the modules implementation within the EUA is described in Sec. 3.3.

An evaluation of NEWNADA with real traffic containing different types of network attacks, including DDoS, worms, and buffer-overflow attacks can be found in D4.3.

3.2 System design

In this section we describe the design of our Unsupervised Network Anomaly Detection system within the EUA. NEWNADA runs in three consecutive steps, analyzing packets captured in a single-link at consecutive time slots of fixed duration. Fig. 3.1 depicts a modular, high-level description of NEWNADA's design. Each of the three modules is responsible for one of the three consecutive monitoring and analysis tasks. The first step is accomplished by the Multi-Resolution Change-Detection module, and consists in detecting an anomalous time slot in which the unsupervised machine-learning based analysis will be performed. For doing so, packets captured at each time slot are aggregated in standard *5-tuples* IP flows. IP flows are additionally aggregated at different *resolution* levels in what we shall refer to as *macro-flows*, using network prefix and IP address (either IPsrc or IPdst). A macro-flow represents all the IP flows coming from or directed towards the same sub-network or network host.

Different time series are then constructed for consecutive time slots, using simple traffic metrics such as number of bytes, number of packets, number of macro-flows, and number of SYN packets per time slot. A basic change-detection algorithm based on absolute deltoids [CM05] is finally used to detect an anomalous behavior in these multiple time-series. Tracking anomalous behaviors from multiple metrics and at multiple resolutions (i.e. $/8$, $/16$, $/24$, $/32$ network mask) provides additional reliability to the change-detection algorithm, and permits to detect both single source-destination and distributed anomalies of very different characteristics. Sec. 3.2.1 presents additional details on this module.

The second step takes as input **ALL** the n macro-flows contained in the time slot flagged as anomalous (i.e., no filtering or flow-removing process is performed by the first module). Each of these macro-flows is described by a set of m traffic attributes or *traffic features*, like number of source hosts, number of destination ports, or packet rate. Let $\mathbf{X} \in \mathbb{R}^{n \times m}$ be a matrix of traffic features, describing the n different macro-flows. The Unsupervised Machine Learning based Analysis module detects *outlying* macro flows in $\mathbf{X}$ (i.e., macro-flows which are remarkably different from the rest) using a robust multi-clustering algorithm, based on a combination of Sub-Space Clustering (SSC) [PHL04], Density-based Clustering [EKSX96], and Evidence Accumulation Clustering (EAC) [FJ05] techniques. NEWNADA's clustering algorithm is capable of identifying anomalous traffic structures and to rank their degree of rareness within the m-dimensional features' space generated by the set of m traffic features.

The selection of the m features used in $\mathbf{X}$ to describe the macro flows is a key issue to any anomaly detection algorithm, but it becomes critical and challenging in the case of unsupervised detection, because there is no additional information to select the most relevant set. In general terms, using different traffic features permits to detect different types of anomalies. In current version of NEWNADA we shall limit our study to detect well-known attacks (DDoS, worms, buffer-overflow attacks, etc.), using a set of standard traffic features widely used in the literature. However, NEWNADA can be easily extended to detect other types of anomalies, considering different sets of traffic features. In fact, more features can be added to any standard list to improve detection

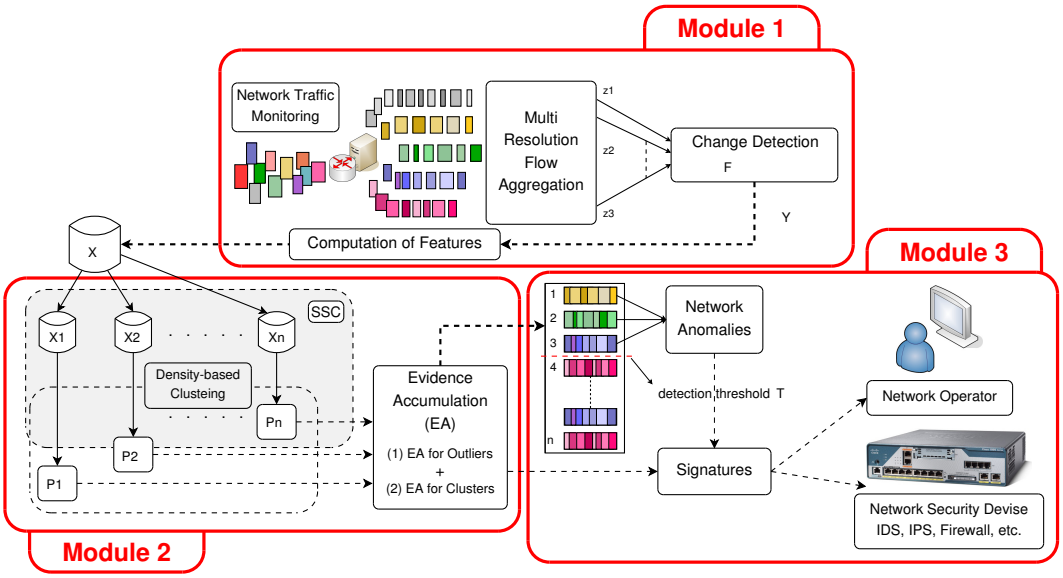

Figure 3.1: High-level description of NEWNADA. Module 1 is responsible for the Multi-Resolution Change-Detection algorithm of NEWNADA. Module 2 performs the Unsupervised Machine-Learning based Analysis of the traffic flows highlighted by Module 1. Finally, Module 3 Characterizes the detected anomalies.

results. In fact, more features can be added to any standard list to improve detection results. For example, we could use the set of traffic features generally used in the traffic classification domain [WZA06] for our problem of anomaly detection, as this set is generally broader; if these features are good enough to classify different traffic applications, they should be useful to perform anomaly detection. The main advantage of the Unsupervised Machine Learning based Analysis module of NEWNADA is that we have devised an algorithm to highlight outliers respect to any set of features, and this is why the algorithm is highly applicable.

For example, according to previous work on signature-based anomaly characterization [FO09], simple traffic features such as number of source/destination IP addresses and ports (nSrcs, nDsts, nSrcPorts, nDstPorts), ratio of number of sources to number of destinations, packet rate (nPkts/sec), average packet size (avgPktsSize), and fraction of ICMP and SYN packets (nICMP/nPkts, nSYN/nPkts) permit to describe standard network attacks such as DoS, DDoS, scans, and spreading worms/virus.

Table 3.1 describes the impacts of different attacks on the aforementioned traffic features. All the thresholds are introduced to better explain the evidence of an attack in some of these features. DoS/DDoS attacks are characterized by many small packets sent from one or more source IPs towards a single destination IP. These attacks generally use particular packets such as TCP SYN or ICMP echo-reply, echo-request, or host-unreachable packets. Port and network scans involve small packets from one source IP to several ports in one or more destination IPs, and are usually performed with SYN packets. Spreading worms differ from network scans in that they are directed towards a small specific group of ports for which there is a known vulnerability to exploit (e.g. Blaster on TCP port 135, Slammer on UDP port 1434, Sasser on TCP port 455), and

Type of Attack	Class	Agg-Key	Impact on Traffic Features
DoS (ICMP/SYN)	1-to-1	IPdst	$\text{nSrcs} = \text{nDsts} = 1, \text{nPkts/sec} > \lambda_1, \text{avgPktsSize} < \lambda_2, \text{nICMP/nPkts} > \lambda_3, \text{nSYN/nPkts} > \lambda_4$.
DDoS (ICMP/SYN)	N-to-1	IPdst	$\text{nDsts} = 1, \text{nSrcs} > \alpha_1, \text{nPkts/sec} > \alpha_2, \text{avgPktsSize} < \alpha_3, \text{nICMP/nPkts} > \alpha_4, \text{nSYN/nPkts} > \alpha_5$.
Port scan	1-to-1	IPsrc	$\text{nSrcs} = \text{nDsts} = 1, \text{nDstPorts} > \beta_1, \text{avgPktsSize} < \beta_2, \text{nSYN/nPkts} > \beta_3$.
Network scan	1-to-N	IPsrc	$\text{nSrcs} = 1, \text{nDsts} > \delta_1, \text{nDstPorts} > \delta_2, \text{avgPktsSize} < \delta_3, \text{nSYN/nPkts} > \delta_4$.
Spreading worms	1-to-N	IPsrc	$\text{nSrcs} = 1, \text{nDsts} > \eta_1, \text{nDstPorts} < \eta_2, \text{avgPktsSize} < \eta_3, \text{nSYN/nPkts} > \eta_4$.

Table 3.1: Features used by NEWNADA in the detection of DoS, DDoS, network/port scans, and spreading worms. For each type of attack, we describe its impact on the selected traffic features.

they generally use slightly bigger packets. Some of these attacks can use other types of traffic, such as FIN, PUSH, URG TCP packets or small UDP datagrams.

In the third and final step, the top-ranked outlying macro flows are flagged as anomalies, using a simple thresholding approach. The automatic anomaly Characterization module additionally uses the evidence of traffic structure provided by the Clustering module to produce filtering rules that characterize the detected anomalies, which are ultimately combined into a new anomaly signature. This signature provides a simple and easy-to-interpret description of the problem, easing network operator tasks. Sec. 3.2.3 presents additional details on this module.

3.2.1 Multi-Resolution Change-Detection (MRCD) Module

NEWNADA performs abrupt-change detection on standard IP flows, aggregated at 9 different macro-flow resolutions l_i. These include, from coarser to finer-grained resolution: *traffic per Time Slot* (l_1:tpTS), *source Network Prefixes* ($l_{2,3,4}$: $\text{IPsrc}/8$, $\text{IPsrc}/16$, $\text{IPsrc}/24$), *destination Network Prefixes* ($l_{5,6,7}$: $\text{IPdst}/8$, $\text{IPdst}/16$, $\text{IPdst}/24$), *source IPs* (l_8: IPsrc), and *destination IPs* (l_9: IPdst). The 7 coarsest-grained resolutions are used for change-detection, while the remaining 2 are additionally used by the second module in the clustering step.

To detect an anomalous time slot, time-series $Z_t^{l_i}$ are constructed for 4 simple traffic metrics that include number of bytes, number of packets, number of macro-flows, and number of SYN packets per time slot, using resolutions $i = 1, \ldots, 7$. Any generic change-detection algorithm $\mathcal{F}(.)$ based on time-series analysis is then used on $Z_t^{l_i}$. In particular, we have decided to use a simple yet efficient change-detection algorithm based on absolute deltoids [CM05].

This algorithm works as follows: every ΔT seconds, the aforementioned traffic metrics $Z_t = \{z_t(1), z_t(2), z_t(3), z_t(4)\} = \{\#\text{bytes}_t, \#\text{pkts}_t, \#\text{flows}_t, \#\text{SYN}_t\}$ are computed. Using Z_t, the absolute deltoids $D_t = \{d_t(1), d_t(2), d_t(3), d_t(4)\} = Z_t - Z_{t-1}$ are computed for current time slot t. The change-detection algorithm $\mathcal{F}(D_{t_0}^{l_i})$ flags an anomalous traffic behavior at time slot t_0 if any of the deltoids $d_{t_0}(j)$ exceeds a detection threshold $\lambda(j)$, $j = 1, \ldots, 4$ in any of the 7 aggregation levels (the analysis is done from

coarser to finer resolution, i.e., from l_1 to l_7 resolution). Each detection threshold $\lambda(j)$ is computed from the standard deviation of the corresponding deltoid $d_t(j)$, obtained from a set of M past measurements:

$$\lambda(j) \;=\; \rho \left[\frac{1}{M-1} \sum_{i=1}^{M} \big(d_i(j) - \bar{d}(j)\big)^2 \right]^{\frac{1}{2}} = \; \rho\, \sigma_d(j) \tag{3.1}$$

where ρ is a scaling factor that permits to adjust the sensitivity of detection. In order to cope with normal traffic variations, each detection threshold $\lambda(j)$ is periodically updated: if no anomalous behavior was flagged during the past M temporal slots, the variance of each deltoid is recomputed from the last M deltoids.

The choice of these 4 simple traffic metrics for change-detection is based on [LCD04], but the algorithm can be used with any other traffic metric sensitive to anomalies. Tracking anomalies at multiple aggregation levels provides additional reliability to the change-detection algorithm, and permits to detect both single source-destination and distributed anomalies of very different intensities. Fig. 3.2 shows how a low intensity DDoS attack might be dwarfed by highly-aggregated traffic flows. The time-series associated with the number of packets, namely $Z_t = \#\mathrm{pkts}_t$, does not present a perceptible deltoid D_t at tpTS aggregation (left); however, the attack can be easily detected using a finer-grained resolution, e.g., at the victim's network ($\mathrm{IPdst}/24$ aggregation, on the right).

The final step performed by the MRCD module consists in computing the $n \times m$ matrix $\mathbf{X}$, which describes the set of n macro-flows present in the flagged anomalous time slot using the m predefined traffic features.

The functionalities of the MRCD module are accomplished by three different sub-modules, depicted in Fig. 3.3. The Network Traffic Capture sub-module uses the libpcap [Lib] library to capture raw traffic at the network interface of analysis in time slots of $\Delta\mathtt{T}$ seconds. In addition, the sub-module computes the different macro-flow aggregations l_1 to l_7 for the 4 different metrics $\{\#\mathrm{bytes}, \#\mathrm{pkts}, \#\mathrm{flows}, \#\mathrm{SYN}\}$.

Fig. 3.4 depicts the API that implements these functionalities. At each time slot, function `<capture_raw_traffic>` builds a traffic structure P_t that contains all the packets on the corresponding slot of duration $\Delta\mathtt{T}$. This structure is used by the `<aggregate_traffic>` function to compute time-series sample Z_t, which will be then used by the Abrupt Change Detection sub-module. Sample Z_t is additionally stored in a buffer of $M+1$ samples that holds the last $M+1$ anomaly-free samples $Z_{t-1}, Z_{t-2}, \ldots, Z_{t-M}$; these are used by the change-detection algorithm to compute the last M anomaly-free absolute deltoids $D_t, D_{t-2}, \ldots, D_{t-M+1}$ to update its detection thresholds.

The Abrupt Change Detection sub-module's API (depicted in Fig. 3.5) permits to flag an anomalous time slot through the function `<change_detection>`, updating the result of the change-detection analysis in the boolean variable $flag$ every $\Delta\mathtt{T}$ seconds. The function `<update_detection_thresholds>` permits to update the

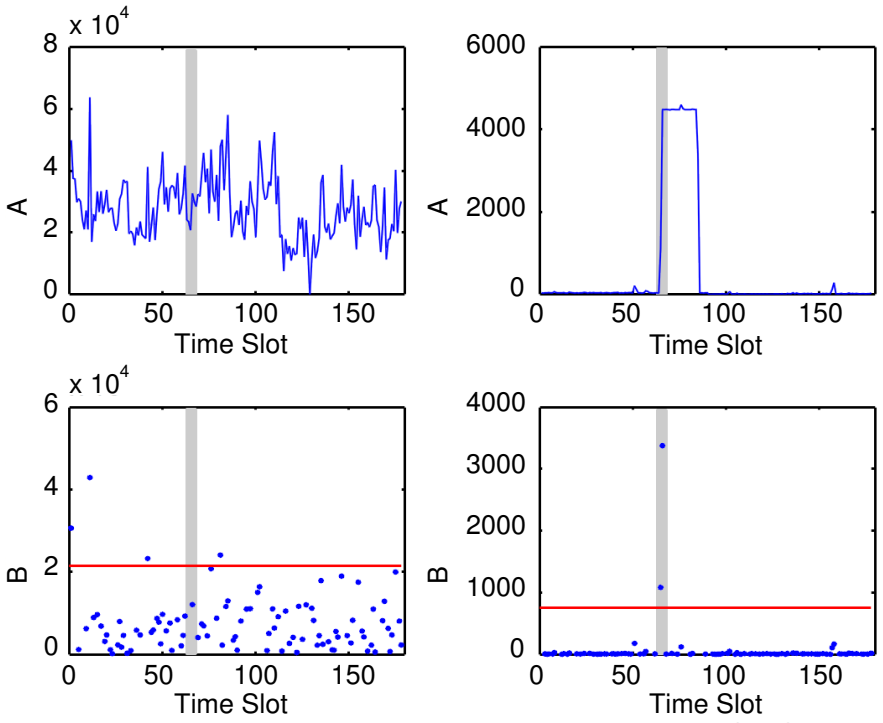

Figure 3.2: Low-intensity anomalies might be hidden inside highly aggregated traffic, but are visible at finer-grained aggregations. The DDoS attack is evident at the victim's network.

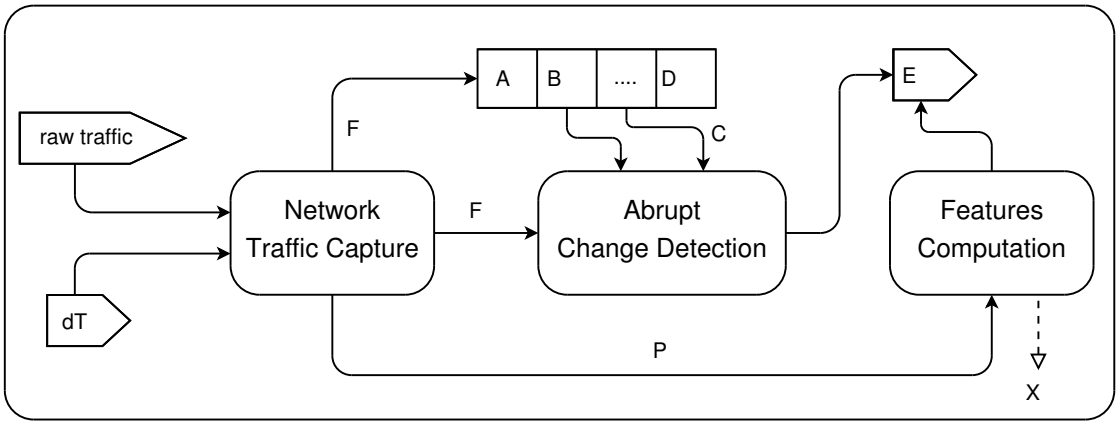

Figure 3.3: Multi-Resolution Change-Detection (MRCD) module functionalities and interactions.

detection thresholds $\lambda(j)$, $j = 1, \ldots, 4$ when no anomalies have been flagged during the last M time slots, according to equation (3.1).

The Features Computation sub-module verifies the existance of an anomalous time slot every $\Delta\mathtt{T}$ seconds through the anomaly $flag$ variable. In case of anomaly detection, the `<compute_features>` function in Fig. 3.6 computes the matrix of traffic features $\mathbf{X}$ describing the set of macro-flows in the anomalous time slot, using as input the traffic structure P_t computed by the Traffic Capture sub-module in the last time slot.

```
/**
* Capture network traffic
* @param duration of periodical capture
* @param network interface where to capture traffic
* @return set of raw traffic packets
*/

  struct* P = capture_raw_traffic(double \Delta_T, char* iface)

/**
* Traffic aggregation and time-series computation
* @param set of raw traffic packets
* @param macro-flow resolution
* @return multi-variable time-series sample
*/

  struct* Z = aggregate_traffic(struct& P, char* resolution)
```

Figure 3.4: MRCD traffic capture sub-module API.

```
/**
* Abrupt change-detection
* @param multi-variable time-series, current sample
* @param multi-variable time-series, previous sample
* @return indication of anomaly
*/

  boolean flag = change_detection(struct& Z_t, struct& Z_{t-1})

/**
* Update of change-detection thresholds
* @param multi-variable time-series, last M anomaly-free samples
* @return updated detection thresholds
*/

  double* \lambda = update_detection_thresholds(struct& Z_{t-1},
                        struct& Z_{t-2},..., struct& Z_{t-M})
```

Figure 3.5: MRCD abrupt change detection sub-module API.

The macro-flow resolution used in the computation of features depends on two criteria, either using the coarsest resolution in which the anomaly was detected, or any other pre-defined resolution, depending on which kinds of attacks or anomalies are being tracked (highly distributed, N-to-1 or 1-to-N, etc.).

```
/**
* Features computation
* @param set of raw traffic packets
* @param macro-flow resolution
* @return traffic features space
*/

   double** X = compute_features(struct& P_t, char* resolution)
```

Figure 3.6: MRCD features computation sub-module API.

3.2.2 Unsupervised Analysis Module

The Unsupervised Analysis module is based on applying clustering techniques on $\mathbf{X}$. The objective of clustering is to partition a set of unlabeled patterns into homogeneous groups of similar characteristics, based on some measure of similarity. Our particular goal is to identify and to isolate the different macro flows that compose the anomaly flagged in the first module, both in a robust way. Unfortunately, even if hundreds of clustering algorithms exist [Jai10, DHS01], it is very difficult to find a single one that can handle all types of cluster shapes and sizes, or even decide which algorithm would be the best for our particular problem [FR98]. Different clustering algorithms produce different partitions of data, and even the same clustering algorithm provides different results when using different initializations and/or different algorithm parameters. This is in fact one of the major drawbacks in current cluster analysis techniques: the lack of robustness.

To avoid such a limitation, we have developed in [MCO11] a divide and conquer clustering approach, using the notions of Sub-Space Clustering (SSC) [PHL04] and multiple clusterings combination. The clustering algorithm combines the information provided by multiple partitions of $\mathbf{X}$ to improve clustering robustness and detection results. We use Sub-Space Clustering to produce multiple data partitions, applying the same density-based clustering algorithm to N different sub-spaces $\mathbf{X}_i \subset \mathbf{X}$ of the original space. Each of the N sub-spaces $\mathbf{X}_i \subset \mathbf{X}$ is obtained by selecting k features from the complete set of m attributes. To deeply explore the complete feature space, the number of sub-spaces N that are analyzed corresponds to the number of k-combinations-obtained-from-m.

NEWNADA uses low-dimensional sub-spaces; using small values for k provides several advantages: firstly, doing clustering in low-dimensional spaces is more efficient and faster than clustering in bigger dimensions. Secondly, density-based clustering algorithms provide better results in low-dimensional spaces [AGGR98], because high-dimensional spaces are usually sparse, making it difficult to distinguish between high and low density regions. Finally, results provided by low-dimensional clustering are more easy to visualize, which improves the interpretation of results by the network operator. We use therefore use $k = 2$, i.e., bi-dimensional sub-spaces, which gives a total of $N = m(m-1)/2$ partitions to combine.

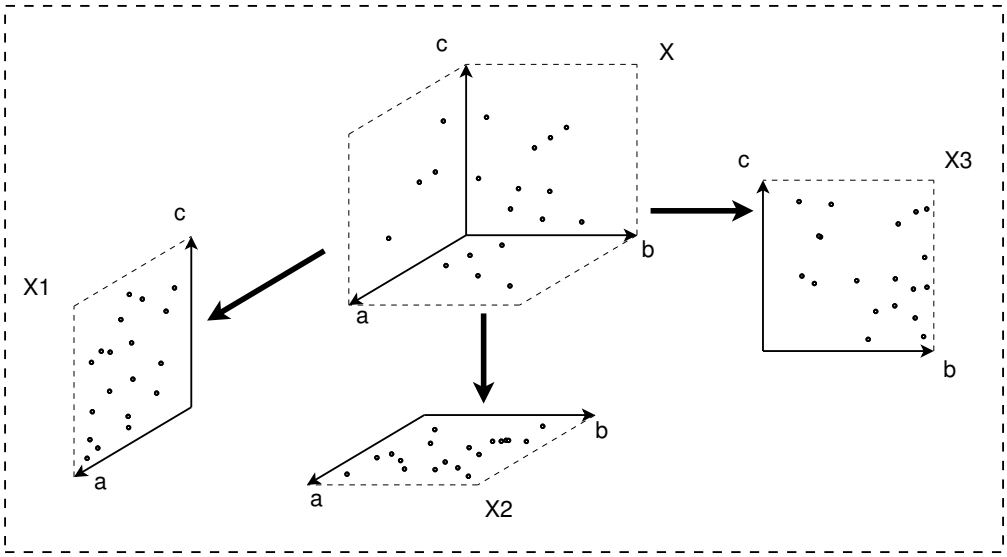

Figure 3.7: Sub-Space Clustering: 2-dimensional sub-spaces $\mathbf{X}_1$, $\mathbf{X}_2$, and $\mathbf{X}_3$ are obtain from a 3-dimensional feature space $\mathbf{X}$ by simple projection. Units in the graph are irrelevant.

Figure 3.7 explains this approach; in the example, a 3-dimensional feature space $\mathbf{X}$ is projected into $N = 3$ 2-dimensional sub-spaces $\mathbf{X}_1$, $\mathbf{X}_2$, and $\mathbf{X}_3$, which are then independently partioned via density-based clustering. Each partition is obtained by applying DBSCAN [EKSX96] to sub-space $\mathbf{X}_i$. DBSCAN is a powerful clustering algorithm that discovers clusters of arbitrary shapes and sizes [Jai10], relying on a density-based notion of clusters: clusters are high-density regions of the space, separated by low-density areas. This algorithm perfectly fits NEWNADA's unsupervised traffic analysis, because it is not necessary to specify a-priori difficult to set parameters such as the number of clusters to identify. The clustering result provided by DBSCAN is twofold: a set of p clusters $\{C_1, C_2, .., C_p\}$ and a set of q outliers $\{o_1, o_2, .., o_q\}$.

To combine the information obtained from the N partitions, NEWNADA uses the notions of multiple-clusterings Evidence Accumulation (EA) [FJ05]. EA uses the clustering results of multiple partitions to produce a new inter-patterns similarity measure which better reflects natural groupings. The algorithm follows a split-combine-merge approach to discover the underlying structure of data. In the **split** step, the N partitions are generated, which in our case they correspond to the SSC results. In the **combine** step, a new measure of similarity between patterns is produced, using a *weighting* mechanism to combine the multiple clustering results. The underlying assumption in EA is that patterns belonging to a *natural* cluster are likely to be co-located in the same cluster in different partitions. Taking the membership of pairs of patterns to the same cluster as weights for their association, the N partitions are mapped into a $n \times n$ similarity matrix S, such that $S(i, j) = n_{ij}/N$. The value n_{ij} corresponds to the number of times that pair of macro-flows $\{\mathbf{x}_i, \mathbf{x}_j\}$ was assigned to the same cluster in the N partitions. Note that if a pair of macro-flows $\{\mathbf{x}_i, \mathbf{x}_j\}$ is assigned to the same cluster in each of the N partitions then $S(i, j) = 1$, which corresponds to maximum similarity.

Algorithm 1 EA4C & EA4O for Unsupervised Anomaly Detection

Initialization:
  Set similarity matrix S to a null $n \times n$ matrix.
  Set dissimilarity vector D to a null $n \times 1$ vector.
for $l = 1 : N$ **do**
  $Q_l = \text{DBSCAN}\ (\mathbf{X}_l, \delta_l, n_{\min})$
  Update $S(i,j)$, $\forall$ pair $\{\mathbf{x}_i, \mathbf{x}_j\} \in C_k$ and $\forall\, C_k \in Q_l$:
    $w_k \leftarrow e^{-\gamma \dfrac{(n_l(k) - n_{\min})}{n}}$
    $S(i,j) \leftarrow S(i,j) + \dfrac{w_k}{N}$
  Update $D(i)$, $\forall$ outlier $o_i \in Q_l$:
    $w_l \leftarrow \dfrac{n}{(n - n_{\max_l}) + \epsilon}$
    $D(i) \leftarrow D(i) + \mathrm{d_M}(o_i, C_{\max_l})\, w_l$
end for
Rank macro-flows: D_{rank} = sort(D)
Set anomaly detection threshold: T_h = find-slope-break(D_{rank})
Find anomalous macro-flows: if $D_{rank}(i) > T_h \rightarrow$ anomalous macro-flow i.
Find anomalous macro-flows: find-max($S(i,j)$) $\rightarrow$ anomalous macro-flows i, j.

This EA algorithm is adapted for the particular tasks of anomaly detection and characterization of NEWNADA. By simple definition of what it is, an anomaly may consist of either outliers or small-size clusters, depending on the resolution of the macro-flows. Let us take a flooding attack as an example; in the case of a 1-to-1 DoS, all the packets of the attack will be aggregated into a single IP flow, which will be represented as an outlier in $\mathbf{X}$. If we now consider a DDoS launched from β attackers towards a single victim, then the anomaly will be represented as a cluster of β flows if the aggregation is done at $\mathrm{IPsrc}/32$ macro-flows, or as an outlier if the aggregation is done at $\mathrm{IPdst}/32$. Taking into account that the number of monitored flows can rapidly scale to thousands even for short time slots, the number of attackers β would have to be too large to violate the assumption of small-size cluster. Besides, if the attack is that massive ($\beta \approx n$), then it can be immediately detected by no mather which means.

The Unsupervised Analysis module is composed of two different EA methods to isolate small-size clusters and outliers: EA for small-clusters identification, EA4C, and EA for outliers identification, EA4O. Algorithm 1 presents the pseudo-code for both methods. EA4C assigns a stronger similarity weight when patterns are assigned to small-size clusters. The weighting function $w_k(n_l(k))$ used to update $S(i,j)$ at each iteration l takes bigger values for small values of $n_l(k)$, and goes to zero for big values of $n_l(k)$, being $n_l(k)$ the number of flows inside the co-assigned cluster for macro-flows pair $\{\mathbf{x}_i, \mathbf{x}_j\}$. The parameter $n_{\min}$ specifies the minimum number of flows that can be classified as a cluster by the DBSCAN algorithm, while δ_l indicates the neighborhood distance between flows to identify dense regions. The parameter γ permits to set the slope of $w_k(n_l(k))$. Even tunable, the algorithm works with fixed values for $n_{\min}$, δ_l, and γ, the three empirically obtained.

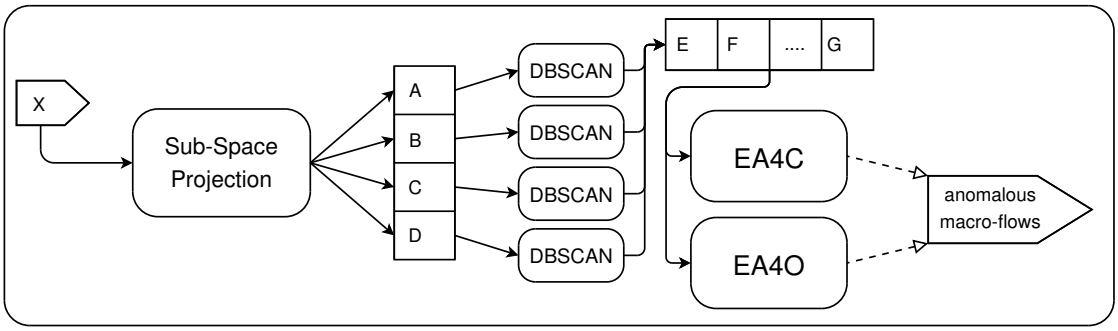

Figure 3.8: Unsupervised Analysis (UA) module functionalities and interactions.

EA4O works with a dissimilarity vector D where the distances from all the different outliers to the centroid of the biggest cluster identified in each partition (referred to as $C_{\max_l}$) are accumulated. The algorithm clearly highlights those outliers that are far from the normal-operation traffic in the different partitions, statistically represented by $C_{\max_l}$. The weighting factor w_l takes bigger values when the size $n_{\max_l}$ of $C_{\max_l}$ is closer to the total number of patterns n, meaning that outliers are more rare and become more important as a consequence. The parameter ϵ is simply introduced to avoid numerical errors ($\epsilon = 1e^{-3}$). Finally, instead of using a simple Euclidean distance, EA4O computes the Mahalanobis distance $\mathrm{d_M}(o_i, C_{\max_l})$ between the outlier and the centroid of $C_{\max_l}$, which is an independent-of-features-scaling measure of similarity.

In the final **merge** step, any clustering algorithm can be applied to matrix S to obtain a final partition of $\mathbf{X}$ that isolates small-size clusters. As we are only interested in finding the smallest-size clusters, the detection consists in finding all the macro-flows with the same, biggest similarity value in S. Regarding outliers detection, macro-flows are ranked according to the dissimilarity obtained in D, and an anomaly detection threshold T_h is set. The computation of T_h is simply achieved by finding the value for which the slope of the sorted dissimilarity values in D_{rank} presents a major change. Anomaly detection is finally done as a binary thresholding operation on D: if $D_{rank}(i) > T_h$, the system flags an anomaly in macro-flow i.

The functionalities of the Unsupervised Analysis (UA) module are accomplished by four different sub-modules, depicted in Fig. 3.8. The Sub-Space Projection sub-module computes the N bi-dimensional sub-spaces $\mathbf{X}_i$ for the multiple clustering analysis. The function `<compute_sub_space>` depicted in Fig. 3.9 computes the projection of $\mathbf{X}$ into the bi-dimensional sub-space defined by the `<dims>` features.

The DBSCAN sub-module performs the clustering analysis on each single sub-space $\mathbf{X}_i$ through the `<dbscan>` function, see Fig. 3.10. DBSCAN parameters $n_{\min}$ and δ_i are automatically computed by the `<dbscan>` function itself: $n_{\min}$ is set at the initialization of the algorithm, simply as a fraction α of the total number of flows n to analyze ($\alpha = 5\%$ of n); δ_i is set as a fraction of the average distance between the macro-flows in sub-space $\mathbf{X}_i$ ($1/10$), which is estimated from 10% of the macro-flows, randomly selected. This permits to fast-up computations. Each partition Q_i computed for each of the N sub-spaces $\mathbf{X}_i$ is stored in a buffer, which is then fed to the EA4C and

```
/**
* Sub-Space computation
* @param traffic space to project in sub-spaces
* @param dimensions of the sub-space
* @return traffic sub-space
*/

       double** X_i = compute_sub_space(double& X, int* dims)
```

Figure 3.9: UA sub-spaces computation sub-module API.

EA4O algorithms. This buffer is additionally used by the Characterization module (see Sec. 3.2.3) to compute the traffic signatures for the identified anomalous macro-flows.

```
/**
* Cluster analysis through DBSCAN
* @param traffic space to partition
* @param minimum number of macro-flows in a cluster
* @param neighborhood distance to identify dense regions
* @return set of clusters and outliers
*/

struct* Q_i = dbscan(double& X_i, double n_min, double \delta_i)
```

Figure 3.10: UA DBSCAN clustering sub-module API.

The last step of the Unsupervised Analysis module is performed by the EA4C and the EA4O sub-modules. Fig. 3.11 depicts the functions <find_anomalies_EA4C> and <find_anomalies_EA4O> that compose the API of these sub-modules, which implement the algorithms presented in Alg. 1. Vectors <int* I> and <int* O> contain the indices of the macro-flows which are identified as anomalous.

3.2.3 Characterization Module

The Characterization module permits to automatically produce a set of K filtering rules $f_k(\mathbf{X})$, $k = 1, .., K$ to correctly isolate and characterize the macro-flows detected as anomalous. On the one hand, such filtering rules provide useful insights on the nature of the anomaly, easing the analysis task of the network operator. On the other hand, different rules can be combined to construct a signature of the anomaly, which can be directly exported towards standard signature-based security and anomaly detection/prevention devices such as IDSs, IPSs, and/or Firewalls.

In order to produce filtering rules $f_k(\mathbf{X})$, the algorithm selects those sub-spaces $\mathbf{X}_i$ where the separation between the anomalous macro-flows and the rest of the traffic is the biggest. The characterization defines two different classes of filtering rule:

```
/**
* Evidence Accumulation to identify small-size clusters
* @param N sets of clusters and outliers
* @return indices of most-similar anomalous macro-flows
*/

       int* I = find_anomalies_EA4C(struct& Q_1,.., struct& Q_N)

/**
* Evidence Accumulation to identify outliers
* @param N sets of clusters and outliers
* @return indices of outlying anomalous macro-flows
*/

       int* O = find_anomalies_EA4O(struct& Q_1,.., struct& Q_N)
```

Figure 3.11: UA Evidence Accumulation $\mathrm{EA4C}$ and $\mathrm{EA4O}$ sub-modules API.

```
/**
* Computation of filtering rules
* @param indices of most-similar anomalous macro-flows
* @param indices of outlying anomalous macro-flows
* @param N sets of clusters and outliers
* @return absolute rules and sorted relative rules
*/

struct* FR = get_filtering_rules(int& I, int& O, struct& Q_1,.., Q_N)

/**
* Generation of signatures
* @param absolute filtering rules and sorted relative rules
* @param max number of relative rules to combine
* @return anomaly signatures
*/

  struct* Sig = combine_filtering_rules(struct& FR, int K)
```

Figure 3.12: UA Anomaly Characterization sub-module API.

absolute rules $f_A(\mathbf{X})$ and *relative* rules $f_R(\mathbf{X})$. Absolute rules are only used in the characterization of small-size clusters. These rules do not depend on the separation between macro-flows, and correspond to the presence of dominant features in the macro-flows of the anomalous cluster. An absolute rule for a certain feature j has the form $f_A(\mathbf{X}) = \{\mathbf{x}_i \in \mathbf{X} : x_i(j) == \lambda\}$. For example, in the case of an ICMP flooding attack, the vast majority of the associated flows use only ICMP packets, hence the absolute filtering rule $\{\mathrm{nICMP/nPkts} == 1\}$ makes sense.

On the contrary, relative filtering rules depend on the relative separation between

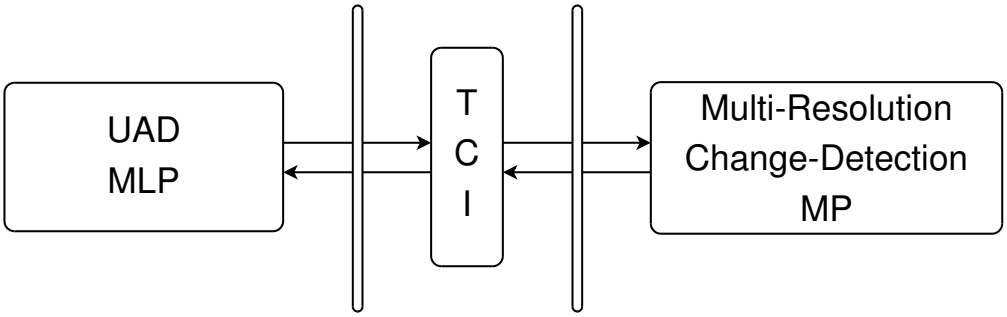

Figure 3.13: NEWNADA XORP processes within the EUA.

anomalous and normal-operation macro-flows. Basically, if the anomalous flows are well separated from the rest of the clusters in a certain partition Q_i, then the features of the corresponding sub-space $\mathbf{X}_i$ are good candidates to define a relative filtering rule. A relative rule defined for feature j has the form $f_R(\mathbf{X}) = \{\mathbf{x}_i \in \mathbf{X} : x_i(j) < \lambda \text{ or } x_i(j) > \lambda\}$. The characterization also defines a *covering relation* between filtering rules: rule f_1 *covers* rule $f_2 \leftrightarrow f_2(\mathbf{Y}) \subset f_1(\mathbf{Y})$. If two or more rules overlap (i.e., they are associated to the same feature), the algorithm keeps the one that covers the rest.

In order to construct a compact signature for the anomaly, the module selects the most discriminant filtering rules. Absolute rules are important, because they define inherent characteristics of the anomaly. As regards relatives rules, their relevance is directly tied to the degree of separation between flows. In the case of outliers, the K features for which the Mahalanobis distance to the normal-operation traffic is among the top-K biggest distances are selected. In the case of small-size clusters, the degree of separation to the rest of the clusters is ranked by using the well-known Fisher Score (FS), and the top-K ranked rules are selected. The FS measures the separation between clusters, relative to the total variance within each cluster. Given two clusters C_1 and C_2, the Fisher Score for feature i can be computed as:

$$F(i) = \frac{(\bar{x}_1(i) - \bar{x}_2(i))^2}{\sigma_1(i)^2 + \sigma_2(i)^2} \tag{3.2}$$

where $\bar{x}_j(i)$ and $\sigma_j(i)^2$ are the mean and variance of feature i in cluster C_j. In order to select the top-K relative rules, the K features i with biggest $F(i)$ value are kept. To finally construct the signature, the absolute rules and the top-K relative rules are combined into a single inclusive predicate, using the covering relation in case of overlapping rules.

Absolute and relative filtering rules and anomalous macro-flow signatures are computed by the `<get_filtering_rules>` and `<combine_filtering_rules>` functions respectively, see Fig. 3.12. The computation of filtering rules takes as input the anomalous macro-flows flagged by the EA4C and EA4O sub-modules, as well as the set of N partitions Q_i generated by the DBSCAN clustering sub-module. The resulting filtering rules are finally combined to obatin the signatures that characterize the flagged anomalies.

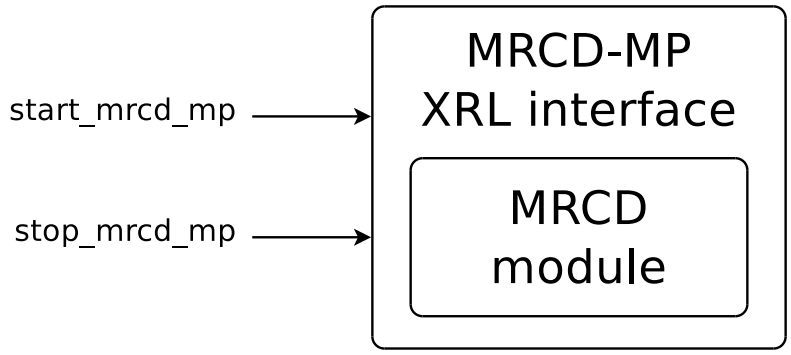

Figure 3.14: MRCD Monitoring Point XORP process.

3.3 Implementation

In this section we describe the implementation of NEWNADA in XORP within the EUA framework, as described by the design presented in Sec. 3.2. NEWNADA is composed of two different XORP processes: a Multi-Resolution Change-Detection Monitoring Point process (MRCD-MP), and an Unsupervised Anomaly Detection and characterization Machine Learning Process (UAD-MLP). Fig. 3.13 depicts NEWNADA XORP processes within the EUA. The UAD-MLP dispatches methods to the MRCD-MP through its local TCI, using the TCI's `<dispatch_push>` method. This method allows the MLP to receive continuous updates from the local MRCD-MP. In current implementation of NEWNADA in XORP, both the MLP and the MP are intended to be run in the same local router where the anomaly detection and characterization is to be done. This restriction avoids the need to transmit the complete matrix of traffic features built by the MRCD features computation sub-module in Fig. 3.6 between remote XORP processes located in separated routers. Currently, the MRCP-MP locally serializes the matrix of traffic features $\mathbf{X}$ to a predefined destination, from which the UAD-MLP reads the features describing the macro-flows in the flagged time slot.

Figure 3.14 depicts the MRCD-MP process, which implements the functionalities of the MRCD module as described in Sec. 3.2.1. The XRL interface of the MRCD-MP in Fig. 3.15 provides two start/stop methods that control the traffic capture, the multi-resolution change-detection, and the features computation tasks. The XRL `<start_mrcd_mp>` permits to start capturing and analyzing traffic in time slots of fixed duration at the desired network interface. At the end of each time slot, the result of the analysis of the MRCD module is returned as a boolean flag indication of presence of anomalies in the last time slot. This information is periodically reported to the local UAD-MLP in the form of a boolean anomaly presence indication. When the MRCD module detects an anomaly, the anomaly flag indication goes to 1 (`<true>`) and the computed matrix of traffic features $\mathbf{X}$ for the corresponding anomalous time slot is locally saved. Network monitoring can be stopped by calling the XRL `<stop_mrcd_mp>` on the MRCD-MP.

Figure 3.16 depicts the "core" of NEWNADA, i.e., the UAD-MLP process, which implements the functionalities of the Unsupervised Analysis and Characterization modules described in Secs. 3.2.2 and 3.2.3. The XRL interface of the UAD-MLP in Fig. 3.17 provides a single method that controls the SSC and EA analysis of the macro-flows de-

```
interface newnada_mrcd_mp/0.1 {
  /**
   * Start the multi-resolution change-detection module
   * @param duration of the time slot for traffic analysis
   * @param network interface of analysis
   * @param boolean indication of detected anomaly
   */
  start_mrcd_mp?duration:u32&iface:txt->flag:bool;

  /**
   * Stop the multi-resolution change-detection module
   * @param network interface of analysis
   */
  stop_mrcd_mp?iface:txt;
}
```

Figure 3.15: MRCD Monitoring Point process XRL interface.

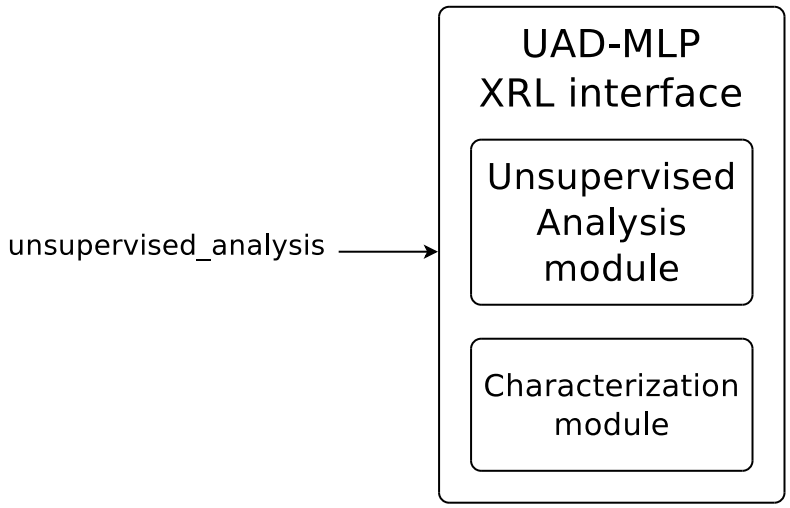

Figure 3.16: UAD Machine Learning XORP process.

scribed by the features' space $\mathbf{X}$, as well as the construction and selection of filtering rules. The XRL `<unsupervised_analysis>` is provided as a callback to the dispatched XRL `<start_mrcd_mp>` on the MRCD-MP, and it is used by the MP to periodically report the result of the Multi-Resolution Change-Detection analysis at the end of each time slot. The `<unsupervised_analysis>` method takes as input the anomaly boolean flag from the MP, and reads the matrix of traffic descriptors $\mathbf{X}$ in case of anomaly indication. A list containing the identified anomalous macro-flows' IPs, as well as a list of signatures describing them are finally returned, completing NEWNADA's anomaly detection and characterization tasks.

```
interface newnada_uad_mlp/0.1 {
  /**
   * Performs the unsupervised anomaly detection and characterization tasks
   * @param boolean indication of detected anomaly
   * @param list of anomalous macro-flows detected
   * @param list of signatures built for the detected anomalous macro-flows
   */
  unsupervised_analysis?flag:bool->anomalous_flows:list<ipv4net>&signatures:list<txt>;
}
```

Figure 3.17: UAD Machine Learning Process XRL interface.

3.4 Conclusion

We have presented *Unsupervised Network Anomaly Detection Algorithm* (NEWNADA). NewNADA is an unsupervised machine-learning based system conceived to meet the objective of automatic detection and characterization of intrusions and attacks/anomalies. NEWNADA design relies on three modules. A first module to capture traffic and anomalous changes. The second module determines the traffic flows responsible of anomaly, without relying on signatures or calibration. Finally. Third, a module that produces filtering irules for the classified flows.

Chapter 4

Path availability and IDIPS

4.1 Introduction

ISP-Driven Informed Path Selection (IDIPS) is our service to meet path availability and performance objectives. IDIPS design is described in Sec. 4.2. We further discuss how our implementation is included in the EUA (Sec. 4.3). Next, we explain in details how to build simple cost functions and combine them to reflect more complex ranking strategies (Sec. 4.3.1). Finally, we provide example of module implementations in Sec. 4.3.2 and conclude in Sec. 4.4. An evaluation of the proposed design can be found in deliverable D4.3.

IDIPS is a data collection and interpretation service that can be used by the other components of the EUA to control the traffic and achieve cooperation. The cost functions (Sec. 4.3.1) influence the way the traffic is controlled. Indeed, the ranks provided by IDIPS are directly used by the components of the EUA in charge of controlling the traffic.

4.2 System design

As illustrated by Fig. 4.1 IDIPS is composed of three independent modules: the *Querying* module, the *Prediction* module, and the *Measurement* module. The Querying module is directly in relation with the client as it is in charge of receiving the requests, computing the path ranking based on traffic qualification provided by the client, and the ISP traffic engineering requirements, and replying with the ranked paths. For the sake of generality, the remainder of this section will use the term *ranking criterion* when referring to traffic qualification. The Measurement module is in charge of measuring path performance metrics if required. Finally, the Prediction module is used to predicting paths performance (i.e., future performance metrics of a given path based on the past measurements).

The Measurement module is the data collection component of IDIPS and the Pre-

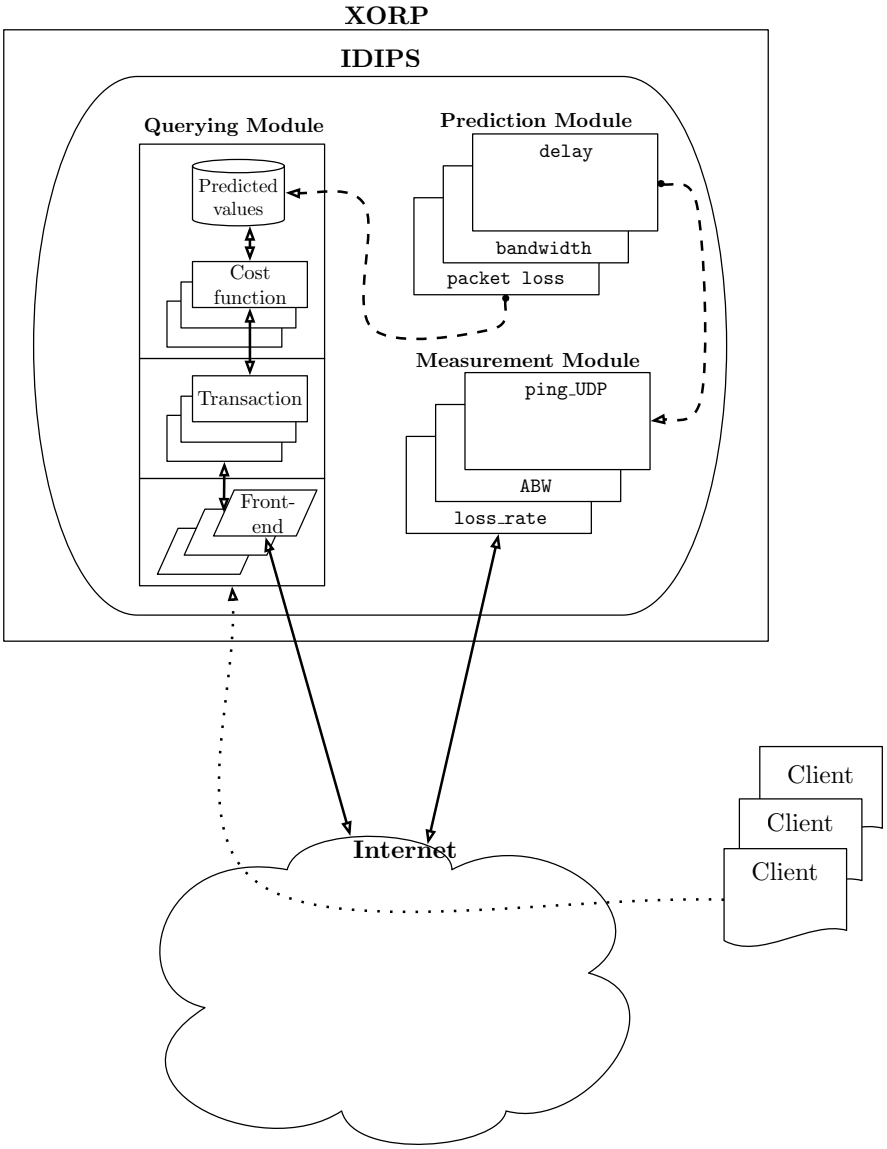

Figure 4.1: IDIPS within the EUA

diction module is the component that is in charge of interpreting the collected data. As described later, the Prediction module is used to control the way the measurements are performed. In other words, the Prediction module and the Measurement module form a feedback loop allowing for optimal measurements.

The ranking criterion provided by clients in their requests might require measuring the network to obtain path performance metrics, such as delay or bandwidth estimation. One of the key advantages of IDIPS is that it avoids clients measuring themselves the network, leading to redundant traffic injected in the network. The Measurement module performs the measurements or asks a third-party to perform the measurements. Those measurements can be active (i.e., probes are sent in the network) or passive (i.e., no additional traffic is injected).

It is possible to predict the performance of a given path if it has been previously measured [YRCR04, DCKM04, PLMS06, Pap07, dLUB05, WSS05, LPS06, LGS07, LHC03, LGP+05, NZ04, FJP+99, NZ02, PCW+03, ST03, LHC05, CCRK04, RMK+08, MS04]. This prediction task is achieved by the Prediction module. Note that a given measurement can be used in several different predictions. For instance, the previous delay measurements can serve for predicting the delay, the jitter, or for determining whether the path is reachable or not.

To enable flexibility, ease of implementation and performance[1], IDIPS clearly separates the Querying, Measurement, and Prediction modules. Each instance module communicates with the other modules thanks to a standardized interface. Therefore, the handling of requests from the clients is strictly separated from the prediction of path performance and path performance prediction is separated from path measurements.

The Querying module receives the ranking requests from the clients and computes the rank for these requested paths based on their predicted future performance. Future paths performance are estimated by the Prediction module that relies on the measurements performed by the Measurement modules.

All along this section, we are using the terms measurements and predictions. However, they have to be understood in their very generic meaning. For IDIPS, a measurement corresponds to any information grabbed from the network. This definition encompasses active measurements like pings, passive measurements like Netflow information [Cla04], or even routing information like BGP feeds. Likewise, a prediction in IDIPS is an information that is likely to be valid in the upcoming future. Therefore, a prediction can be the result of very complex machine learning techniques but also very simple information like the originating AS of the path destination. In other word, a measurement is an information discovered in the past or just at present and a prediction is an information that is likely to be valid in the coming future.

To support as many requests per second as possible, the IDIPS modules are running independently of each others. This independence is ensured through the use of caches. Each module stores its processing results in its local cache. If another module requests a given result, a simple `get` in the appropriate cache will return it.

There may exist several instances of the Prediction and Measurement modules. For example, IDIPS can have a delay Measurement module, a bandwidth Measurement module, a delay prediction module, and a bandwidth prediction module. Deliverable D4.3 provides and evaluates example of Measurement and Prediction modules implementations.

4.2.1 Querying module

Common applications are only able to use one path at a time, even if several exist. In this case, the client only needs to know the very best path returned by IDIPS when it has no additional information about the paths. For this reason, the list of ranked path

[1] IDIPS must potentially handle many ranking requests simultaneously

```
sync_rank_paths
         ? sources & destinations & criterion
         -> ranked_paths_list & ttl
```

Figure 4.2: IDIPS server API for synchronous mode clients

is sorted by rank before being transmitted to the client. Then, the client can safely consider the first path of the list as the very best path (or one best path among all the best paths if several ones have the same lowest value). The other paths are returned only for resiliency (the best path is not valid for the client) or if the client uses the ranked list to refine a local decision. Sorting the paths simplifies the operation at the client.

Paths ranking is done with the use of *Cost Function*. For a given <source, destination> pair, the cost function returns a cost, i.e., a positive integer resulting from metrics combination of a given path. The lower the path cost, the more attractive the path. We chose the cost to be represented by a positive integer for its simplicity (i.e., no complex representation to be processed) and because operators are already used to translate their policies into integers with the BGP local-pref [RLH06]. By definition, the sum of several costs is also a cost. One can for example combine cost functions with an exponentially weighted sum in order to reflect complex strategies or politics as long as the result is rounded into a positive integer. Sec. 4.3.1 explains how to construct cost functions.

To support as many requests per second as possible, the IDIPS modules are running independently of each others. This means that the Querying module never has to wait for a path performance prediction to be computed by the Prediction module to compute the path ranking. When a prediction has to be retrieved by the Querying module, it calls a `get` on the Prediction module for the path attribute it is interested in. The attributes of a path are the predicted metric values as computed by the Prediction module for the path. For the sake of generality, any attribute is encoded as an integer. If an information is too complex to be represented with a single integer, it can always be represented as a set of integers. For example, an $< x, y >$ coordinates can be decomposed in the `x_coordinate` and the `y_coordinate` and a function that needs to use the coordinates just needs to retrieve the `x_coordinate` and the `y_coordinate` to reconstruct the full coordinates. Sec. 4.2.3 gives more details about the interface to retrieve path attributes from the Prediction module.

Depending on its needs, a client can query IDIPS in a *synchronous* or *asynchronous* way. In the synchronous mode, when a request is received by the IDIPS server, the server sends the list of ranked paths back to the client once computed. On the contrary, in the asynchronous mode, when a request is received by the IDIPS server, the server computes the paths ranking but does not send the list back to the client. The requester must explicitly send a special command to retrieve the list of ranked paths. The API that IDIPS presents to clients is depicted in Fig. 4.2 for the synchronous mode and in Fig. 4.3 for the asynchronous mode.

```
async_rank_paths
         ? sources & destinations & criterion
         -> tid

get_all_path_ranks
         ? tid
         -> ranked_paths_list & ttl

get_next_path_rank
         ? tid
         -> source & destination & rank & ttl & more

get_next_n_path_ranks
         ? tid & n
         -> ranked_paths_list & ttl & more

terminate_transaction
         ? tid
```

Figure 4.3: IDIPS server API for asynchronous mode clients

The commands are sent by the client to the server. When the client uses the asynchronous mode, it receives a transaction identifier (tid) back from the server. Every request received by a server is abstracted as a transaction. This tid is the identifier of that transaction on the server. This identifier is used for retrieving the list of ranked path with the `get_all_path_ranks`. If the ranking is not yet computed by the server when the `get_all_path_ranks` is received, an empty list of ranked paths and the invalid `0x0` ttl are returned. The server, in asynchronous mode, always returns immediately a result when it receives an `async_rank_paths` or a `get_all_path_ranks`. The client must then poll the server until it has retrieved the list. This behavior is used to avoid the server to maintain too much state about the clients, it only maintains ranking state (linked with tid). To avoid the need of client polling, signaling could be used to let the server inform the client that the transaction is ready but it thus means that the server must maintain state about the client, which is what we want to avoid while using the asynchronous mode. Polling is by definition avoided in the synchronous mode. It is worth to notice that a ranking call can be implemented as being blocking or non-blocking at the client side, independently of the client to server communication mode. The typical use of a blocking call is when the path to exchange data cannot be changed once the flow is started. Then, the best path must be used. The client must then wait for the path ranking before being able to exchange data. On the contrary, non-blocking call is used when the client can change the path it uses while exchanging data. For example, a shim6 [NB09] host starts exchanging data with a path arbitrarily selected by following the rules of RFC3484 [Dra03]. If the data transfer is long enough, shim6 could decide to switch to the best path computed by IDIPS. In this case, the flow can start as soon as possible, even if the path used to exchange data might be sub-optimal at the beginning.

To avoid this waste of resources, IDIPS also offers the possibility to retrieve one path at a time with the `get_next_path_rank` that returns the best path that has not yet been retrieved by the client. To use the best working path, the client can use the algorithm presented in Fig. 4.4 where `handle_path` is the client function that needs the path and that returns true when no more path is required. The `more` parameter returned by the `get_next_path_rank` indicates if there is still a path to retrieve for the transaction. Optionally, the client can explicitly ask IDIPS to terminate the transaction. If not, IDIPS should eventually terminate it automatically. Instead of considering retrieving the rankings one by one or all at a time, the more generic `get_next_n_path_ranks` is also proposed where the client specifies the number of paths that must be returned by IDIPS. The equivalent of GET_ALL_PATH_RANKS corresponding to a specified number higher or equal to the number of sources time the number of destinations while a value equal to one corresponds to the `get_next_path_rank`. However, in most of the cases, a client is interested by either one or all the paths.

Changing the paths to always use the ones with the best performance might result in oscillations [AAS03, GDZ06]. Mechanisms to avoid oscillations [AAS03, GDZ06] can be implemented in the Querying module. However, dealing with the oscillation problem is out of the scope of our study that focuses on the architectural part of the performance based traffic engineering problem.

```
more := true

WHILE more
DO
   (src, dst, rank, more) := get_next_path_rank(tid)
   IF handle_path(srcs, dst, rank)
   THEN
       STOP
   END
DONE

terminate_transaction(tid)
```

Figure 4.4: One-by-one path ranking retrieval algorithm

```
start_measurement ?  source & destination & interval

stop_measurement  ?  source & destination

set_interval      ?  source & destination & interval

get_measurements  ?  source & destination
                  -> measurements
```

Figure 4.5: Measurement module API

4.2.2 Measurement Module

The Measurement module is in charge of measuring the paths. The measurements can be active or passive. For example, an active measurement could be a ping while passive measurement could be the count of the number of TCP SYNs entering the network.

The Measurement module API presented in Fig. 4.5 is two fold. The `start_measurement`, `stop_measurement`, and `set_interval` commands determine the targets to measure while the `get_measurements` is used to retrieve the last measurements of a path.

Measurements are always defined between a source and a destination and are performed periodically (with a configurable interval between the measurements). In case of passive measurements, the sources and destinations as well as the passively obtained information are extracted periodically from the passively collected traces. The possibility to modify the interval of a measurement is not mandatory but is more convenient as it allows one to adapt the measurement rate dynamically without disrupting a measurement campaign. If such a command is not available, it means that the measured values must be stored outside the Measurement module. Indeed, without the `set_interval` command, the measurement has to be stopped, then re-started from scratch meaning that

```
start_prediction ?  path

stop_prediction  ?  path

get_prediction   ?  path
                 -> prediction
```

Figure 4.6: Prediction module API

all the state in the Measurement module instance is lost for this measurement. Finally, the `get_measurements` command returns all the measurements performed so far for the <source, destination>.

It is important to notice that the decision of measuring a path is done either by configuration or triggered by the Prediction module, not directly by the requests. However, the content of the requests can be seen as passive measurements and can be used to dynamically determine the paths to measure.

4.2.3 Prediction Module

The prediction module contains all the intelligence of IDIPS. Indeed, IDIPS is a service that aims at determining the best paths to use. However, determining the best path to use is a prediction exercise as the future behavior of a path is seldom known, particularly when considering inter-domain paths. Determining how to predict a path behavior is out of the scope of this section. This section presents, instead, how a Prediction module has to be implemented in IDIPS.

As already said earlier, IDIPS modules are running independently. However, the Querying module needs to know the path attributes computed by the Prediction module. In addition, the Prediction module has to know the path it has to predict the performance metric for. To this aim, the Prediction module provides the API presented in Fig. 4.6.

This API has two components. On the one hand, the `start_prediction` and `stop_prediction` commands are used to specify the path to predict performance metric for. On the other hand, the `get_prediction` command is used to retrieve the predictions.

`get_prediction` always returns a value. If the attribute value is not defined, an error or a meaningful default value is returned. For example, if the bandwidth of a path is not known, a default value of zero can be returned making the path less interesting than any other path.

The decision of measuring or predicting a path is highly related to the deployment policies, the topology, and the traffic. The decision of predicting a path is thus not provided by IDIPS but is considered case by case by the Prediction module or by configuration. There exist three ways of determining if a prediction has to be started

or stopped. First, an operator can manually determine the path to predict and uses `start_prediction` and `stop_prediction` commands to do so. Second, a Prediction module instance can determine by itself if a path is worth being measured or not. For example, if a Prediction module received enough `get_prediction` for a path it is not predicting yet, it can decide to start predicting it. In this second case, the `start_prediction` and `stop_prediction` commands are not used. Finally, a Prediction module instance can predict that a path has to be predicted and command another Prediction module to start predicting the path. For example, a Prediction module instance can be in charge of predicting if a path is important or not based on the traffic it carries. If the path is considered as important, it can ask to start the delay prediction for that particular important path.

To predict the future path behavior, a Prediction module often needs information from the Measurement module. Like the Querying module can retrieve a prediction with a simple `get`, the Prediction module can retrieve the measurements from the Measurement module with the `get_measurements` (see Sec. 4.2.2). The Prediction module can use the last measurements to predict the future behavior of a path. Based on the prediction and on its quality, the Prediction module can decide to modify the frequency at which a measurement has to be performed (with the `sec_interval` command) or ultimately to start or stop a measurement. In addition, because a Prediction module aims at providing the path performance for the near future, the `get_prediction` only returns one result as opposed to `get_measurements` that returns a list of measurements. Obviously, this API does not preclude an extended API that would return more information about the quality of the prediction (for example a TTL) or several predictions at once.

Path asymmetry is common in the Internet [PPZ+08] and some metrics like the bandwidth strongly depends on the followed direction. It is then important that the IDIPS Measurement and Prediction modules take this factor into account to accurately rank the paths.

4.3 Implementation

Sec. 4.2 presents a potential generic design for IDIPS. In this section, we present how we have implemented IDIPS within the XORP framework [HHK03].

As in the generic design presented in Sec. 4.2, our implementation is decomposed in the Querying, Prediction, and Measurement modules. The querying module is a single XORP process, while there are as many XORP processes as required to implement the Measurement and Prediction modules. For example, if IDIPS requires to measure the delay and the bandwidth, the Measurement module will contain two XORP processes. One implementing a delay measurement and the other implementing the bandwidth measurement. Fig. 4.1 shows the IDIPS design in the EUA, while Fig. 4.7 shows how the different modules interact with each others.

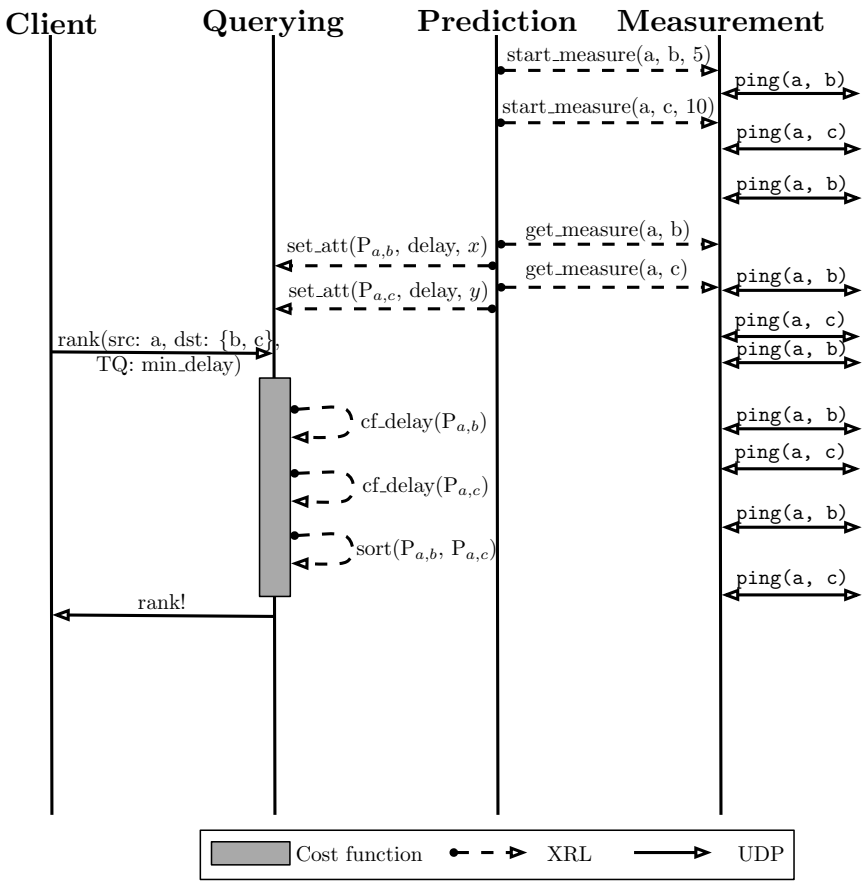

Figure 4.7: Example of modules interactions in IDIPS

The Querying module is decomposed in three main parts: (i) the *Front-end* part, (ii) the *Transaction* part, and (iii) the *Cost function* part. The Front-end part receives the requests from clients and returns the ranking results. The Transaction part processes the requests received by the Front-end and computes the path rank for these requests. Finally, the Cost function part implements the cost functions. Any EUA process can request paths ranking just by sending an XRL to the Front-end of the Querying module.

IDIPS must potentially handle many ranking requests at the same time. To support a potentially high load, requests are abstracted into *transactions*. Therefore, for each request, a transaction instance is created by a unique identifier. Each transaction runs independently of the others and maintains the list of sources, the list of destinations, and the path ranking criterion. If the request uses the synchronous mode, the transaction also maintains information to send the reply to the requester. When a request is received, the Front-end instantiates an empty transaction and adds all the paths from the request. At that stage, the paths are computed blindly: for each source s, for each destination d in the source and destination lists, the $< s, d >$ path is added to the transaction. Once all the paths have been added to the transaction, the `run` method is called on the instance.

The job of the `run` method is to determine the cost and the rank of each path, according to the path ranking criterion and to build the sorted list of ranked paths. A transaction is *ready* once the ordered list of ranked paths has been built completely. The cost of each path is determined by calling the appropriated cost function on the path. Once the cost is determined for a path, the path, tagged with its cost is added to the priority queue `_costs`. The `_costs` structure is maintained ordered by the path cost. It means that at any time, the ith entry in `_costs` has a lower or equal cost than the $i + 1$th entry. The transaction is set ready once the cost and rank of each path is known. If the request was in the synchronous mode, the transaction triggers the transmission of the reply to the request once the transaction becomes ready. If the request was in the asynchronous mode, the method stops. As long as the transaction is not ready, a call to retrieve a path for an asynchronous mode request returns an error.

We implemented the call of cost functions in two different ways. By using XRL or by directly calling the method on the querying module class instance. We use the XRLs to parallelize the processing. However, as the processing of XRL is centralized (via the `finder`) and because the management of XRLs is sequential and implemented with a list, this implementation does not improve the performance. Even worse, it reduces the number of requests IDIPS is able to sustain and may cause ranking failures because XRLs can be lost. Indeed, the XRLs are enqueued in a list limited in size. Therefore, once the list is full, XRLs can be lost. The performance can also drop because an XRL at position i in the queue will not be dispatched to the Querying module before the XRLs prior to position i have been dispatched to their target process. Even if the processes are different. With an experiment where the requests ask to rank 50 different paths, we observed a drop of 54% of requests per second supported by IDIPS compared to an implementation calling the cost function directly without XRLs. We also noticed about 12% of failing transactions and a time to compute the rank 56 times higher with the XRL implementation. However, for the requests that succeeded, the time perceived by the client was 13% faster with the XRL implementation. The time perceived by the client is the time elapsed between the sending of the request and the reception of

the ordered list of ranked paths. Despite the better client perceived time with the XRL implementation, we recommend not to use the XRL implementation. Indeed, without the use of XRLs, IDIPS can handle more simultaneous requests and does not face loss of requests due to the limited size of the XRL queue.

Sec. 4.2 proposes to keep the modules independent thanks to the use of getter functions: when a module needs information from another module, it sends a get to the module to retrieve the values. In our implementation, every module implements such getters. However, we also implemented a path attributes cache within the Querying module. This cache stores, for each path, all the known attributes for the paths. The attribute values are computed by the Prediction module. This cache is based on a *push* model. It means that it is not the Querying module that populates it but the Prediction module that pushes the values to that cache. The querying module thus implements the `set_attribute` and `get_attribute` XRLs. Therefore, when a prediction is computed, the Prediction module immediately calls the `set_attribute` XRL on the Querying module to set the attribute value for the path that as just been computed. This mechanism is implemented to speed-up the cost computation for the paths. Indeed, as presented in Sec. 4.3.1, the cost of a path is computed with a cost function that potentially needs the attributes of the path. Thus, without an attribute cache at the Querying module, an XRL must be called to the appropriate Prediction module instance for each attribute to retrieve. However, calling XRL implies some delay that can be non negligible if the Prediction module is not running on the same host as the Querying module. For this reason, the Querying module does only rely on this cache. If the cache has no entry for the path attribute, it is considered that the path is not under measurement/prediction and the Cost function must determine an appropriate cost. It is important to remark that our implementation does not allow the prediction module to determine by itself that a path merits to be predicted. Indeed, the Querying module never calls the `get_attribute` on the prediction module. So, the prediction module cannot count the number of failing calls. However, one could imagine a Measurement module instance monitoring the cache misses at the querying module. The prediction module could then determine the paths that are worth being predicted.

The notion of module is translated into XRL interfaces in EUA. Except for the Querying module, there might be several C++ classes implementing a module and possibly several instances of a class as illustrated in Fig. 4.1. Each class must implement the XRL interface corresponding to the module it is related to. Fig. 4.8 gives the XRLs that must be implemented by the class implementing the Querying module. It is important to notice that the interface for the querying module is only composed of the setter and the getter for the path attributes. It does not include an interface for clients to query IDIPS. Indeed, XRL interfaces are only related to the implementation. Nevertheless, we implemented the client-related commands described in Fig. 4.2 and Fig. 4.3 with XRLs to make IDIPS usable directly by any process in the EUA. Fig. 4.9 lists the XRLs that must be implemented by the classes implementing the Measurement module. Finally, Fig. 4.10 shows the XRLs that the classes implementing the Prediction module must implement. Each class implementing one and only one technique. For example, one class can implement a UDP ping for the Measurement module and another can measure the path bandwidth and one class can implement a delay bandwidth product predictor

```
interface idips_querying/0.1 {
  /**
   * Get a path attribute
   * @param path to get the attribute from
   * @param name of the attribute
   * @param value of the path attribute
   * @param rpath echo of path
   */
  get_attribute?path:txt&name:txt->value:u32&rpath:txt;

  /**
   * Set a path attribute
   * @param path to set the attribute to
   * @param name of the attribute
   * @param value of the path attribute
   */
  set_attribute?path:txt&name:txt&value:u32;
}
```

Figure 4.8: Querying module XRL interface

based on the delay and bandwidth measurements.

The whole process is presented in Fig. 4.7. The Prediction module asks an instance of the Measurement module (i.e., the delay measurement instance) to measure a path. A path to measure is defined by a source and a destination. For the sake of generality, the source and the endpoint of any path to measure is represented textually, meaning that it can be a name, an IP address of a network interface, or any other suitable information. Each path installed in a Measurement module is periodically measured with a configurable interval between measurements (e.g., 5 for path (a,b) and 10 for (a,c) in Fig. 4.7). The use of IP prefixes instead of IP addresses is particularly interesting to aggregate information. For example, if a site has one IP prefix p/P for its clients and that the performance are considered to be the same for any of them, then all the paths can be aggregated by using the p/P source instead of the client IP address.

The `start_measurement` XRL function triggers the measurement of the path defined by the `source` and `destination` parameters. The path is then measured every `interval` seconds (e.g., 5 for the path (a,b) in Fig. 4.7).[2]

The various Measurements module instances keep locally the last measurements they obtained for the paths they are measuring. When a Prediction module needs a measurement, it sends a `get_measurement` XRL to the adequate instance of the Measurement module and retrieves the measurements for the path. The measurement is then sent to the Querying module, with the `set_attribute` function, for being stored in the Predicted values storage.

[2]To avoid synchronization, the time between two measurements should be set to be equal to the `interval` parameter on average.

```
interface idips_measurement/0.1 {
  /**
   * Start periodically measuring a destination
   * @param destination destination to measure
   * @param interval interval in seconds between two measurements
   */
  start_measurement?source:txt&destination:txt&interval:u32;

  /**
   * Stop measuring a destination
   * @param destination destination to stop measuring
   */
  stop_measurement?source:txt&destination:txt;

  /**
   * Change measurement interval for a destination
   * @param destination destination to change the measurement interval
   * @param interval new measurement interval for the destination
   */
  set_interval?source:txt&destination:txt&interval:u32;

  /**
   * @params destination  destination to get the past measurements
   * @params measurements list of measurements
   * @params clean remove elements after retrieving them
   */
  get_measurements?source:txt&destination:txt
              &clean:bool->measurements:list<u32>;
}
```

Figure 4.9: Measurement module XRL interface

```
interface idips_prediction/0.1 {
  /**
   * Start a prediction model for a path
   * @param path to predict
   * @param src source IP for the measurements
   * @param dst destination IP for the measurements
   */
  start_prediction?path:txt&src:ipv4&dst:ipv4;

  /**
   * Stop a prediction model for a path
   * @param path to stop the prediction for
   */
  stop_prediction?path:txt;

  /**
   * Get the prediction for a path
   * @param path to get the prediction for
   * @param prediction for the path
   */
  get_prediction?path:txt->prediction:u32;
}
```

Figure 4.10: Prediction module XRL interface

4.3.1 High Level Cost Functions Implementation

In this section, we show how to construct simple fundamental cost functions and how to combine them to implement an ISP policy. Our example is based on a situation in which an ISP has three customer families: (i) *premium users* always requiring the best available performances, (ii) *standard users* requiring a good performance/cost trade off, and (iii) *light users* always requiring the lowest cost. The traffic engineering changes between the night and the day for standard users: during the day, a lower cost is preferred while during the night, the performance is preferred. The monetary cost of a path depends on the 95^{th} percentile load of the link used to reach the Internet.

In our example, we assume that the prediction module feeds the querying module with the following information:

- routing reachability of the paths. A path is reachable if there exists a route in the FIB to forward traffic from its source to its destination, this information is stored in the `REACHABILITY` attribute
- originating ASN. The originating Autonomous System Number (ASN) of a path is the originating AS number of the prefix of the destination as discovered by BGP. This information is stored in the `ORIGIN` attribute
- monetary cost of the paths. The monetary cost of a path is the expected cost it would represent to carrying one additional Mega bit per second of traffic on it.

Algorithm 2 Example of cost function for the reachability

Ensure: Integer value representing the result of this Cost Function. is_reachable_cfsrc, dst

```
1: reachable ← get_attribute(<src,dst>, REACHABILITY)
2: return reachable
```

Algorithm 3 Example of cost function for the path locality

Ensure: Integer value representing the result of this Cost Function. locality_cfsrc, dst

```
1: origin ← get_attribute(<src,dst>, ORIGIN)
2: if origin = LOCAL_ASN then
3:    return 0
4: end if
5: return 1
```

Algorithm 4 Example of cost function for the cost minimization

Ensure: Integer value representing the cost of using the path defined by *src*, *dst*. minimize_cost_cfsrc, dst

```
1: cost ← get_attribute(<src,dst>, COST)
2: return cost
```

This cost is computed by applying the 95^{th} percentile technique [DHKS09] and is stored in the `COST` attribute

- available bandwidth of the paths. The available bandwidth of each path is estimated and is expressed in Mbps stored in the `ABW` attribute
- customer family. A customer can be premium, standard or light user. The customer family, stored in the `FAMILY` attribute, of a path is determined simply by considering the source of the path and ignoring its destination

We first have to define if a destination is reachable or not from a given source address. A path, defined by a <source, destination> pair, has its `REACHABILITY` attribute equal to 1 if it is reachable. Otherwise, the attribute is set to the maximum integer value. The cost function `is_reachable_cf`, implemented in Algorithm 2, thus makes reachable destinations more preferable than unreachable ones.

The locality of a path is determined by the originating AS number of the path destination. If the destination prefix is originated by the operator, the path is considered local. Algorithm 3 shows how to implement the `locality_cf` cost function that prefers local paths over non-local ones. In this function, `LOCAL_ASN` is operator AS number.

Algorithm 4 shows the `minimize_cost_cf` cost function that returns the monetary cost of using a path. This function makes path with the lowest monetary cost more attractive. To avoid oscillations, it is a good idea to use classes of monetary costs instead of the exact monetary cost. For example, the `COST` attribute could be the reminder of the division of the monetary cost by x instead of being the raw value of the monetary cost.

Algorithm 5 Example of available bandwidth cost function

Ensure: Integer value representing the result of this Cost Function.

```
1: MAX_BW capacity of the network available_bw_cfsrc, dst
2: abw ← get_attribute(<src,dst>, ABW)
3: return (MAX_BW – abw)
```

Algorithm 6 Example of customer family cost function

Ensure: Integer value representing the customer family for traffic from *src* to *dst*. customer_family_cfsrc, dst

```
1: family ← get_attribute(<src,dst>, FAMILY)
2: return family
```

Algorithm 7 Example of a complex cost function

Ensure: Encounters customers requirements

```
1: PREMIUM_USER = 1
2: STANDARD_USER = 10
3: LIGHT_USER = 20 customer_management_cfsrc, dst
4: if (is_reachable_cf (src, dst) = MAX_INTEGER) then
5:    return (UNREACHABLE)
6: end if
7: customer ← CUSTOMER_FAMILY_CF(src, dst)
8: if (customer = PREMIUM_USER) then
9:    cost ← AVAILABLE_BW_CF(src, dst)
10: end if
11: if ((customer = STANDARD_USER ∧ DAY) ∨ customer = LIGHT_USER) then
12:    cost ← MINIMIZE_COST_CF(src, dst)
13: end if
14: if (customer = STANDARD_USER ∧ NIGHT) then
15:    cost ← AVAILABLE_BW_CF(src, dst)
16: end if
17: return ( LOCALITY_CF(src, dst) · cost ) + cost
```

When considering bandwidth, the best paths are those having the highest available bandwidth. The implementation of a cost function preferring paths with the highest bandwidth is not straightforward. Indeed, IDIPS, by definition, always prefers the lowest cost while in terms of bandwidth, the highest is the best. Thus, to prefer the paths with the highest bandwidth, the value of the available bandwidth is subtracted to the highest theoretical available bandwidth for the operator (i.e., the total network capacity). Algorithm 5 provides the implementation of such a cost function, `MAX_BW` being the highest theoretical available bandwidth in the network.

As for cost minimization, the customer family cost function only has to return the customer family. Algorithm 6 shows the implementation of this cost function. In the system, the family 1 corresponds to premium users, 10 is for standard users and 20 for light users.

The previous algorithms can be combined by the network operator to build more complex policies. Algorithm 7 combines all the blocks in order to reflect the operator policies proposed earlier in this section. In particular, Algorithm 7 first checks whether the destination `dst` is reachable from the source `src`. If the path is reachable, it applies the policies previously defined, based on the `FAMILY` attribute. For *premium* clients available bandwidth is always preferred. For *standard* clients the applied policy depends on the time period; the available bandwidth is used as cost function during the night, while cost minimization is preferred during the day.

The last line gives preference to a local paths. This line is an example of weighted sum of cost functions. More particularly, the cost result by the `CUSTOMER_MANAGEMENT_CF` is a weighted sum of the costs from other cost functions, weight by the cost returned by a cost function. The principle in the example is to double the cost if the path is not local.

4.3.2 Examples of IDIPS module implementation

This section presents two examples of module implementation. We first present a Measurement module that implements a UDP ping and then describe a Prediction module that implements an average delay predictor. The Prediction module uses the measurement module to predict the delay of the paths.

Measurement module example For the sake of the example, we propose a UDP ping Measurement module. This module does not aim at being used in a real environment where more robust measurements techniques should be used. To estimate the round-trip delay between a <source, destination> IP pair, we send a UDP segment to the destination on a port number that is very unlikely to be open. If the port is not opened and if no filtering applies, an ICMP port unreachable is expected to be returned to the Measurement module. The sending of the UDP segments is done by using the XORP socket API. XORP sockets are similar to the POSIX sockets except that they are asynchronous and that they are implemented with XRLs. In the reminder of this section we will use the term *socket* to refer to the XORP socket abstraction. A XORP process that wants to use a socket has to implement the `socket4_user`[3] XRL interface. This interface defines several XRL like `error_event` or `recv_event` that respectively indicate if an error occurred with the socket or if bytes are ready to be read on a socket. The `socket4_user` is used to signal the XORP process about events on the sockets it is in charge of. To open, bind, connect, listen, send data on or close a socket, the IDIPS must use an XRL Socket Client. XRL Sockets Clients are classes that implement the `socket4` XRL interface and are directly provided in the XORP framework.

To implement the UDP ping, we create one connected UDP socket per <source, destination> IP pair and periodically send a UDP segment with it. The time at which the packet is sent is stored for later use. Because the destination does not listen on the port, it sends an ICMP port unreachable that eventually triggers the call of the `error_event` XRL in our process. The error indicates on which socket the error arrives and the nature

[3] `socket6_user` for IPv6

of the error. The delay is thus simply computed by doing $now - measure$ where now is the time at which the XRL is called and $measure$ is the time at which the probe was sent.

The module needs to keep some state about the <source,destination> IP pairs it measures. To do so, different datastructure are required. First, the `_destinations` map maintains measurement information for each <source, destination> IP pair. This information contains the interval at which the pair must be measured and the list of the measured delays for the pair (the closer to the end of the list, the more recent the measurement). Once a delay has been measured for a pair, it is appended to its measured delay list. When the `get_measurements` command is called on the measurement module, this is the measured delay list for the requested pair that is returned. Two other datastructures are used to map a socket identifier to a pair and vice versa. The `_socket_info` maps gives information about the socket indexed by the socket identifier. The related information is the source and destination addresses and the time at which the last segment has been sent on this socket (the $measure$ variable). The `_sockets` map is the opposite of the `_socket_info`. `_sockets` gives the socket identifier for any pair. The `_socket_info` is unfortunately required as there exists no way in XORP to retrieve meta information on a socket like those we need.

The IP pairs are measured periodically. To implement these periodic probings, we use a XORP periodic timer. Every second, this timer calls the `loop` method of our process. When this method is called, a UDP segment is sent to each <source, destination> IP pair that should have been measured at the latest when the `loop` method is called. To efficiently determine the pairs to measure at the `loop` call, the `_to_measure` priority queue is maintained for each source covered by the measurement module. The key in the priority queue is the time at which the measurement has to be done and the value is the destination address. When a measurement is sent by the `loop`, the entry is removed from the priority queue and the next measurement time is computed for that entry. The new measurement time is then added to the priority queue. Fig. 4.11 shows the pseudo-code of the `loop` method.

Lines 17 – 21 ensure the measurement periodicity of <source, destination> pair that has not been stopped. The salt is used to avoid synchronization of measurements and is a small random value [AKZ99].[4] With Fig. 4.11, we can see that stopping a measurement by calling the `stop_measurement` does not apply immediately and an ultimate probe is sent after such a call. We can also see that there is never more that one entry par <source, destination> pair in the queue which is optimal from a memory point of view.

Fig. 4.12 shows how the ICMP port unreachable is processed by our module.

It is not possible, without changing XORP to associate a time to an event on a socket. This explains why line 0 is required in the algorithm of Fig. 4.12. The retrieval of the time has to be carried out as soon as possible to limit the inaccuracy of the delay estimation.

[4] In our implementation, the salt is zero.

```
00 FOREACH src IN _to_measure
01 DO
02      WHILE _to_measure[src] IS NOT EMTPY
03      DO
04          entry := _to_measure[src].pop
05
06          IF entry.key > NOW
07          THEN
08             MOVE TO NEXT SOURCE
09          END
10
11          dst := entry.address
12          socketid := _sockets[src][dst]
13          _socket_info[socketid].last_call := NOW
14
15          send_UDP_probe(socketid, src, dst)
16
17          IF (src, dst) NOT STOPPED
18          THEN
19              entry.key := NOW
                      + _dsts[src][dst].interval
                      + salt
20              _to_measure[src].push(entry)
21          END
22      DONE
23 DONE
```

Figure 4.11: Measurement module `loop` method pseudo-code

```
SOCKET4_USER_0_1_ERROR_EVENT(socketid, error)
00 now := NOW
01 IF error = ICMP_PORT_UNREACHABLE
02 THEN
03      si := _socket_info[socketid]
04      measure := si.last_call
05      delay := now - measure
06      _destinations[si.source][si.dst].measurements.append(delay)
07 END
```

Figure 4.12: UDP ICMP port unreachable management

Prediction module example The Measurement module presented above does delay measurement by the mean of UDP pings. The Prediction module example in this section uses the round-trip-delays measured by the UDP ping measurement module to predict the delay expected for the paths in the near future. The Prediction module simply averages the last round-trip-delays measured for a <source, destination> IP pairs. The average delay is the prediction of the delay for the path defined by the pair.

In this module, a path is defined by a source and a destination IP address. When a `start_prediction` command is received by the prediction module, it requests the UDP ping measurement module to start a measurement for the <source, destination> IP pair that defines the path the delay prediction has to be performed for. The prediction module then periodically retrieves the list of the last measurements for the path. Because the prediction module is the single one to use the UDP ping prediction module, it requests the measurement module to flush its memory. The prediction module then computes the average of the measured delays in the list. This average is considered as the future value of the delay until the next retrieval of the measurements list for the path.

The prediction module maintains two datastructures. On the one hand, the `_paths` map maps a path to a <source, destination> IP pair. On the other hand, the `_delays` map stores the predicted delay for each path.

To speed-up the Querying module processing, the prediction module also pushes the prediction delays to the Querying module path attributes collection. That is, when the Querying module needs the delay prediction, it does not need to request the prediction module. Doing so limits the use of XRLs and thus the number of context switches.

Our example implementation has no other intelligence. Indeed, the list of measurements is retrieved at the same rate for each path (once every 10 seconds thanks to a XORP periodic timer) and the prediction module requests the UDP ping measurement module to send a probe every second. However, it would not be a hard task to modify the module to enable an adaptive measurement rate and an adaptive measurements list retrieval.

4.4 Conclusion

IDIPS is our service to meet path availability and performance objectives. IDIPS architecture is composed of three modules. The Querying module receives the path ranking requests from the client (e.g., a routing protocol) and computes the cost for the paths contained in the request. The rank is based on the cost associated to the path and computed by the cost functions. The cost functions implement the network operator high level policies. Cost functions are fed by the prediction module. The prediction module uses path measurement information collected by the measurement module. The prediction module aims at predicting the future performance of the paths based on what has been observed in the past by the measurement module. As its name reveal it, the measurement module is a module in charge of measuring the paths. Measurements can be either active or passive. The frequency and the types of measurements that need to be

performed by the measurement module are decided by the prediction module. Indeed, the prediction module uses learning techniques to predict the future performance of the paths. By comparing the predicted value and the measured value, it is thus possible to determine the quality of the prediction and adapt the measurements accordingly.

IDIPS is designed to be flexible. First, measurements are abstracted into integers. So that anything that can be translated into an integer (or a set of integers) can be interpreted as a measurements by IDIPS. For example, the length of a BGP path is a measurement in the IDIPS sense. IDIPS is a path ranking mechanism and to remain as generic as possible, DPS also abstracts the path notion. A path is simply a source, destination pair. Sources and destinations are simple opaque keys. A source (or destination) can thus be an IP address, an IP prefix, or even a name. However, in the implementation, for efficiency reasons, we implemented this as IP prefixes. Finally, the rank is an abstraction of the costs. The lowest the rank value, the better the path. The rank is directly computed from the costs. The cost of a path is implemented by a cost function. A cost function abstracts a policy into a positive integer. The lowest the cost value of a path the better it is. Cost functions must respect the transitivity relationship, which is not the case of ranks. Ranks are abstractions of costs to hide computation and topology details to the clients.

A detailed evaluation of the IDIPS architecture can be found in D4.3.

Chapter 5

Network recovery & resiliency / OSPF SRG inference

5.1 Introduction

OSPF SRG inference is used to improve the recovery process of the OSPF protocol for multiple link failures. For the inference module to be able to cluster and data-mine failure occurences, these failures must first be detected by OSPF. Such functionality must be added to the OSPF protocol message processing routine, since the link-state advertisement (LSA) by definition only contain updates about the current routing topology as seen by the advertising routers. This means that in order to detect link failures, incoming LSAs must be correlated with LSAs received earlier. Any link failure detected this way will then be reported to the inference module using the XRL dispatch mechanism.

This failure data is used to create SRG inference tables which are then sent back to the OSPF module for future usage. Consequently, functionality to receive these tables and use them in the rerouting process again requires changes to the OSPF module XRL interface and process.

In Sec. 5.2, we show the general flow and information models of the OSPF/SRG inference integration. This include the high-level view of information exchange, and a definition of link failure and SRG table information. Sec. 5.3 has the implementation of these models into the EUA. This leads to a pair of data collection and control interfaces from and into OSPF. We als odiscuss changes from the earlier Xorp 1.6 implementation developed at IBBT.

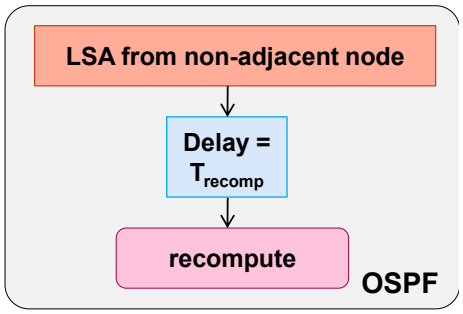

Figure 5.1: High-level flowchart for normal OSPF LSA processing

5.2 System design

This section describes the flow and information models that are used in the interaction between the OSPF and inference modules.

The original OSPF process is shown on Fig. 5.1. LSAs from non-adjacent nodes are processed as per section 13 of RFC 2328[Moy98], and then a recompute is scheduled after a certain hold-off time T_{recomp}. This hold-off time serves to provide additional routing stability. It allows for a certain batch of LSA updates (from several advertising routers) to arrive during the hold-off period, after which a route recompute is performed based on all newly received information. A recompute is not re-scheduled if a previous one was still scheduled. This means that a recompute will occur T_{recomp} seconds after a first of a batch of LSAs has been received, and that recomputes are spaced at least T_{recomp} seconds apart, limiting the rate of recompute that can be performed. If no LSAs are received after the last recompute, no further recomputes are scheduled normally; a stable network topology will generally only cause recomputes through the LSA aging (and expiration) process. The hard-coded default in Xorp for T_{recomp} is 1 second. For reference, the minimum HELLO interval that can be configured is also 1 second, with 10–40s being a typically used value. This means that T_{recomp} is mostly sufficiently low for regular operation of OSPF, still providing a fast enough response in terms of routing table and shortest path tree recomputation. Part of the SRG inference mechanism however will serve to improve reaction speed of OSPF and therefore a method to lower T_{recomp} without compromising OSPF stability will be presented.

On Fig. 5.2 we show a high level overview of the OSPF LSA reception and route recomputation process, integrated with the normal OSPF flow. Some functionality and checking is added in-between the LSA reception (and link-state database insertion) and shortest-path tree recompute.

Upon reception of an LSA, the LSAs are now correlated with existing link-state database entries. A general router LSA will contain a number of advertisements for links (which can be of several types). These are stored into the link-state database as per normal OSPF procedure. The occurrence of a link failure is defined as the disappearance of such a link from a certain router LSA, i.e., when a certain link is present in the link-state database, but not in the newly received LSA. When a link failure is detected, it is

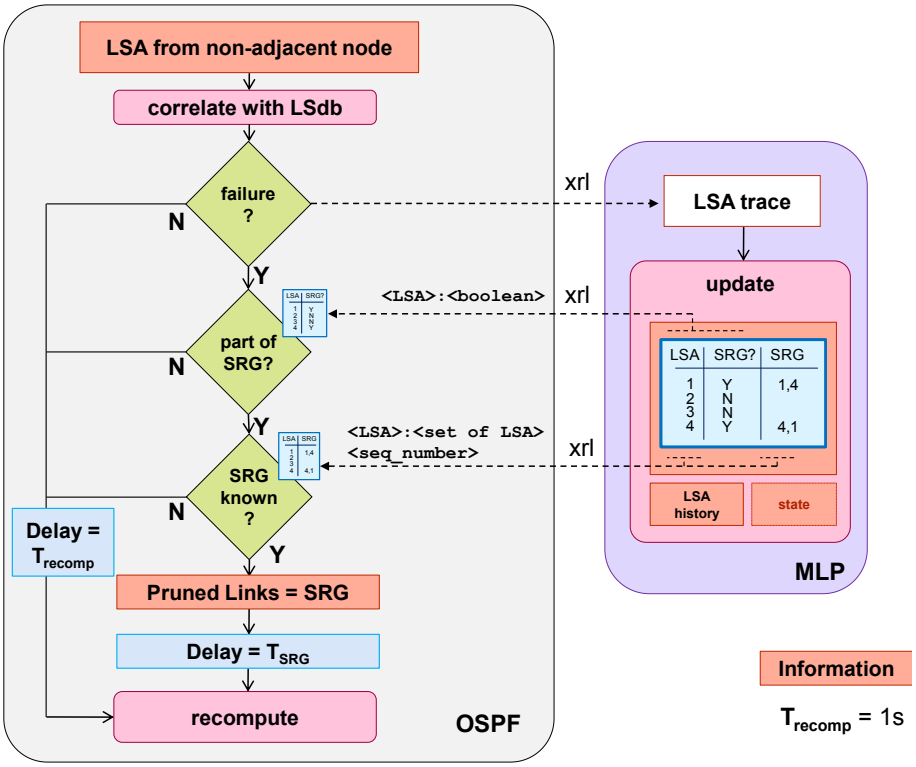

Figure 5.2: High-level flowchart for SRG inference

reported as an LSA trace to the SRG inference running in the machine learning process (MLP) through a data collection interface. When no failure is detected, SRG prediction is not in effect and normal operation is resumed, scheduling a recompute with normal hold-off time.

If however a failure is indeed detected, then the shared risk group prediction and link pruning process is continued. First, the detected failing link is checked against a list of known SRG, or rather, a list of links in known SRGs. For failures not belonging to a known SRG, we abort the prediction process and continue with normally scheduled recomputation, but note that the failure will have been reported to the SRG inference module and will therefore soon appear in the list of known SRGs. Next, we check the SRG table received from the SRG inference module to see if a suitable SRG can be identified. This may not be the case as the SRG table contains probabilities $P(SRG_i|link_j)$, indicating the (predicted) probability SRG_i will occur upon detection of $link_j$ failing. For a certain link A, none of the $P(SRG_i|A)$ may exceed a set threshold, in which case no SRG can be identified with sufficient confidence.

If an SRG can be inferred, the SRG is expanded into a set of links which become a list of pruned links. These links will be filtered out in the shortest-path tree recomputation process (which has been modified in order to do this). The recompute is scheduled with hold-off time T_{SRG}, which is shorter than the default T_{recomp}. Stability is ensured as the SRG inference mechanism reduces the number of routing recomputations.

```
Normal_LSA = "link i is up"
Failure_LSA = "link i is down"

SRG = set of Failure_LSA
SRG_set = set of SRG

SRG_table = set of Failure_LSA x set of SRG → double
```

Figure 5.3: Information model

The list of known SRGs and SRG table containing predictive probabilities is received by the OSPF module from the SRG inference module through a control interface. Note that the sending of SRGs and SRG table is shown as two separate information exchanges on the figure, though generally these can be performed together through a single XRL dispatch.

Several types of information containers are necessary for the SRG inference mechanism and its implementation in the OSFP module. Some additional state is added to OSPF using these containers. Fig. 5.3 shows these containers used in storing and exchanging data. A `Normal_LSA` is provided as reference. It is a representation of a 'link' block in a OSPF router LSA. It indicates a certain link is up (available in the topology), and contains all information to uniquely identify this link. Conversely, `Failure_LSA` is similar to `Normal_LSA`, however it indicates the identified link is in fact down (failing). It is the result of the correlation between incoming `Normal_LSA`s and `Normal_LSA`s stored in the link-state database; in fact, the `Failure_LSA`s are `Normal_LSA`s extracted from the link-state database (see Sec. 5.3 for more information). All other information containers build on this `Failure_LSA` type.

A shared risk group (`SRG`) is a set of such `Failure_LSA`s. This means that an `SRG` can easily be broken up in its constituent `Failure_LSA`s (the links of the shared risk group) for comparison with failing links or expansion into a set of links to be pruned from the topology.

`SRG_set` contains a set of shared risk groups. **SRG_set** as a state in the OSPF module contains the list of known SRGs mentioned earlier; it uses the `SRG_set` container type.

`SRG_table` is a two-dimensional table, giving a floating point value (probability) for (`Failure_LSA`, `SRG`) pairs. As a state **SRG_table**, the SRG inference table is stored as an additional state in the OSPF module. The inference table is serialized form of the information learned in the MLP. The total state inside the MLP may be quite a bit larger than the inference table itself, since it is not a 'snapshot' of current inference knowledge but is also used to continue learning with future data collection.

Figure 5.4: Outline of SRG table

Finally, two further states are added to the OSPF module. **Failing_links** is a list of links to be pruned at the next recomputation step. Since recomputation is scheduled somewhere into the future (at most T_{SRG} later when an SRG can be inferred), this list needs to be stored temporarily. Mostly this list will be empty under normal operation. It may also be updated a couple of times as each router LSA is received, before the final recomputation is performed. `Failing_links` is a set of `Failure_LSA`s, so it has the same type as `SRG`. **SRG_links** contains the list of all links inside one (or more) known `SRG`. It is the union of `SRG`s in **SRG_set** (and therefore of type `SRG`). It may be constructed ad-hoc from the list of known SRGs when needed, but is kept as a state to facilitate look-up in checking whether a `Failure_LSA` is a known SRG link.

Fig. 5.4 shows the general outline of the SRG table. It is used to determine probabilities for a certain failing link or LSA (a row in the table). The columns provide occurrence probabilities for each SRG. The sum of a row of probabilities $\sum_{j=1}^{n} P(SRG_j|L_i)$ should be ≤ 1. The final column in the figure represents the probability of a singleton SRG $\{L_i\}$ occurring, consisting of the failing link L_i itself. Whether this last column is sent from the inference module to OSPF is determined by the assumption made about the sum of probabilities. If we assume the sum of a row of probabilities is exactly 1, then $P(\{L_i\}|L_i)$ is the left-over probability $1-\sum_{j=1}^{n} P(SRG_j|L_i)$ and does not need to be included. Otherwise it will be included, and there will still be a left-over probability $1 - \sum_{j=1}^{n} P(SRG_j|L_i) - P(\{L_i\}|L_i)$. It can be used to assign weight and/or confidence in the SRG table to a row of probabilities. Note that $\{L_i\}$ is itself of type `SRG`, like SRG_j so there are no typing problems with its inclusion as an SRG in **SRG_table** or **SRG_set**.

5.3 Implementation

This section details the process and data models that were constructed, starting from the general design of the SRG inference mechanism. We concentrate on the changes to the OSPF module in terms of code. Also, we explain the XRL interfaces needed for this use case.

Fig. 5.5 briefly shows the interaction between the relevant code paths in the OSPF and MLP module. A number of methods corresponding with events triggered through XRL messages have been changed or added. The figure shows these added or changed

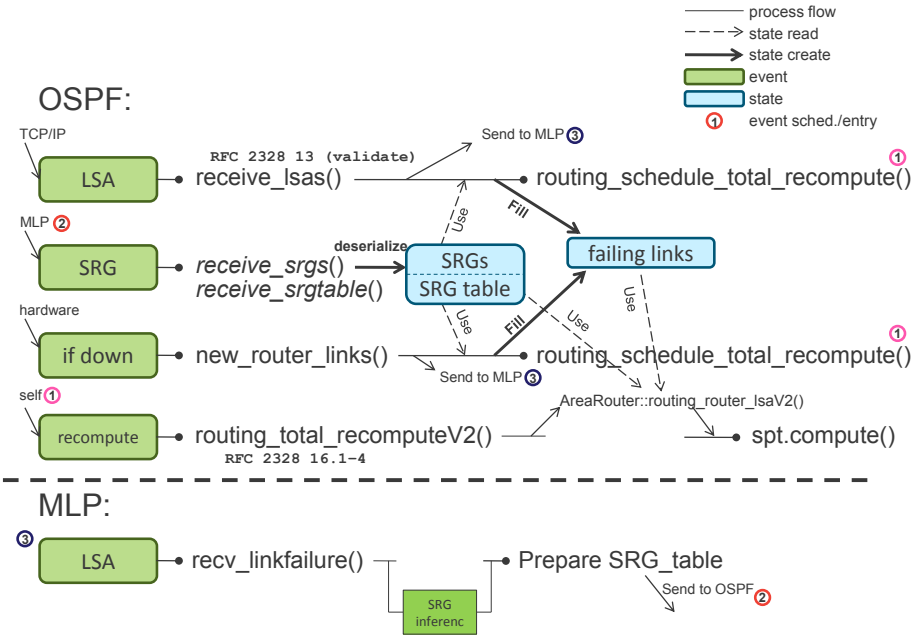

Figure 5.5: OSPF process flow

events: LSA reception event, SRG reception event, hardware interface down event and recompute event for the OSPF module. LSA trace reception event for the MLP module (running the SRG inference). The interactions with state are shown on the figure through arrows indicating either state created or read. The calling of other events is shown by small circled numbers.

On the MLP side of the process flow, the reception of link failures as LSA traces in `recv_linkfailure()` leads to a straight-forward update and learning phase governed by the implemented SRG inference algorithm, resulting in a SRG inference table that is passed back to the OSPF module causing an SRG reception event. The sending of LSA traces is initiated in the OSPF module for topology change events, i.e. external LSA reception or interface up/down hardware triggers.

On the OSPF side of the process flow, an SRG reception event is added (which requires additions to the XRL interface of OSPF, as detailed below). The SRG reception event receives SRG inference information an deserialized it into usable **SRG_set** and **SRG_table** states, which are used by the prediction and pruning functions.

The LSA reception (triggered by a OSPF protocol message from an external router) and interface down events are changed so that they now include checking for link failure (correlating with the link-state database), sending failure to the MLP, predicting using SRG inference table state, filling links to be pruned into the **Failing_links** state, and finally scheduling a `routing_total_recomputeV2()` event with the appropriate hold-off time. Note that the **SRG_links** state is omitted for clarity.

Finally, the recompute event is changed to prune links in **Failing_links** from the topology graph before execute the shortest-path tree recomputation (which then leads to filling in the routing tables).

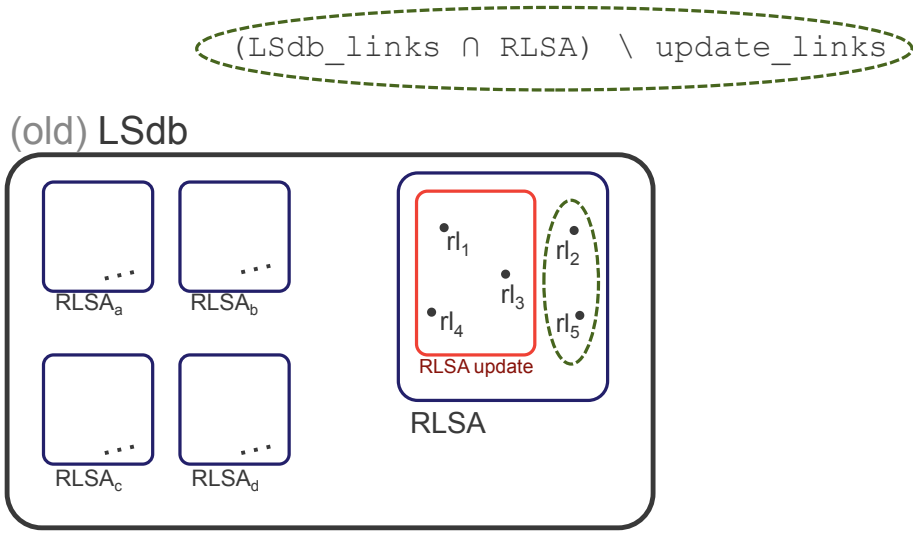

Figure 5.6: Correlating incoming LSAs with old link-state database to find failing links

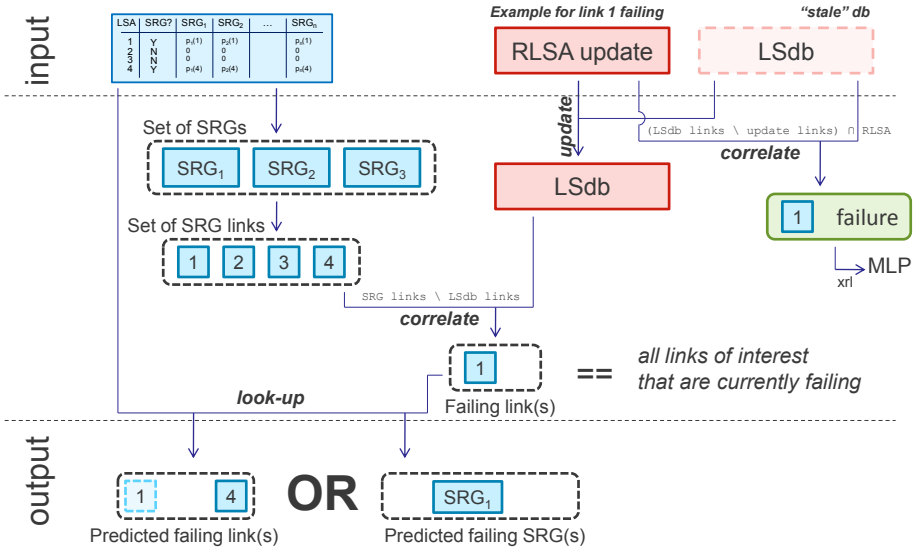

Figure 5.7: Link failure detection and failing links update

Fig. 5.6 shows how link failure are identified in the `receive_lsas()` code path. The (old) link-state database contains a set of router LSAs, each consisting of a number of router links. When a router LSA is received, we find the corresponding router LSA in the old link-state database $(LSdb \cap RLSA)$, and subtract the set of links in the updated router LSA from this set. This identifies the failing router links.

On Fig. 5.6, we see the full link failure detection and failing links update (through inference) process. We show this in terms of input to, and output from the procedure. Link failure detection on the top-right is done as in the previous figure. From the **SRG_set**, we build a set of SRG links **SRG_links** (this can be done in this code path, or at the time of reception of a SRG inference table update). With the **SRG_links** set, we can find all failing links of interest in the currently known topology (available in the update link-

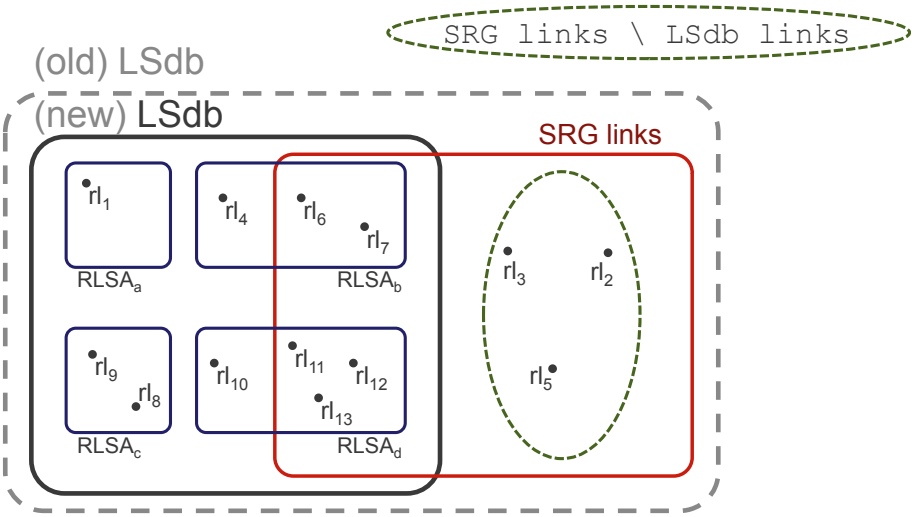

Figure 5.8: Correlating update link-state database and set of SRG links to find list of failing links of interest

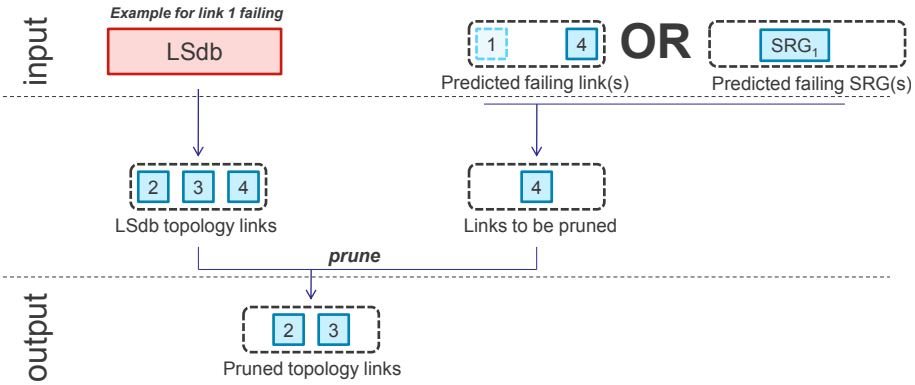

Figure 5.9: Use of set of failing links in pruning the shortest-path table

state database). Failing links of interest are links that are failing and part of **SRG_links**, since it is for these links we have predictive probabilities available in **SRG_table**. Using this set of failing links of interest, we infer predicted failing links or shared risk groups, which are then finally expanded and stored and **Failing_links**.

Fig. 5.8 shows, for an example, how to determine this set of failing links of interest.

Failing_links is used in pruning the topology before shortest-path tree recompute (Fig. 5.9), in the `routing_total_recomputeV2()` code path.

Link failures can be detected basically with the update of router LSAs (RLSA). These updates can occur in two different types. On the one hand, there are remote RLSA updates received through OSPF protocol messages (`receive_lsa()` code path as explained above—this code path is in fact implemented in a method `Area_router::receive_lsas()` of the Area_router class (which provides the area routing functionality of OSPF). On the other

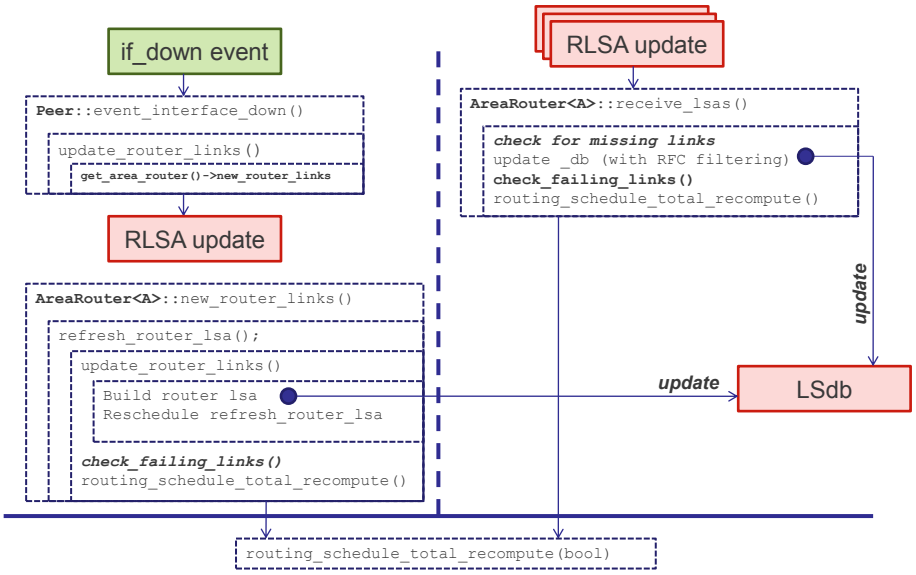

Figure 5.10: Use of set of failing links in pruning the shortest-path table

hand, these update can have a local trigger - such as an network interface up/down event (e.g. implemented in `Peer::event_interface_down()`). However, they can also be caused by periodic local RLSA refresh (`Area_router::refresh_router_lsa()`) or the discovery of new (or changed) local router links (`Area_router::new_router_links()`). The interaction of these methods is shown on Fig. 5.10. Most of the functionality for detecting link failures and inferring SRGs is done in `check_failing_links()`, which calls the functionality that was added to the OSPF module. A `bool` parameter was added to the `routing_schedule_total_recomputeV2(bool = false)` method, indicating an expedited recompute scheduling in case an SRG was inferred. T_{SRG} is used when this parameter is true, otherwise the original T_{recomp} is used.

Fig. 5.11 shows where the states mentioned earlier are implemented in the OSPF module. `Failure_LSA`s are implemented as a `RouterLink` which is used internally in the Xorp OSPF code for storing router links from router LSAs. The probabilities in **SRG_table** are stored using `floats`. **Failing_links** is used only within the Area_router class, so the data structure is defined in `area_router.hh` as `_failing_links`. The other states are defined in `ospf.hh`: **SRG_set** as `_srgs`, **SRG_links** as `_srglinks` and **SRG_table** as `_srgtable`.

The `RouterLink` corresponds to the structure of router links and encapsulating router LSA as defined by the OSPF protocol (Fig. 5.12). To identify a `RouterLink` or in fact a `Failure_LSA`, we need the following information: **LS age**, **Advertising Router**, **LS sequence number**, **Link ID**, **Link Data**, **Type**. These six pieces of data are serialized when exchanging a `Failure_LSA` in-between the OSPF and MLP module.

To summarize the process flow, an example is given on Fig. 5.13 for the exchange between OSPF (left) and MLP (right) upon the reception of an LSA and consequent detection of a link failure. As the OSPF module has only one event loop, all code

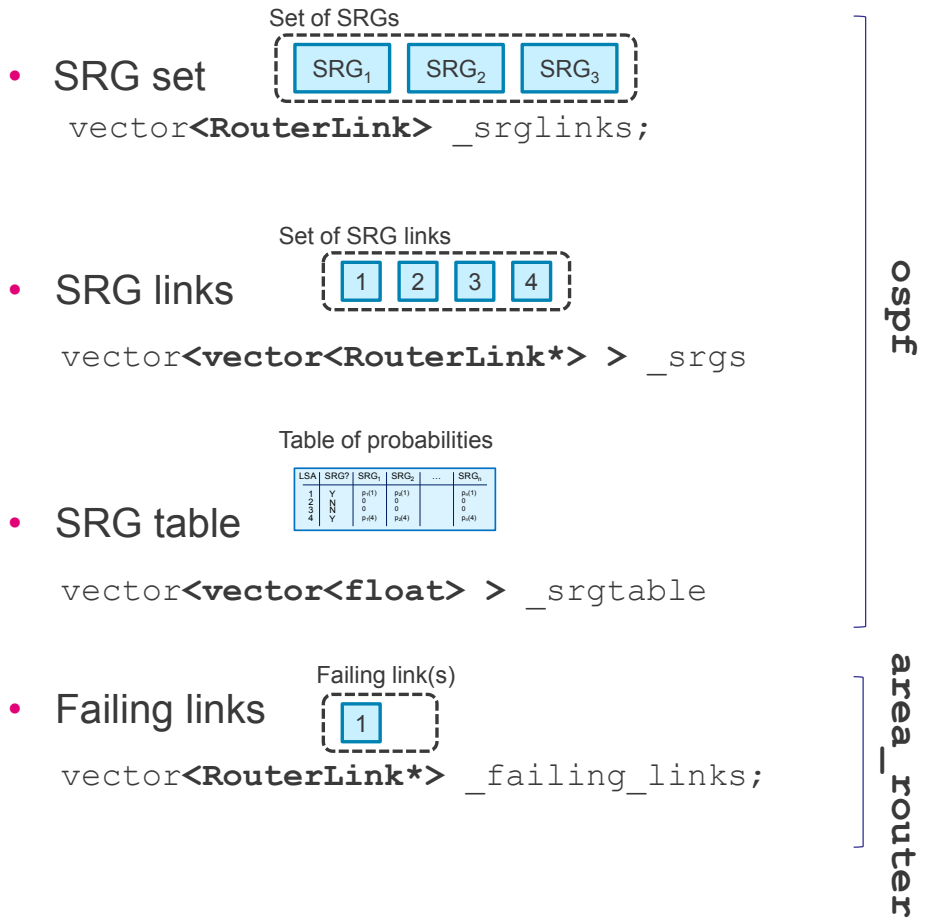

Figure 5.11: Implementation of state in OSPF module

paths must execute sequentially. Most importantly, this means that the response of the MLP with update SRG inference information will typically be processed only after the `receive_lsas()` code path has finished, and often will even not be in time for the schedule reroute. Therefore, we envisioned the interaction such that the OSPF module can do inference by itself from its existing `SRG_table` state, without having to wait for an SRG inference answer from MLP, which would defeat the purpose of the SRG inference mechanism, namely faster recovery times (for multiple link failures).

Still, we show the three relevant code paths for OSPF next to eachother on the figure, but keep in mind that at all times, only one of them can be active—the MLP module of course runs in a different process with its own eventloop, so it can perform calculations and event handling parallel to the OSPF module.

In the example in the figure, the SRG inference table updates are sent during the rerouting process, which causes the update of the OSPF `SRG_table` state to be delayed until this recompute is finished.

De-coupling the data collection (link failure reporting) and control (SRG inference table update) was done with the implementation over XRL in mind. There are two versions of the SRG inference mechanism implementation and XRL interface defini-

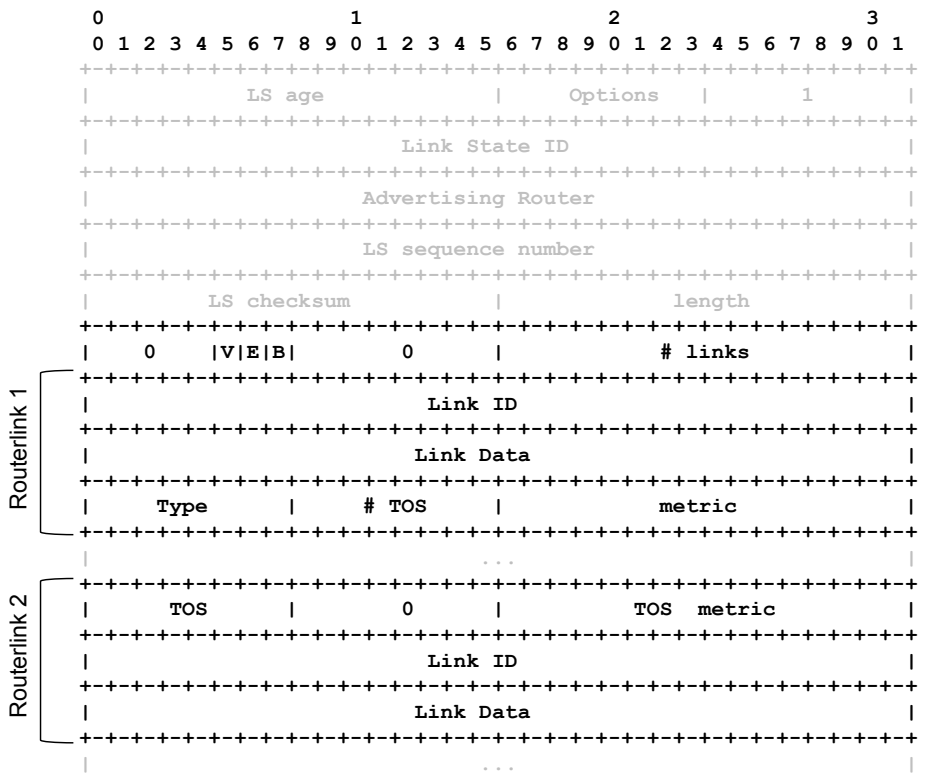

Figure 5.12: Structure of router LSA and contained router links

tions. A first implementation uses Xorp 1.6 and was used as proof-of-concept during the 2010 ECODE review, and was also used for public demonstration of the SRG inference technique[ea10, ea11]. We concentrate on the second implementation which uses the EUA TCI and push mechanisms (based on Xorp 1.8).

The interface `linkfail` (Fig. 5.14) is used to receive LSA traces containing failing links and must be implemented by the MLP. As can be seen, `link_failure` has as parameters the **LS age**, **Advertising Router**, **LS sequence number**, **Link ID**, **Link Data**, **Type** values that identify a failing link. These are passed as 32-bit numbers.

In the Xorp 1.6 case, the MLP registers with the router manager and advertises this linkfail target in order to receive LSA traces from OSPF. This causes a problem within Xorp 1.6 if the MLP goes offline and reconnects to the router manager. For performance reasons, XRL look-up is cached for processes, an XRL target reconnecting with a different IP or port number causes an inconsistency in this cache, leading to an assertion failure and finally a crash in the calling module (in this case OSPF). Modules implementing targets should be brought up by the router manager only and not manually.

For the EUA-based implementation, target within the MLP are not allowed (they cannot be advertised to the TCI), but instead a push mechanism is available, which allows a module to push data to the MLP using a regular (non-target) call-back. In the EUA case, the call-back and push will use the `link_failure` signature to send LSA traces from OSPF to MLP. This data collection is mediated through a monitoring point (MP) which in fact does implement the `linkfail` target. The MLP registers with the

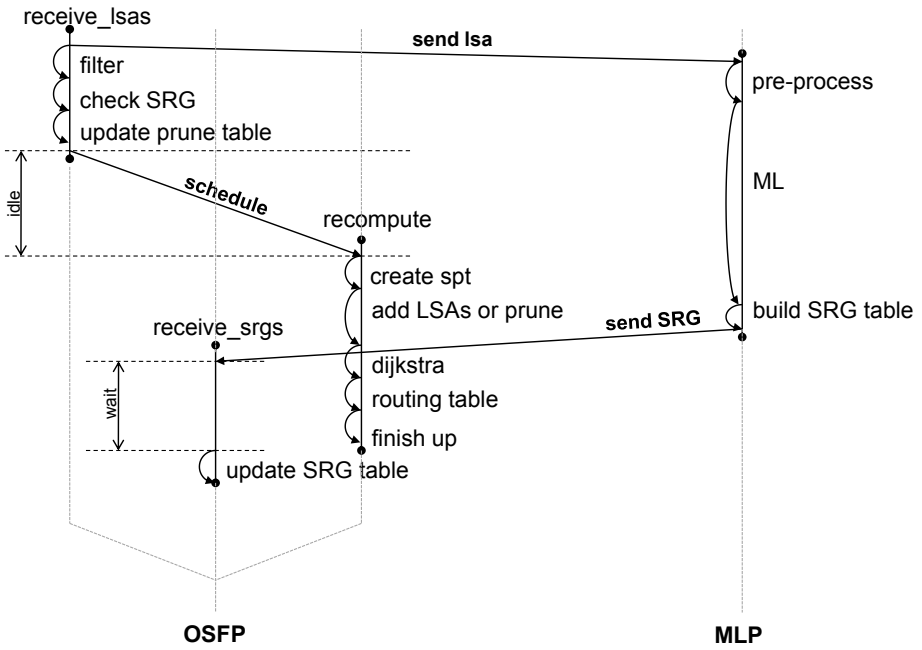

Figure 5.13: Example of OSPF (left) and MLP (right) interaction in time

```
interface linkfail/1.0 {
    /**
     * Report failing link
     *
     * @param failing routerlink data
     */
     link_failure?ls_age:u32&ls_seq:i32&ar:u32& \
         rl_type:u32&rl_link_id:u32&rl_link_data:u32;
}
```

Figure 5.14: XRL interface for receiving link failure reports

MP (a module outside of OSPF), while OSPF sends the LSA traces to the MP, which forwards the traces to the MLP.

This MP also implements the regular control of the OSPF function, in that its target will accept SRG inference table updates. Its interface is shown on Fig. 5.15.

`failure_push` and `failure_cancel` are used to (de)register interest with the MP in receiving LSA trace updates (provided to the MP, regardless of interest registration, from OSPF through the MP's `linkfail` interface). `set_srgs` and `set_srgtable` are used to pass a set of SRGs or a full SRG table respectively.

`set_srgs` is used when loading a simple set of SRGs into OSPF's **SRG_set**, where prediction is then done using simple matching and no probability. This is only done when the SRG inference algorithm in the MLP only uses clustering and does not calculate probabilities. For `set_srgtable`, **SRG_table** but also **SRG_set** are updated from the parameters passed (either `set_srgs` or `set_srgtable` should be called

```
interface eua_srg_mp/0.1 {
    enable_eua_srg_mp ? enable:bool;
    start_eua_srg_mp;

   /**
    * SRG table upload functionality
    */
    set_srgs ? srgs:list<binary>
    set_srgtable ? srgs:list<binary> & links:list<binary> & \
        table:binary;

   /**
    * Linkfail push functionality
    */
    failure_push ? cbxrl:txt & prefix:txt;
    failure_cancel ? cbxrl:txt & prefix:txt;
}
```

Figure 5.15: XRL interface for the SRG monitoring point

for an SRG inference information update, not both).

The SRGs are presented a list of binary data. Each binary atom contains the information of a single SRG. The atom itself is constructed from the concatenation of binary data of links.

Links are presented a binary atom, which is the concatenation of the six 32-bit identifying numbers (in `linkfail` order). `set_srgtable` has a list of links as parameters, identifying each of the rows of the SRG table.

The actual table data itself, i.e., the probability values are presented as binary atom as well. Normally this is just the concatenation of n `floats` (in network byte order), where $n = |srgs| \times |links|$. However, the binary format for the `table` parameter is not fixed, for example, a format may be decided to represent partial updates, or compressed data.

Note that in the earlier Xorp 1.6 implementations, loosely typed lists were allowed (e.g., `srgs:list` instead of `srgs:list<binary>`. The earlier implementation uses nested lists of `u32` (32-bit number) to represent the SRGs, where a link is a list of numbers, and SRGs are a list of link (a list of a list of numbers). Xorp 1.8 does not support nested list (i.e., `srgs:list<list<u32> >`, etc.) in interface definitions.

The SRG inference information is passed on to the OSPF module, which has some changes to its interface to accept these uploads (Fig 5.16), matching the `set_`$\star$ methods in the MP interface.

Fig. 5.17 provides a comparison between the first Xorp 1.6 implementation (on the left), and the newer implementation using the EUA platform (on the right) for a three-node network. In the EUA case, the TCI and SRG MP are added. Initially, this feature seems to have little impact on the SRG inference, however, as Fig. 5.18 shows, it allows

```
interface ospfv2/0.1 {
    ...

    /**
     * Receive SRG updates (ECODE)
     */
    recv_srgs ? srgs:list<binary>;
    recv_srgtable ? srgs:list<binary> & links:list<binary> & \
        table:binary;

    ...
}
```

Figure 5.16: Changes to the OSPF interface (partial)

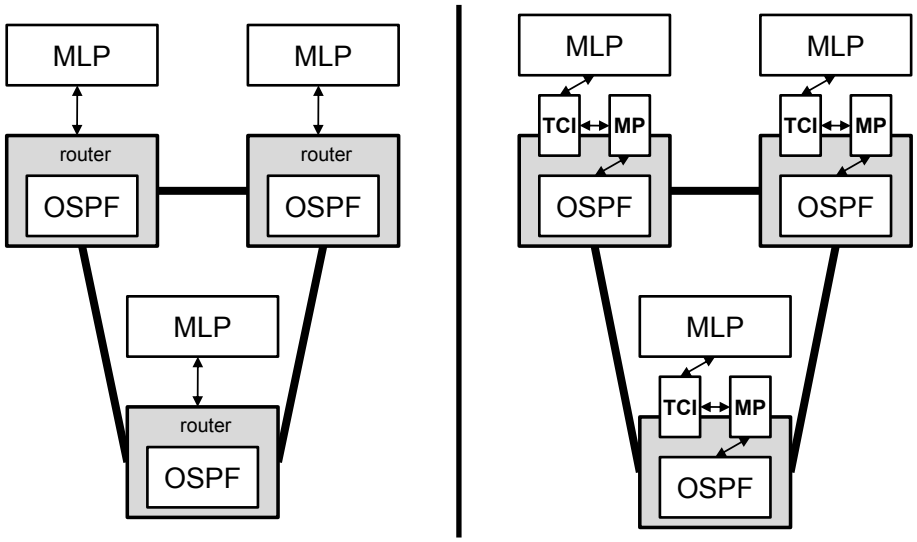

Figure 5.17: High-level comparison of Xorp 1.6 (left) and EUA/Xorp 1.8 (right) implementation

for some interesting distributed/centralized SRG inference scenarios.

On the left of the figure, we see the scenario where a single MLP has control over all of the network. This means it receives link failure LSA traces from all nodes, and updates the SRG inference tables for all OSPF instances (presumably with the same table data). This centralized scenario is not possible with the node local SRG inference implementation on Xorp 1.6.

Similarly, we can keep a SRG inference MLP for each node (right part of the figure), but use the TCI functionality to allow the MLPs of the nodes to communicate, e.g., exchanging inference model parameters or learned values.

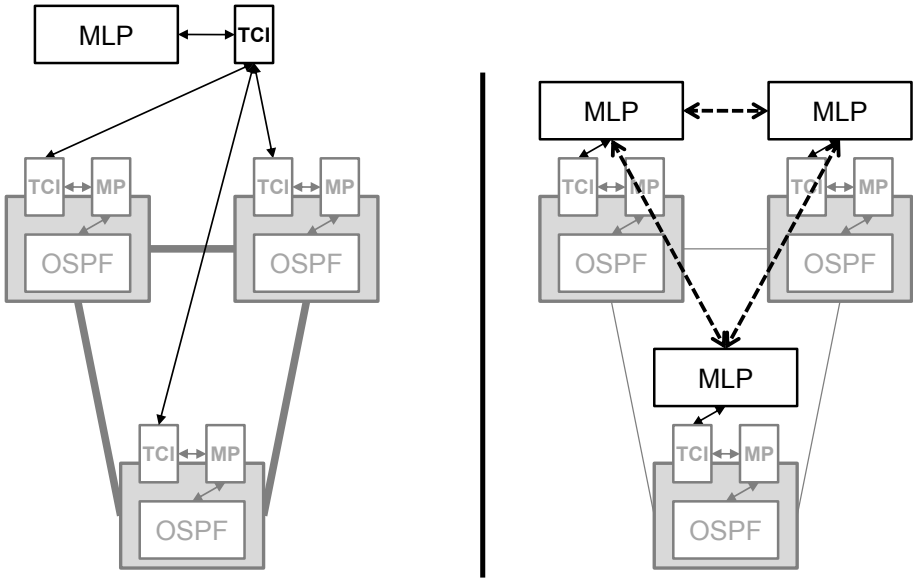

Figure 5.18: Distributed/centralized SRG inference scenarios

5.4 Conclusion

We have presented an OSPF SRG (shared risk groups) inference mechanism to improve the OSPF protocol recovery process in case of links failure. Our solution correlates link-state advertisements received earlier from OSPF routers. We modified OSPF link state packets to carry failure information. The failures discovered by OSPF are reported in the EUA to the inference module with XRLs. Failure information is used to infer SRG. The infered SRGs are then sent back to the OSPF module. Upon re-routing event, OSPF uses the SRGs to determine to use the route presenting less failure risk.

Chapter 6

Conclusion

In this deliverable, we presented how to design and implement in the ECODE Unified Architecture (EUA) representative use cases of the project.

On the one hand, adaptive sampling and anomaly detection are pure traffic monitoring engines that use learning engines to improve their efficiency. IDIPS, meanwhile, acts as an aggregator of monitoring information for tools requiring to select the best paths, for any arbitrary definition of best. To this aim, IDIPS provides a uniformed architecture to use monitoring engines like an anomaly detection system or an adaptive traffic sampling system. IDIPS also provides a simple interface for clients to obtain informed path ranking based on these monitoring information. For compatibility reasons with the existing flow monitors, our adaptive traffic sampling mechanism is not implemented directly in XORP. However, to be integrated in the EUA, the adaptive traffic sampling mechanism is interfaced with a wrapper that is able to translate EUA requests into adaptive traffic sampling implementation primitives and vice versa. Finally, the network recovery & resiliency has been completely integrated into the EUA. For the success of this integration the XORP OSPF implementation has been adapted be integrated directly in the EUA as well.

Bibliography

[AAS03] A. Akella, Shaikh A., and R. Sitaraman. A measurement-based analysis of multihoming. In Proc. ACM SIGCOMM, August 2003.

[ACP09] G. Androulidakis, V. Chatzigiannakis, and S. Papavassiliou. Network Anomaly Detection and Classification via Opportunistic Sampling. IEEE Network, 23(1):6–12, January 2009.

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In Proc. of the ACM SIGMOD International Conference on Management of Data, 1998.

[AKZ99] G. Almes, S. Kalidindi, and M. Zekauskas. A Round-trip Delay Metric for IPPM. RFC 2681 (Proposed Standard), September 1999.

[CCRK04] M. Costa, M. Castro, R. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In Proc. 24th International Conference on Distributed Computing Systems, March 2004.

[CIB+06] Gion Reto Cantieni, Gianluca Iannaccone, Chadi Barakat, Christophe Diot, and Patrick Thiran. Reformulating the monitor placement problem: Optimal networkwide sampling. In Proc. of CoNeXT, 2006.

[Cis00] Cisco. Netflow services and applications. White Paper, 2000.

[Cla04] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[CM05] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. IEEE Transactions on Networking, 13(6):1219–1232, December 2005.

[DCKM04] F. Dabek, R. Cox, K. Kaashoek, and R. Morris. Vivaldi, a decentralized network coordinated system. In Proc. ACM SIGCOMM, August 2004.

[DHKS09] Xenofontas Dimitropoulos, Paul Hurley, Andreas Kind, and Marc Stoecklin. On the 95-percentile billing method. In Passive and Active Measurements Conference (PAM), April 2009.

[DHS01] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern Classification - second edition. Wiley Publisher, 2001.

[dLUB05] C. de Launois, S. Uhlig, and O. Bonaventure. Scalable route selection for IPv6 multihomed sites. In Proc. IFIP Networking, May 2005.

[Dra03] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484 (Proposed Standard), February 2003.

[ea10] B. Puype et al. SRLG inference in OSPF for improved reconvergence after failures. In Proceedings of joint ServiceWave 2010/FIA Ghent, December 2010.

[ea11] B. Puype et al. OSPF failure reconvergence through SRG inference and prediction of link state advertisements. In Proceedings of ACM SIGCOMM'11, pages 468–469, August 2011.

[EKSX96] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In Proc. of 2nd International Conference on Knowledge Discovery and Data Mining (KDD 96), 1996.

[EV02] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In Proc. of ACM SIGCOMM, 2002.

[FJ05] A. Fred and A.K. Jain. Combining Multiple Clusterings Using Evidence Accumulation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 27(6):835–850, June 2005.

[FJP+99] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gruniewicz, and Y. Jin. An architecture for a global Internet host distance estimator service. In Proc. IEEE INFOCOM, March 1999.

[FO09] G. Fernandes and P. Owezarski. Automated Classification of Network Traffic Anomalies. In Proc. of 5th International ICST Conference on Security and Privacy in Communication Networks, 2009.

[FR98] C. Fraley and A. E. Raftery. How Many Clusters? Which Clustering Method? Answers Via Model-Based Cluster Analysis. The Computer Journal, 41(8):578–588, 1998.

[GDZ06] R. Gao, C. Dovrolis, and E. Zegura. Avoiding oscillations due to intelligent route control systems. In Proc. IEEE INFOCOM, April 2006.

[HHK03] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: an open platform for network research. SIGCOMM Comput. Commun. Rev., 33:53–57, January 2003.

[HV03] N. Hohn and D. Veitch. Inverting sampled traffic. In Proc. of IMC, 2003.

[Jai10] A. K. Jain. Data Clustering: 50 Years Beyond K-Means. Pattern Recognition Letters, 31(8):651–666, 2010.

[KME05] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of internet traffic. In Proc. of SIGMETRICS, 2005.

[LCD04] A. Lakhina, M. Crovella, and C. Diot. Characterization of Network-Wide Anomalies in Traffic Flows. In Proc. of 2nd ACM Internet Measurement Conference, 2004.

[LGP+05] E. K. Lua, T. Griffin, M. Pias, H. Zheng, and J. Crowcroft. On the accuracy of embeddings for Internet coordinate systems. In Proc. USENIX Internet Measurement Conference (IMC), October 2005.

[LGS07] J. Ledlie, P. Gardner, and M. I. Seltzer. Network coordinates in the wild. In Proc. USENIX Symposium on Networked System Design and Implementation (NSDI), April 2007.

[LHC03] H. Lim, J. C. Hou, and C.-H. Choi. Constructing internet coordinate system based on delay measurement. In Proc. ACM SIGCOMM Internet Measurement Conference (IMC), October 2003.

[LHC05] H. Lim, J. C. Hou, and C-H. Choi. Constructing Internet coordinate system based on delay measurement. IEEE/ACM Transactions on Networking, 13(3):513–525, June 2005.

[Lib] Libpcap. Libpcap: a Portable C/C++ Library for Network Traffic Capture. `http://www.tcpdump.org`.

[LPS06] J. Ledlie, P. Pietzuch, and M. I. Seltzer. Stable and accurate network coordinates. In Proc. International Conference on Distributed Computing Systems, July 2006.

[MCO11] J. Mazel, P. Casas, and P. Owezarski. Sub-Space Clustering & Evidence Accumulation for Unsupervised Network Anomaly Detection. In Proc. of 3rd COST-TMA International Workshop on Traffic Monitoring and Analysis, April 2011.

[Moy98] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. Updated by RFC 5709.

[MS04] Y. Mao and L. Saul. Modeling distances in large-scale networks by matrix factorization. In Proc. ACM SIGCOMM Internet Measurement Conference (IMC), October 2004.

[NB09] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[NZ02] T. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In Proc. IEEE INFOCOM, June 2002.

[NZ04] T. S. E. Ng and H. Zhang. A network positioning system for the Internet. In Proc. USENIX Annual Technical Conference, June 2004.

[Pap07] V. Pappas. Coordinate-based routing for overlay networks. In Proc. International Conference on Computer Communications and Networks (ICCCN), August 2007.

[PCW+03] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS), February 2003.

[PHL04] L. Parsons, E. Haque, and H. Liu. Subspace Clustering for High Dimensional Data: a Review. ACM SIGKDD Explorations Newsletter - Special Issue on Learning from Imbalanced Datasets, 6(1), June 2004.

[PLMS06] P. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. Seltzer. Network-aware overlays with network coordinates. In Proc. IEEE International Conference on Distributed Computed Systems Workshops (ICDCSW), July 2006.

[PPZ+08] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. A measurement study of internet delay asymmetry. In Proceedings of the 9th international conference on Passive and active network measurement, PAM'08, pages 182–191, Berlin, Heidelberg, 2008. Springer-Verlag.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[RMK+08] V. Ramasubramanian, D. Malhki, F. Kuhn, I. Abraham, M. Balakrishnan, A. Gupta, and A. Akella. A unified network coordinate system for bandwidth and latency. Technical Report MSR-TR-2008-124, Microsoft Research, September 2008.

[ST03] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In Proc. IEEE INFOCOM, March 2003.

[VS10] Hui Zhang Vyas Sekar, Michael K Reiter. Revisiting the case for a minimalist approach for network flow monitoring. In Proc. of IMC, 2010.

[WSS05] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: a lightweight network location service without virtual coordinates. In Proc. ACM SIGCOMM, August 2005.

[WZA06] N. Williams, S. Zander, and G. Armitage. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. ACM SIGCOMM Computer Communication Review, 36(5), October 2006.

[YRCR04] Ming Yang, X. Rong, Li Huimin Chen, and Nageswara S. V. Rao. Predicting internet end-to-end delay: an overview. In in Proc. of 36th IEEE Southeastern Symposium on Systems Theory, pages 210–214, 2004.