mPlane

an Intelligent Measurement Plane for Future Network and Application Management

ICT FP7-318627

Database Layer Design

Author(s):	FTW	Arian Bär (Ed.), Pedro Casas, Alessandro D'Alconzo
	POLITO	Alessandro Finamore, Marco Mellia
	EURECOM	Antonio Barbuzzi
	SSB	Gianni De Rosa
	ENST	Dario Rossi, Jordan Augé, Marc-Oliver Buob
	NETvisor	Tivadar Szemethy
	TID	Ilias Leontiadis
	NEC	Maurizio Dusi

Document Number:	D3.2
Revision:	1.0
Revision Date:	08 Nov 2013
Deliverable Type:	RTD
Due Date of Delivery:	15 Nov 2013
Actual Date of Delivery:	08 Nov 2013
Nature of the Deliverable:	(R)eport
Dissemination Level:	Public

Abstract:

This document describes the design of the database layer used in the mPlane project. Starting from the per use case algorithms defined in D3.1 we infer the types and format of data stored in mPlane repositories. In addition, we give a description of external data sources, which might either be mirrored inside an mPlane repository or accessed via the provided API. By the term mPlane repository we understand a logical or physical instance, providing data access, storage or both to other mPlane components. We continue by giving a description of data indexing and in-repository data processing. The in-repository data processing focuses on handling data streams of very high volume, continuously arriving at the repository. The document is completed, by a detailed description of the integration of repositories into the general mPlane architecture by giving an overview of the exposed capabilities. This part focuses on how mPlane repositories are used by the supervisors and reasoners from WP4.

Keywords: databases, repositories, storage, big data

Disclaimer

Contents

Document change record

Version	Date	Author(s)	Description
0.1	21 Aug 2013	Arian Bär (FTW) ed.	Initial draft
0.2	01 Oct 2013	Arian Bär (FTW) ed.	Section 3.2 completed
0.3	08 Oct 2013	Antonio Barbuzzi (EUR)	Sections 2.3.2 and 2.3.3 completed
0.4	09 Oct 2013	Gianni Da Rosa (SSB)	Section 4.3 completed
0.5	15 Oct 2013	Alessandro Finamore (Polito)	Sections 2.3.1, 3.1, and 2.2 completed
0.6	25 Oct 2013	Alessandro D'Alconzo, Pedro Casas (FTW)	final editing
0.7	29 Oct 2013	Arian Bär (FTW) ed.	Final draft for review
0.8	07 Nov 2013	Arian Bär (FTW) ed.	Including changes from review
1.0	08 Nov 2013	Arian Bär (FTW) ed.	Final version

1 Introduction

This document describes the main functionalities offered by mPlane repositories. In particular, the document focuses on how the stored and pre-processed data is exposed and made accessible to other mPlane components and how external data sources can be accessed.

In the mPlane architecture, the repositories form a storage and large-scale data analysis layer (referred to as repository layer) between the monitoring probes, which collect data, and the supervisors, which access the pre-processed data for further analysis. The repository layer fulfills two main tasks: (i) data storage and access, and (ii) large-scale data processing. In the first case, an mPlane repository is capable of storing the data collected by the measurement layer as well as exposing the data it contains to other mPlane components, including external mirrored data sources. In the second case, the repository is also capable of pre-processing the measurements collected by the mPlane probes, relying both on large-scale data processing frameworks and other analysis approaches, as described later in this document. We use the term pre-processing to indicate that in the general case, further analysis is performed on the supervision layer, using the analysis modules and the intelligent reasoner.

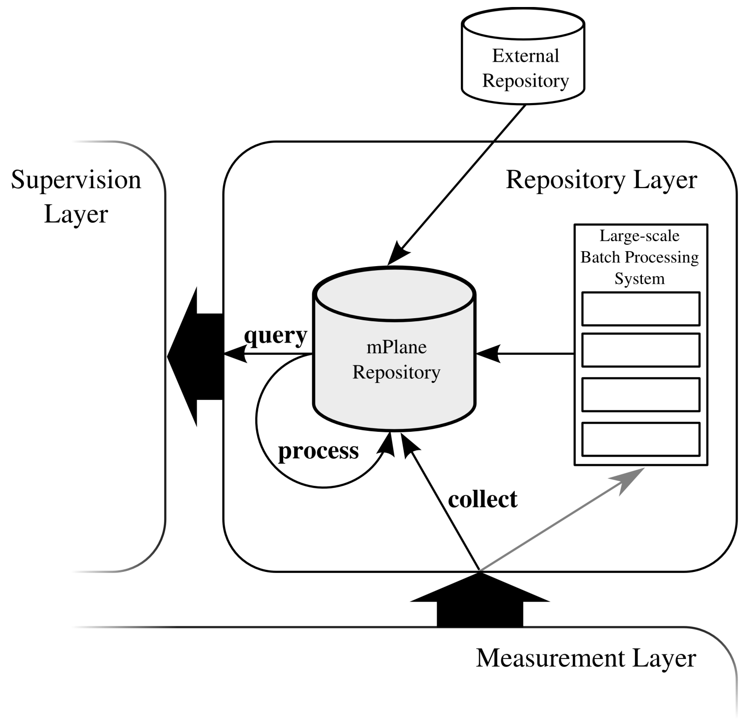


Figure 1: Overview of the interactions of mPlane repositories with the measurement layer, large-scale batch processing systems working inside the repositories layer itself, the supervision layer, and the external repositories.

In general terms, the concept of an mPlane repository is generic, and it can be materialized by different kinds of technologies, such as: SQL databases (e.g., DBStream), NoSQL databases (e.g., MongoDB), in-memory databases (e.g. VoltDB), key-value data stores (e.g., Redis), large-scale data analysis frameworks (e.g. Hadoop), etc.. In fact, the mPlane architecture does not impose any specific type of technology to be used, and the final prototype implementation foresees a variety of solutions. What it is mandatory is that such repositories have the ability to store and expose the data to other mPlane components.

The most well known and widely used approach for storing data is through the usage of Relational Database Management Systems (RDMS), which are traditionally used when it comes to the storage and efficient retrieval of data. Therefore, while this document presents a design which is generic enough to support many different types of repositories, it elaborates on some particular RDMS-based systems (namely DBStream) and large-scale data analysis frameworks (namely Hadoop) as a unified mPlane repository providing data storage, data processing, and data accessing capabilities. Having said that, the aim of this document is to additionally describe the tasks which are common among different data storage systems, providing hints on how an individual system can become an mPlane repository.

This document is structured as follows: the part focuses on the storage of the data coming from other mPlane components, additionally providing an overview of the data which will be imported to and stored in the repositories. The second part gives an overview of the pre-processing mechanisms available in the repositories. The last part shows how the data is stored in the repositories and how it is made accessible to other mPlane components.

Figure 1 presents a general overview of the different components of the repository layer, including their interactions with other mPlane layers. Large-scale batch processing systems are shown as a separated element from repositories since they do not make data directly available to other mPlane components; they might either be used together with systems like HBase or Pig on top, or data might be imported into a RDBM System which is part of the mPlane. Section 2.3.3 further explains and develops this concept.

2 Data Import

This chapter gives a general overview how data arrives at repositories in mPlane. The data imported into repositories might later be accessed by mPlane supervisors and reasoners from WP4. Section2.1 gives an overview of the different types of data imported and processing in repositories. The following Section 2.2 introduces potential external data sources which might be used from inside the mPlane. The last Section 2.3 details mechanisms which can be used to ship data from mPlane probes to repositories.

2.1 Data from mPlane Use Cases

This section gives an overview of data from mPlane probes which is imported and processed in mPlane repositories. The section is structured by the use cases importing the data, as defined in D1.1. A description of the measurement systems providing these data, as well as a deeper explanation of the measures and data sets currently available in mPlane is provided in D5.1.

A mapping from the original type names of the data sources to the mPlane type registry is given in this section. This mapping to mPlane types only conserves the current state of type names. Since mPlane is an active, living project, changes to mPlane type names are likely happen in the future.

2.1.1 Anomaly Detection

The data used in this use case represent metrics exported by Tstat probes. Among them are metrics describing TCP connection information, traffic from popular HTTP services (e.g., YouTube, Facebook, etc.) served by major CDNs (e.g., Google CDN, Akamai, etc.) and more detailed information about flows serving videos.

Two different log types from Tstat probes are imported into the mPlane repository (e.g. DBStream) for this use case:

log_tcp reports metrics for each individual TCP connection. These include global metrics (e.g., total number of bytes/packets sent/received, start/end time, etc.), L4 metrics (e.g., number of SYN/ACK/FIN packets, congestion window, min/max/avg/std round trip time, etc.), and L7 metrics (e.g., labels related to the application related to the traffic, analysis of the SSL certificates, etc.).

log_streaming reports metrics related to HTTP video streaming services. To simplify post-processing, some of the metrics of log_tcp are replicated: e.g., bytes exchanged, traffic classification labels, etc. In addition many metrics related to video streaming service and the downloaded videos (e.g., videoid, duration, size, etc.) are present in log_streaming.

The data used by the anomaly detection use case will be a combination of data from log_tcp and log_video. From each of the log formats only certain metrics are used. Those metrics are reported in detail in the following tables.

The metrics from log_tcp which are especially useful for the anomaly detection use case are the following:

mPlane type	Tstat name	Description
client.ip4, client.ip6	Anonomized Client IP address	
server.ip4, server.ip6	Server IP address	
server.port	Server port	
start.ms	TCP flow start time	
end.ms	TCP flow end time	
rtt.avg.ms	Average RTT	Average RTT computed measuring the time elapsed between the data segment and the corresponding ACK.
rtt.min.ms	Minimum RTT	Minimum RTT observed during connection lifetime.
rtt.max.ms	Maximum RTT	Maximum RTT observed during connection lifetime.
rtt.stddev.ms	Standard derivation of the RTT	Standard deviation of the RTT.
rtt.sample.count	RTT sample count	Number of valid RTT observations.
bandwidth.partial.kbps	data bytes/duration	TCP flow download/upload throughput (avg)
octets.tcp	data bytes	Number of bytes transmitted in the payload, including retransmissions.
octets.duplicate	rexmit bytes	Number of retransmitted bytes.
packets.tcp	data pkts	Number of segments with payload.
packets.duplicate	rexmit pkts	Number of retransmitted segments.
packets.outoforder	out seq pkts	Number of segments observed out of sequence.
host.name	Full qualified domain name	The full qualified domain name returned by the DNS server.

For video flows also those additional fields should be present:

mPlane type	Tstat name	Description
rate.video.kbps	Video total datarate	Total data rate as indicated in payload.
duration.video.s	Video duration	Video duration as indicated in the payload.
width.video.px	Video width	Video width as indicated in the payload
height.video.px	Video height	Video height as indicated in the payload

In Repository Pre-process

As a first step, the plain Tstat logs of the above mentioned formats are imported into the mPlane repository DBStream. There, the continuous processing engine is used to pre-process the imported logs and generate the data format described above.

Data volumes

For passive probes data volumes are clearly related to the number of monitored users. Based on the description reported in the deliverable D5.1 [16], Tstat probes are deployed in three different scenarios:

Fastweb probes monitor the activity of about 50,000 residential users accessing to the Internet either using ADLS or Fiber-To-The-Home (FTTH) technologies. Depending of the amount of traffic monitored by each probes, the amount of compressed output data vary from about 2.25 GB/day (i.e., 70 GB/month) up to 20 GB/day (i.e. 620 GB/month).

Politecnico di Torino scenario corresponds to a typical academic network. More than 25,000 students, administrative, professors and researchers access the Internet via wired Ethernet LAN, or IEEE 802.11a/b/g WiFi access points. The amount of data captured is more variable with respect to the Fastweb scenario, but it can top 3 GB/day considering a typical day of the week, i.e., it is in the same order of the ``smaller'' Fastweb probe.

NetVISOR scenario corresponds to a networking lab in which about 50 researchers operate. The network is also used for experimentation with several test beds. Given the small scenario, the amount of data capture corresponds to only about 100 MB/day (i.e., 3 GB/month).

2.1.2 QoE Troubleshooting

The data used in this use case are metrics exported by FireLog [11] probes, representing several aspects of a web session.

mPlane type	**FireLog metrics**	**Comments**
start.ms	session_start	Start time of a given web session.
end.ms	load_ts	The page load event time is used as the end time of a session.
duration.ms	full_load_time	
source.ip	localaddress	
source.port	localport	
destination.ip	remoteaddress	
destination.port	remoteport	
client.public.ip	client_ip_by_server	FireLog servers also record the client's public IP as seen from the outside.
octets.http.request	http_request_bytes,	Number of requested bytes.
octets.http.header	http_header_bytes,	Number of header bytes.
octets.http.body	http_body_bytes,	Number of body bytes.
octets.http.cache	http_cache_bytes	Number of cached bytes.
rtt.ms	tcp_cnxting	TCP level RTT.
rtt.app.ms	app_rtt	Application level RTT.
rtt.ping.ms	ping_rtt	Ping RTT, actively measured, used as a reference.
uri	uri	Each individual object URI.
uri.session	session_url	URL of a given web page.
load.cpu.avg	CPU	CPU average utilization per session.

Data Volume

The amount of data depends highly on the number of users using FireLog and will be measured as part of the implementation at a later point in time. In addition, different websites served by the user might result into different data sizes.

2.1.3 Support Daas Troubleshooting

The following features are defined in a time-window (length-configurable) basis, for each flow:

mPlane type	**Semantic**	**Description**
start.ms	TCP flow start time	
end.ms	TCP flow end time	
packets.ip	sum of ip packets	
octets.ip	sum of ip bytes	
packets.tcp	sum of tcp packets with payload	packets.tcp does include TCP w/o payload, so would it be possible to have a counter tracking packets w/o payload.
octets.tcp	sum of tcp bytes	
x-nec-daas.histogram	TCP payload length distribution, histogram of TCP lengths	A 1441-length array with the counters of #TCP segments with a given size.
rtt.ms	per packet RTT	

In Repository Pre-process

Within the repository operations can be run on a stream-processing platform to extract statistics for each flow over a time window (which can differ from the one we send data to the repository), such as:

- Throughput per flow
- Average RTT per flow
- Mean and standard deviation of the (TCP-) packet sizes

The pre-processed features might be sent to a reasoner running statistical classifier at a later point in time.

Data Volume

Probes send such data every second, so for each flow there will be around 1500 elements * 4 bytes each = 6KB/s.

2.1.4 Media Curation

The exported data reflects the HTTP content requests made by the end customers of a certain ISP. The data format has the following general structure:

`<Timestamp, URL, referrer, user agent, anonymized user id, probe identifier>`

The used types are described in more detail and mapped to existing as well as newly added mPlane types in the following table:

mPlane type	**Semantic**	**Description**
time.ms	Timestamp	Time when the HTTP GET request is observed.
url	URL	Name + GET (e.g. name: www.domain.com and GET: /content/photo.png --> URL = www.domain.com/content/photo.png).
http.referrer	HTTP referrer	The URL contained in the referrer field of the HTTP GET request.
http.user_agent	HTTP user agent	The user agent field of the HTTP GET request.
user.id	User ID (anonymized)	An ID that uniquely identifies the user who requested the content.
observer.ip4, observer.device	Probe identifier	

In addition, the use case requires an active probe (Web Scraper) that acts upon request. This probe takes as an input a URL to visit and gives as an output the content of the HTML head of the URL.

In Repository Pre-process

A set of filtering algorithms will be implemented in mPlane repositories. Those algorithms will perform tasks like: exclude a set of useless URLs (those pointing to images, css files, queries that pass variable, etc), and retain only URLs that are of general purpose and that can be recommended to the users of the system. It is still unclear which algorithms can run in a repository and which algorithms will be part of the implementation of a reasoner as part of WP4. This definition will taken at a later point in time when it is more clear which streams of data are available in which repository and what are the best virtual and physical entities for pre-processing this data in mPlane.

Data Volume

For a probe aggregating around 20.000 users, based on a preliminary study on a representative day, the expected volume should be the following: 770 rows/second on average (a peak of 1700 rows per second was observed). Each row has a mean size of 357 characters/bytes.

2.1.5 Measurements for Multimedia Content Delivery

In this use case the measured data corresponds to active measurements, targeting the quality of video streaming as perceived by the end customers. A large number of set-top-box-like probes periodically download content from a multimedia streaming service (e.g. YouTube, HLS, etc.), were streams typically have multiple, different quality/bandwidth alternatives.

The probes perform quality measurements (e.g. throughput) and provide results with typically 1 minute resolution. All probes are operated by a single ISP within its own, clearly defined network.

The following metrics are reported:

mPlane type	Description
video.id	ID of the video stream being monitored.
rate.video.kbps	Video nominal bitrate.
request-delay.video.ms	Delay of request to first byte of video.
bufferstall.video.count	Buffer underrun/playout glitch events.
server.ip4, server.ip6	Server IP address.
rtt.dns.ms	Server DNS resolution delay.
bandwidth.partial.kbps	TCP flow download throughput (avg).

In addition, for each probe, the nominal bandwidth of its connection is reported using the mPlane type: bandwidth.nominal.kbps.

Active RTT measurements:

mPlane type	Description
rtt.ms	RTT measurements between probes distributed in the ISP and the end-customer probes.
packet.loss	Packet loss measurements between probes distributed in the ISP and the end-customer probes.

Passive measurements in the ISP network:

Passive probe(s) operated by the ISP monitor a sub-set of the streaming flows being downloaded from the end probes (potentially all of them). The passive probe(s) are deployed at key points (e.g. backbone routers, CDN peering points) where a significantly large portion (possibly all) of streaming traffic can be observed.

The measurements it reports include:

mPlane type	Description
load.link	Load of the link.
load.path	Load of the network path.
bandwidth.partial.kbps	TCP flow download throughput, per flow (avg).

In Repository Pre-process

The following processing tasks might be handled inside mPlane repositories.

- Identify probes with low TCP flow download throughput with respect to its nominal bandwidth.
- Identify probes with high fluctuations of TCP flow download throughput.
- Correlation of throughput measurements of end user probes with ISP probe measurements.

Data Volume

For each probe three records are stored inside mPlane repositories for each minute. One record for each of the following types:

- HTTP video streaming.
- Active RTT/loss measurements.
- Passive ISP measurements.

2.1.6 Mobile Network Troubleshooting

A combination of probes across various parts of the network provide the required input in order to identify the core causes of poor mobile user experience.

The application probe provides on-demand information as perceived from the application point of view. Currently we focus on mobile video experience. Therefore the following information is logged.

mPlane type	**Description**
rate.video.kbps	Total data rate as indicated in payload.
duration.video.s	Video duration as indicated in the payload.
video.id	ID of the video stream being monitored
request-delay.video.ms	delay of request to first byte of video
bufferstall.video.count	number buffer underrun/playout glitch events
bufferstall.duration.ms	duration of buffer underrun/playout glitch event

The mobile OS probe offers on-demand information considering the device capabilities and the device status (CPU, Memory). Furthermore, an important aspect of this probe is to measure the cellular network conditions (associated cell tower, signal strength, bit errors, transmit characteristics, power state of the device, etc).

mPlane type	**Description**
load.cpu.avg	Device cpu load.
load.cpu.max	Max device cpu load.
load.mem.avg	Average device memory load.
snr.avg	Signal strength information with the connected point.
snr.max	Max signal strength information with the connected point.
snr.min	Min signal strength information with the connected point.
disconnect.count	Number of disconnections during flow.
handover.count	Number of hangovers during flow.
connectivity.ip	IP of used interface.
connectionpoint	ID of connected interface (e.g., cell ID, or AP mac).
bandwidth.nominal.avg	Nominal bandwidth of connected interface.
bandwidth.nominal.min	Nominal bandwidth of connected interface.
location.lat	Location information (if available).
location.lon	Location information (if available).
location.accuracy	Location information (if available).
device.id	Device/user identifier.

Furthermore, the mobile probe runs a version of Tstat. All the metrics described in section 2.1.1 are also extracted *per-flow*.

The mobile ISP probe captures, both passively and actively aggregated information at each access point (AP) such as number of associated devices, overall traffic, channel utilization, etc.

mPlane type	Description
device.id	AP identifier.
load.cpu.avg	Router/AP cpu load.
load.cpu.max	Max Router/AP cpu load.
Per connected client	(over a window that is configurable)
snr.avg	Signal strength information with each connected client.
snr.max	Max signal strength information with each connected client.
snr.min	Min signal strength information with each connected client.
connectivity.ip	IP of each client.
bandwidth.nominal.avg	Nominal bandwidth of connected client.
bandwidth.nominal.min	Nominal bandwidth of connected client.
disconnect.count	Number of disconnections during flow.
octets.lost	Lost packets.
octets.ip	Transfer statistics.
octets.tcp	Transfer statistics.
octets.udp	Transfer statistics.
Wireless interface	(over a window that is configurable)
connected.client.count	Number of connected devices to the AP.
load.link	Total load of the wireless interface .
octets.lost	Lost packets.
octets.ip	Transfer statistics.
octets.tcp	Transfer statistics.
octets.udp	Transfer statistics.
Backbone connection	(over a window that is configurable)
load.link	Total load of the wireless interface .
octets.lost	Lost packets.
octets.ip	Transfer statistics.
octets.tcp	Transfer statistics.
octets.udp	Transfer statistics.
bandwidth.nominal	nominal bandwidth of the backbone.

Furthermore, as with the mobile probe, the mobile ISP also runs Tstat. All the metrics described in section 2.1.1 are also extracted *per-flow*. Finally, the core network and service provider probes measure the performance of the core network and the CDN/Service provider as described in the previous sections.

Data volume

Currently the mobile probe (and Android probe) generates approximately 800KB of compressed data per day per devices. Uncompressed this is about 14MB/per day per device. Furthermore, the mobile ISP/AP probe generates on average 5MB/day data.

Notice that at the moment we log almost everything. However, as we are talking about constrained mobile devices, we will aim not to upload more than 1-2MB per day per device (primarily when the device is connected to WiFi and charging). Some unnecessary measurements will be only kept on the phone/router/cells and get discarded later.

Finally, there are currently 15 devices and 2 routers running the probe so the total volume will be relatively low.

2.1.7 Verification of Service Level Agreements

The metrics reflect downlink and uplink throughput and bandwidth measurements performed from end-devices during active tests, on top of download and upload file transfers from a centralized server. In particular, for each set of tests performed on a specific probe, the following metrics are computed, and exported through the following structure:

`<Probe_ID, User_ID, Timestamp, RTT, download throughput TCP, download throughput UDP, download throughput HTTP, upload throughput TCP, upload throughput UDP, upload throughput FTP, packet loss downlink, packet loss uplink, fraction downlink failure events, fraction downlink failure events, RTT, Jitter, Link Capacity Estimation>`

The used types are described in more detail and mapped to existing as well as newly added mPlane types in the following table:

mPlane type	Description
bandwidth.partial.kbps	Multi layer (UDP, TCP, HTTP) download/upload flow throughput (avg).
packets.lost	Packet loss in downlink/uplink.
downlink.failures	Transmission data failure in downlink (fraction of failed file download events).
uplink.failures	Transmission data failure in uplink (fraction of failed file upload events).
rtt.ms	RTT measurements between probes and server.
delay.jitter	Delay variation.
bandwidth.imputed.kbps	Estimated link capacity.

Data volume

The log size is about 50 KB, and in some cases it may be up to a maximum of 200 KB. Measurements are performed every 20 minutes, and 5 consecutive measurement tests are exported every 2 hours.

The number of probes and the resulting data volume is as follows:

- User SLA tests: in this case, all the interested users can download the probe agent to perform the tests; according to experience, in rash period, about 100 user probes can export measurement results consecutively, resulting in a top average export volume of 50MB per hour.
- ISP SLA tests: in such a case, taking the Italian ISP context where FUB actually operates, there are 20 regions in all Italy to cover, each one with a maximum of 10 probes, one for each local

ISP. A top average export volume of 100MB per hour can be expected in this scenario.

2.2 External Repository Access

This section describes which external data sources we plan to use in mPlane and how those can be accessed by mPlane components.

2.2.1 Network Diagnostic Tool

Network Diagnostic Tool (NDT) is a sophisticated network active probing framework developed within the Google mLab initiative [17]. It consists of several components but it is basically a client/server application. Both client and server processes are used to perform a specific set of tests. The server processes include a basic web browser (fakewww) to handle incoming web based client requests. The server also runs a second process (web100srv) that performs the specific tests needed to determine what problems, if any, exist. The web100srv process then analyzes the test results and returns these results to the client. In contrast to most other active probing platforms the collected data are publicly available in raw format (i.e., archives stored in the Google cloud), via an SQL interface, and also through an advanced web portal.

A rich set of metrics are available and data collected span across the last few years. They represent then a very interesting source of data.

2.2.2 Maxmind GeoIP Organization DB

To enable the study of CDNs and cloud services it is fundamental to able to classify the traffic not only with the services accessed by the users, but also with the ``organization'' (e.g., Akamai, Google, Facebook, Level3, etc.) responsible for those services. For instance, considering HTTP queries, the contacted hostnames can contain some keywords (e.g., `facebook`, `google`, `akamai`, `amazon`, etc.) which highlight this association. Unfortunately, the presence of such keywords is not always guaranteed (e.g., the service might adopt HTTPS).

Generally speaking, there is the need of a mapping function capable of associating an IP address to the name of the organization which owns it. The `whois` Unix client is an example of such technology. More interesting solutions are the Maxmind GeoIP Organization Database [15], a commercial tool developed by Maxmind. Differently from `whois`, the Maxmind database maintains a static table of (IP address, organization name) pairs which enable fast lookup operations.

Maxmind is not the only company offering databases performing such a mapping function. Other vendors include IP2LOCATION [13], WhatIs MyIPAddress [20], WipMania [22], IPligence [14], and Neustar [18]. All these solution share the same basic principles and unfortunately, to the best of our knowledge, it is not know up to which extent they offer similar results. Nevertheless, their adoption is strategic when targeting the study of CDNs and cloud services.

We point out that the very same service could be accessed, among many others, via TopHat as describes in Section 2.2.3. This raises the opportunity for either developing mPlane wrappers for each specific service, or for ``aggregators'' such as TopHat that already gathers data from multiple external repositories using a single common API interface.

2.2.3 TopHat

The TopHat [6, 3] measurement service was developed in the context of the OneLab [9] Experimental Facility with the dual role of performing large-scale network measurements, and of supporting testbed experimenters with measurements, beyond its original goal of monitoring the PlanetLab Europe overlay network. TopHat supplies measurements to its users thanks to its own TDMI platform, which is supplemented by drawing upon several independent specialized measurement and monitoring infrastructures, some of which have a proven track record of excellence in providing specialized measurements to the research community. TopHat is responsible for transparently collecting and aggregating measurements originating from various autonomous sources. It relies on a modular and extensible component, named Manifold, for its interconnection framework, as well as its API and GUI interfaces.

The TopHat Dedicated Measurement Infrastructure (TDMI) is TopHat's own measurement infrastructure. It consists in modular probing agents that are deployed in a slice of the various PlanetLab nodes and probe the underlying network in a distributed efficient manner. In addition, they probe outwards to a number of target IP addresses that are not within PlanetLab. The aim of TDMI is to obtain the basic information required by TopHat. It implements such algorithms as Paris Traceroute to remove the artifacts arising from the presence of load balancers in the Internet topology, as well as FastPing, which can efficiently measure delays towards a large amount of end hosts on the Internet.

TopHat currently provides access to a set of interconnected system of different types, among which other measurement platforms (e.g. SONoMA), some system-level monitoring infrastructure and testbed resources (to determine under which conditions the measurements were performed). Whereas access to these active measurement probes is WP2 related, and hence is described more in details in mPlane Deliverable D21 (Sec. 3.7), in this context TopHat is relevant as it also grants access to a set of *existing repositories*, namely:

- Team Cymru IP-to-ASN mapping service [1]
- MaxMind Geolite [15]
- GeorgiaTech AS taxonomy dataset [2]

We notice that some of the above (e.g., MaxMind) can also be access directly, making the effort of writing an mPlane gateway that is specific for that service. Yet, by writing an mPlane gateway toward TopHat, this would enable to access a larger sets of external repositories with a single API. Adding to this the already mentioned advantage in terms of active measurement probes described in Deliverable D21 (Sec. 3.7), this makes it a natural candidate for integration.

2.2.4 Mirrored Sources

The integration with external data sources is strictly dependent from the characteristics of the data.

- **NDT** offer a large volume of data so mirroring/integrating it as a whole is not a valuable option. However, the framework offers very efficient APIs and query language to access to the data. Moreover, it is still possible to occasionally downloaded data archives for running specific analysis.

- **Maxmind GeoIP Organization DB** consists of only a 60 MB binary (or CSV) file. This compact format make it easy to be managed and replicated on several repositories. However, to avoid multiple join operations, input data can also be ``enhanced'' by running the join operation once and storing the result for later processing. The database is released with the option of monthly updates which make it easy to maintain.

2.3 mPlane Import Mechanisms

This section describes mechanisms to transfer high amounts of data between different physical machines which might be used to transfer data from mPlane probes to repositories. Although the mPlane architecture provides a description of a potential mechanism for data transfer between probes and repositories, we review some already implemented solutions in this section. The mechanisms presented here focus on high data volumes and failure recovery. Repositories might import data using one of those mechanisms along with the collect verb or an extension of it, see Section 4.

The remainder of this section describes the log shipping system used together with Tstat probes called log_sync, followed by a description of Apache Flume, used for log shipping together with Hadoop and other systems. The section ends with a description of Apache Sqoop, enabling data exchange between Hadoop and RDBMS. Apache Sqoop might be utilized to enable in-repository queries, which combine the processing power of traditional RDBMS with the parallel map reduce processing capabilities offered by Hadoop.

2.3.1 Tstat log_sync

Tstat (TCP STatistic and Analysis Tool) is an Open Source passive monitoring tool developed by the Telecommunication Network Group of Politecnico di Torino [10, 19]. Tstat rebuilds TCP/UDP connections and monitor them to extract a set of aggregated metrics. It supports different output formats including text log files and Round Robin Databases (RRDs) (please refer to deliverable D2.1 and D5.1 for more details [16]). Output data are then periodically exported using a framework called `log_sync`. This system is based on a client-server paradigm. Servers run on the probes and are responsible for preparing the data for the export. This includes applying compression and create proper *archives* which represent a monolithic data update. Data transfers are initiated by the client which contacts a servers to i) obtain an updated catalog of archives ready to be exported and ii) requests the archives it is interested into (typically all). Once a transfer is complete, the client consolidate the data on larger repositories or push the data on BigData/DBMS system for further processing.

In a scenario based on Tstat probes `log_sync` need to handle few probes. A centralized deployments result then in a simple and effective solution. In this case a single client and a single large repository should be sufficient. To handle more complex scenarios, different clients can be used to pull data from servers. However, the system is designed so that each server can communicate only with a specific client, while a client access to data from multiple probes. This semi-centralized architecture allows to obtain a simple and easy to maintain system. Moreover, by imposing specific relationships between clients and servers, it is possible to adopt Access Control List (ACL) policies to enforce entity authentication and improve the overall system security.

In the following are reported some details of the system architecture. A prototype is already build and deployed to control Tstat probes described in the deliverable D5.1 [16]. Notice however that

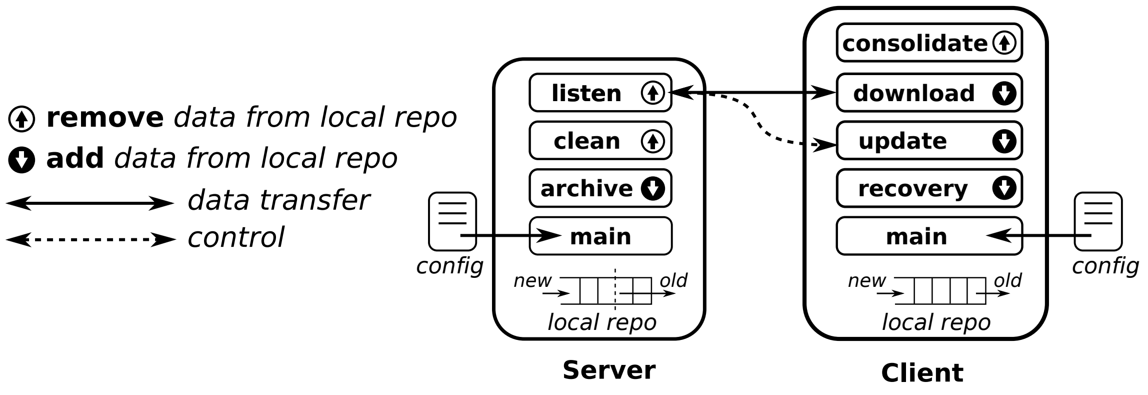


Figure 2: `log_sync` client and server modules.

further improvements are needed and not all the feature of the system are already implemented.

2.3.1.1 Exploiting parallel HTTP downloads

`log_sync` uses HTTP/HTTPS as communication protocol, adopting a RESTfull interface to define the set of application commands. This design choice is based on two motivations. First, HTTP/HTTPS is a very flexible protocol and adopted in several systems today. Second, beside data export functionalities, `log_sync` also incorporates a ``dashboard'' which offers a quick status report on the probes activities (e.g., CPU, memory, disk utilization, system load, etc.). By adopting HTTP/HTTPS, the dashboard can incorporated into web portals offering an easy way to control probes status.

Client and server need to perform several operations to ensure successful data transfer and data consistency. In particular, given that the volume of data collected by the probes can be large, it is fundamental to design a system capable of fully exploit the available download bandwidth. Standard file transfer tool such as `scp` and `rsync` for example transfer a single file at time using a single TCP connection and, due to some internal buffering mechanisms, they usually do not saturate the available download bandwidth. To guarantee optimal performance the download process has to based on parallel downloads, i.e., difference pieces of the files are downloaded at the same time over different connections.

Another important aspect to consider when designing the data transfer is the possibility of network failures. Considering a homogeneous scenario in which both `log_sync` clients and servers are inside the same network (e.g., an ISP internally managing its probes), we can assume a very low failure rate. However, in case the system needs to manage probes located in different networks (e.g., Tstat probes located in different countries), transfer failures are more realistic. Given the need to use parallel download, using multiple connections, each one downloading a different file can be inefficient. In fact, a failure can result in the need to entirely re-download all interrupted transfers. Instead, by splitting large files in smaller blocks of data individually transferred, we can improve resiliency to failures. In fact, in case of errors the systems need only to re-download the missing or corrupted file blocks.

2.3.1.2 System components

Particular attention has been dedicated to provide enough flexibility for the configuration of the system. In particular, beside offering command line switches, both client and server can be controlled through INI configuration files, i.e., text files containing a list of key-value pairs. For example, beside providing basic information to specify the probes registered in the system, it is possible to configure more fine grained parameters such as the logical size of each block to transfer and the maximum number of parallel connections.

The software has been written in Perl with a quite extensive usage of multi-threading. For efficiency, `log_sync` client and server have been implemented as a collection of modules, each one dedicated to a specific operation. Fig. 2 shows the internal composition of both client and server. Notice that both client and server handle a local repository. As highlighted by the arrows in the figure, modules interact with the local repository either adding or removing data from it.

A server is composed by the following modules:

main, is responsible for the configuration of the server by parsing both command line options and configuration files. Based on the provided configuration, it starts the other modules and control the overall status of the server.

archive, is periodically started to check for availability of new data generated by the probe. These data are then prepared to be exported by creating tarball, i.e., a set of files/directories which constitute a new set of statistics are compressed and archived together using the `tar` command line tool to create monolithic block of data.

For each archive the module also generates a *metafile* that describes the archive properties. These include for example the list the files included in the archive along with their attributes (e.g., size, timestamps, etc.). A *metafile* also includes the list of logical block in which the archive is split. This is used by the parallel download process as further described later.

Both the archives and the associated *metafiles* are stored in the local repository of the probe until they are removed by the `clean` module;

clean, periodically removes old files from the local repository. In particularly, the cleaning in based on a LRU policy which removes from the repositories only the files older than a configurable amount of hours/days or by explicit requests either by the main module or a `log_sync` client. This is highlighted in Fig. 2 by the vertical dashed line which divide the local repository in two parts: all data on the right of the threshold are too old and can be removed. This means that the system can be configure to do not immediately delete files which have been correctly transferred. In fact, some errors might occur before the client consolidating data on the repository. By using a LRU cleaning policy, the system still offer the possibility to recover from data corruption or losses after the transfer.

listen, implements the communication interface through which `log_sync` clients can communicate with the server, as show in Fig. 2. The core of the module is a simple web server which listening on a configurable TCP port and runs further operations based on the input commands received. For example, with the HTTP query `GET /repository/list` a client can obtain the list of archives ready to be transferred. To download a file in a single transfer the client needs to issue a request such `GET /repository/download/filenameXYZ`, while in case it is interested only on a portion of the file the request becomes
`GET /repository/download/filenameXYZ?range=START:END`.

A client is instead composed by the following modules:

main, similar to the server module, is responsible for the configuration of the client by parsing both command line options and configuration files. It is also responsible for the overall functioning on the client and control the status of the other modules.

recovery, periodically invoked by the main module to inspects the local client repository looking for broken files related to unsuccessful previous transfers. For each broken file found the module reschedules its download either as whole or only for the broken/missing portions.

update, periodically starts the procedure to pull data from registered probes. More in details, the module contacts all registered probes asking for an updated list of the archives available to be transferred, i.e., the list of the archives *metafiles* in the repository. Each *metafile* is in turn downloaded to retrieve the logical partitioning on the associated archive in block, and each block is scheduled for download using the `download` module;

download, corresponds to a simple FIFO queue in which are collected ``objects'' to be downloaded. These correspond to either files to be download at once (e.g., *metafiles*) or specific portion of a file (e.g., a block of an archive). The queue in internally handled with a listener which wait until data are available in the queue and extract up to a configurable amount of objects which are downloaded in parallel using a multiple threads, each associated to a different TCP socket. As shown in Fig. 2, this require communication with the `listen` module of the server.

consolidation, as suggested by the name, it consolidates the downloaded archives on a larger NAS. Optionally it can also start some configured pre-processing operations to transform the data before consolidate them on the NAS.

2.3.2 Apache Flume

Apache Flume [4] is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of data from many different sources to a centralized data store. It is a top-level project at the Apache Software Foundation.

The basic unit of data in flume is called *Event*. To Flume, an event is just a generic blob of bytes. Each *Event* flows from a *Source* to a *Sink* through a *Channel*. A *Flume Agent* is a process that consists of a *Source*, a *Channel* and a *Sink*. A Source consumes Events from a client or another Flume agent and stores it in a Channel. The Channel is a temporary buffer that holds the events until they are consumed by a Sink. The Sink removes events from a Channel and stores it in into an external repository, like HDFS, or forward it to the Sink of a chained Flume agent.

Flume is designed to transport and ingest regularly-generated event data over relatively stable, potentially complex topologies. The key property of an event is that they are generated in a continuous, streaming fashion. If your data is not regularly generated then Flume will still work, but it is probably overkill for your situation.

Flume is designed in order to guarantee the reliable delivery of events. The sources and sinks respectively store and retrieve data to and from the Channels using a transaction based mechanisms. A transaction is a indivisible unit of work that must succeed or fail as a complete unit. No intermediate states are possible. The use of transactions ensures that each event is reliably passed from point to point in the flow. This mechanism works also in case of multi-hop flows, where more flume agents are chained: indeed, the sink of the previous agent and the source of the next hop agent use transactions in order to store events in the channel of the next hop.

The channel manages also recovery from failures. In fact channels can support durable storage, backing up the data in the local file system or in a database.

Flume supports also the multiplexing of events to more than one destination. An event can be replicated (or selectively routed) to more than one channel. In case of replication, each flow is sent to multiple channels. In case of selective routing, an event is delivered to a subset of channels when an attribute matches a series of rules.

Flume supports also load balancing and handling of failures.

Failover Sink Processor maintains a prioritized list of sinks, guaranteeing that so long as one is available events will be processed and delivered. The failover mechanism works by relegating failed sinks to a pool where they are assigned a cool down period, increasing with sequential failures before they are retried. Once a sink successfully sends an event, it is restored to the live pool.

Load balancing sink processor provides the ability to load-balance flow over multiple sinks. It maintains a list of active sinks on which the load must be distributed and selects the one to use using either round robin or random selection.

2.3.3 Data Exchange with large-scale Batch Processing Systems

Large-scale batch processing systems operate on a very large amount of data, which can scale to the order of thousands of terabytes. A classic example of such kind of systems is Apache Hadoop (`hadoop.apache.org`), which implements a computational paradigm named MapReduce, where the application is divided into many small fragments of work, each of which can execute or re-execute on any node in the cluster. In general, this kind of systems provides their own implementation of a storage layer, in order to minimize large data transfers. The Apache Hadoop Distributed File System (HDFS) is the de-facto standard filesystem for these systems. HDFS stores large files (typically in the range of gigabytes to terabytes) across multiple machines.

In a typical data processing workflow, data is first imported in the HDFS from the probes; the result of the analysis is stored in the HDFS and possibly analyzed again. The ratio between the data to analyze and the output of the analysis varies: jobs can expand their inputs as well compress them. In dependence of the characteristics of the jobs (the jobs can produce only temporary files) and of the produced output (data volume can be huge), some part of data remains on HDFS and is not imported into any mPlane repositories. Note that data that needs to be processed by large scale systems is directly loaded into HDFS from the probes, without passing through any relational database.

The transfer of data from Hadoop to relational databases is done using an Open Source project, Apache Sqoop (`http://sqoop.apache.org/`), the de-facto industry standard for data import and export from Hadoop to relational databases. Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases. Sqoop can both import data from external relational database into HDFS and populate tables in Hive and HBase, and export data from Hadoop to relational databases. Any import or export operation can be automatized and can happen periodically or event based, since Sqoop is highly integrated with Oozie (`http://oozie.apache.org/`), a workflow manager for Hadoop. Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance. The import process is illustrated in Figure 3. The dataset being transferred is divided into different partitions and a map-only job is launched with individual mappers responsible for transferring a slice of this dataset. Each record of the data is handled in a type safe manner, thanks to the hidden interaction with the database metadata to infer the data types.

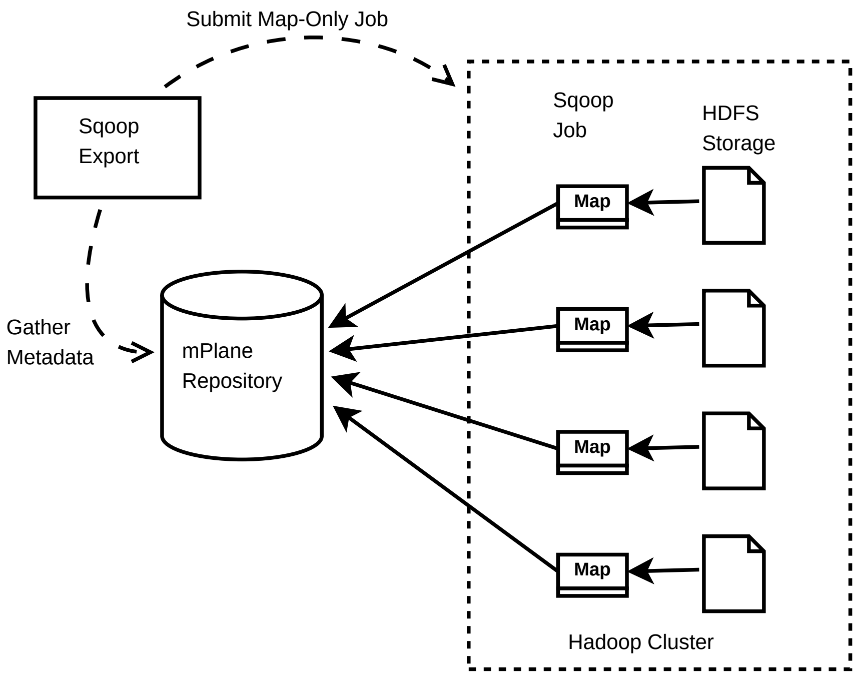


Figure 3: Overview of the export process of data from HDFS to any mPlane repository, using Sqoop

Export is done in two steps. The first step is to introspect the database for metadata, followed by the second step of transferring the data. Sqoop divides the input dataset into splits and then uses individual map tasks to push the splits to the database. Each map task performs this transfer over many transactions in order to ensure optimal throughput and minimal resource utilization.

Since Sqoop breaks down export process into multiple transactions, it is possible that a failed export job may result in partial data being committed to the database. Therefore, in order to isolate production tables during the import process, intermediate *staging tables* can be used. They acts as auxiliary tables, used to stage exported data. Staging tables are first populated by the map tasks and then merged into the target table once all of the data has been transferred, in a single transaction.

The connection with external databases is done using connectors or JDBC. Sqoop includes connectors for various popular databases such as MySQL, PostgreSQL, Oracle, SQL Server and DB2. It also includes fast-path connectors for MySQL and PostgreSQL databases. Fast-path connectors are specialized connectors that use database specific batch tools to transfer data with high throughput. Sqoop also includes a generic JDBC connector that can be used to connect to any database that is accessible via JDBC.

The degree of parallelism depends on the number of launched map tasks, and it can be configured independently from the size of the data to import. By default, only four tasks in parallel are used for the export process. Anyway, this degree can be tuned appropriately in the final system in order to achieve the best trade-off between load on the database and import speed. Indeed additional tasks may offer better concurrency, but if the database is already overcharged additional load may decrease performance.

Note that the export is not executed as an atomic operation: partial results from the export will become visible before the export is done. But atomicity is not required, since the amount of data involved in the export would lead to a useless enormous complexity. In fact, exports are performed by multiple writers in parallel. Each writer uses a separate connection to the database; these have separate transactions from one another. Sqoop uses the multirow INSERT syntax to insert up to 100 records per statement. Each INSERT is executed in a single transaction within a writer task. This ensures that transaction buffers do not grow without bound causing out-of-memory conditions. Therefore, an export is not an atomic process.

3 Data Processing

Some repositories in mPlane, like DBStream, are not only used simply to store data, but also to process the imported data continuously. First, data can be processed to generate index structures, enabling fast access to certain sub sets of the imported data as shown in Section 3.1. Second, data can be aggregated, multiple data streams can be combined or data can be enriched with information from external data sources. This data processing is called materialized view generation and further detailed in Section 3.2. Although materialized view generation might be important for some use case it is not considered to be a standard feature among mPlane repositories.

3.1 Data Indexing

Data indexing is strictly related to the type of analysis to run. Generally speaking, we report a set of considerations that can enable a faster lookup operation on the collected data:

- **Time**: data collected from monitoring probes are intrinsically related to time. This is true independently if the probe is active or passive and if measures are occasional or continuous. As such, the larger the data set, the greater the need of an index to enable fast searches.

  Indexes can corresponds to specific data structure (e.g., B-trees, hashtables, etc.) or can be the result of data partitioning. For instance, both Tstat and NDT collect text logs files in a hierarchy of directories related to the time of capture (e.g., set of logs are grouped on a hourly base).

- **End-users' characteristics**: one of the primary roles of monitoring systems is to study the performance and behavior of hosts located in a specific/private network. For instance, ISPs can be interested in understanding which Internet applications their customers access to and what are the obtained performance. However, not all the users present similar characteristics. In fact, they might use different devices (e.g., laptop, tablets, smartphones, set-top-boxes, etc.), access technologies (e.g., ADSL, FTTH, 3G/4G, WiFi) or Internet applications (e.g., P2P, video streaming, cloud storage, etc.).

  All these characteristics are valid factors to consider when considering data indexing. Notice that monitoring probes can be programmed to directly partition the traffic based on the previous characteristics. For instance a passive probes might track only specific device types (e.g., Android devices) or streaming services (e.g., YouTube). However, by offloading advanced filtering function to repositories the probes can results simpler while increasing the overall system flexibility (e.g., collect once and post-process multi times by changing the target function).

- **CDNs and cloud services network infrastructures**: similarly to the previous point, data collected from monitoring probes can be used to study the infrastructure of the services accessed by end-users. For instance, by inspecting the network properties of the YouTube servers contacted it is possible to infer their location. Moreover, by using technologies such as the Maxmind GeoIP Organization name database previously introduced, it is possible map each server IP address to the name organization which owns it. In other words, it is possible to isolate the Akamai traffic from the whole aggregate and study the CDN network infrastructure (e.g., data center location, load balancing and caching policies). In these cases both the organization names and server IP addresses correspond to important information to index.

3.2 Materialized View Generation in DBStream

DBStream is a continuous analytics system. Its main purpose is to process and combine data from multiple sources as they are produced, create aggregations, and store query results for further processing by external analysis modules or visualization. The system targets continuous network monitoring but it is not limited to this context. For instance, smart grid, intelligent transportation systems, or any other use case that requires continuous process of large amounts of data over time can take advantage of DBStream.

DBStream combines on-the-fly data processing of Data Stream Management Systems (DSMS) with the storage and analytic capabilities of Database Management Systems (DBMS) and typical ``big data'' analysis systems such as Hadoop [21]. In contrast to DSMSs, data are stored persistently and are directly available for later visualization or further processing. As opposed to traditional data analytics systems, typically importing and transforming data in large batches (e.g., days or weeks), DBStream imports and processes data in small batches (in the order of minutes). Therefore, DBStream is like a DSMS in the way that data can be processed fast, but streams can be re-played for past data. The only limitation is the size of available storage. DBStream thus supports a native concept of time. At the same time DBStream provides a flexible interface for data loading and processing, based on the declarative SQL language used by all relational DBMSs. Two important features of DBStream are:

- It supports incremental queries defined using a declarative interface based on the SQL query language. Incremental queries are those which update their results by combining newly arrived data with previously generated results rather than being re-computed from scratch (see Sec. 3.2.2 for more details). This enables efficient processing of two interesting groups of queries. First, a set of items can be monitored over time by looking at the last state plus the new data, e.g., monitoring the set of all server IP addresses that are active within a sliding window of time such as in the last two weeks. Second, aggregated variables can be kept for the elements of the monitored set, e.g., the number of bytes uploaded and downloaded by each IP over a sliding window of time.

- In contrast to many database system extensions, DBStream does not change the query processing engine. Instead, queries over data streams are evaluated as repeated invocations of a process that consumes a batch of newly arrived data and combines them with the previous result to come up with the new result. Therefore, DBStream is able to reuse the full functionality of the underlying DBMS, including its query processing engine and query optimizer.

DBStream is built on top of a SQL DBMS back-end. We use the PostgreSQL database in our implementation, but the DBStream concept can easily be used with other databases and it is not dependent on any specific features of PostgreSQL.

In the mPlane project DBStream will serve as one example for a repository and processing facility. At the moment already a sample implementation, importing data from Tstat [10] monitoring probes exists. One of the next steps is to integrate data exchange with Hadoop as already mentioned in Section 2.3.3. Most importantly, data from some of the sources mentioned already in Section 2.1 is imported and processed in DBStream.

3.2.1 Architecture

In DBStream, *base tables* store the raw data imported into the system, and *materialized views* (or views for short) store the results of queries such as aggregates and other analytics - which may

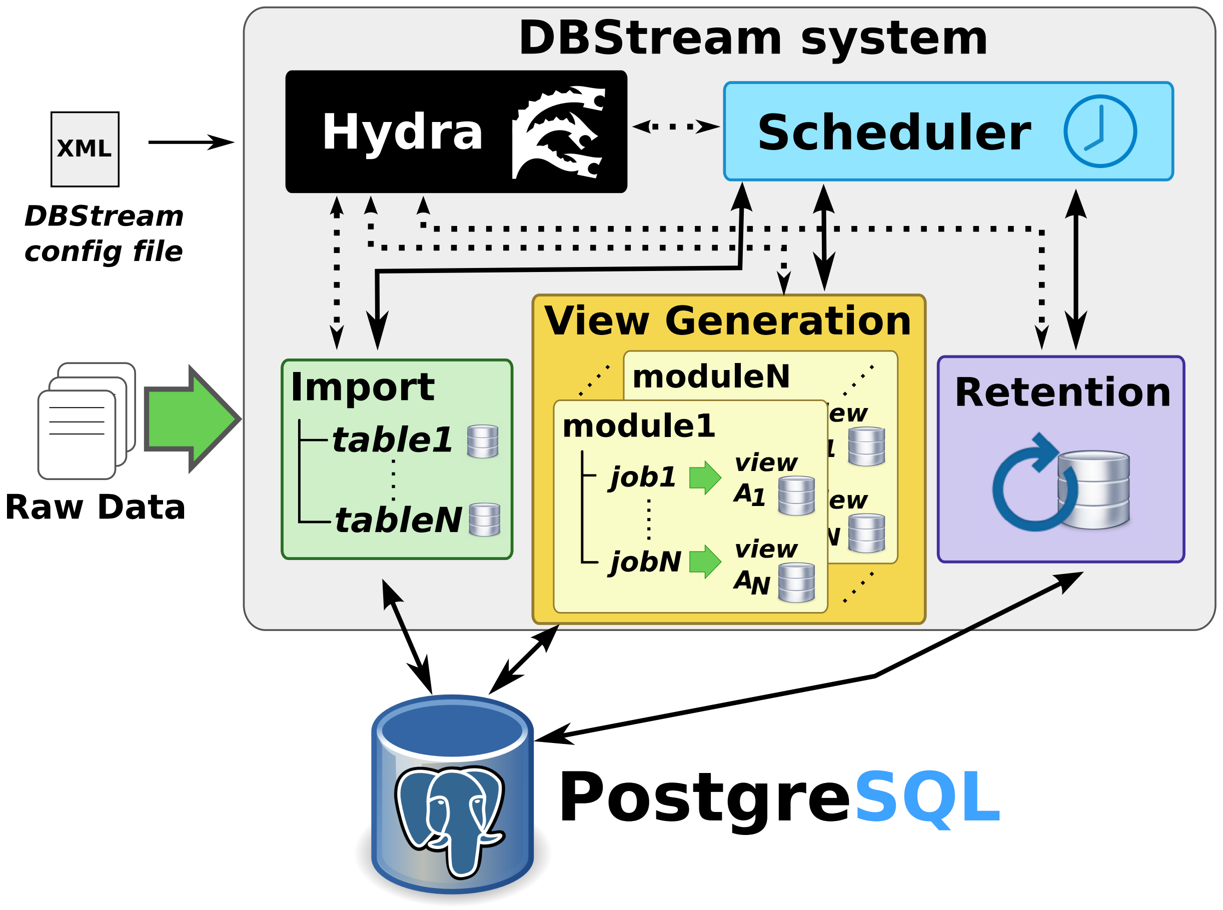


Figure 4: General overview of the DBStream architecture.

then be accessed by ad-hoc queries and applications in the same way as base tables. Base tables and materialized views are stored in a time-partitioned format inside the PostgreSQL database, which we refer to as *continuous tables*. The entity that updates a base table or a view in response to the arrival of a new batch of data is called a *job*. We refer to a batch of data as a window or slice; for example, if new data arrive every minute, then each batch is a non-overlapping window/slice of size one minute.

Each time partition of a base table corresponds to data from one window. Each time partition of a view corresponds to the result of a query at that point in time. That is, in addition to the current result of the given query, DBStream stores previous results for historical analysis. Time partitioning makes it possible to insert new data without modifying the entire table; instead, only the newest partition is modified.

Fig. 4 gives a high-level overview of the DBStream architecture. DBStream consists of a set of modules running as separate operating system processes. The `Scheduler` defines the order in which jobs are executed. `Import` modules may pre-process the raw data if necessary, and signal the availability of new data to the `Scheduler`. The scheduler then runs jobs that update the base tables with newly arrived data and create indexes, followed by incrementally updating the materialized views and creating indexes. Each view update is done by running an SQL query that retrieves the previous state of the view and modifies it to account for newly arrived data (we will discuss this in more detail in Section 3.2.2); new results are then inserted into a new partition of the view and indexes are created for this partition. For each base table, a `View Generation` module contains the job definitions that maintain the views defined on that base table. Finally, the `Retention` module monitors base tables and views, deleting old data based on predefined storage size quotas and other data retention policies.

The central module of DBStream is the `Scheduler`, which decides which job will run next. For each base table, the availability of a new window of data triggers a new job; when done, a new job will be triggered for each view defined over this base table, and then new jobs will be triggered for views defined over these views and so on (i.e., views may be defined over other views, forming deep hierarchies). If the system falls behind, a given table or view may have multiple jobs pending, one corresponding to each window of data that must be loaded. The scheduler ensures that, for any given table or view, windows are processed in chronological order.

Another critical function of the scheduler is to avoid resource contention: with a very large number of base tables and views, we need to limit the number of concurrent update jobs. Our experience so far has shown that the maximum number of concurrent jobs may be set to the number of parallel threads that can be executed by the CPUs in the system. Thus, the scheduler will allow jobs to be executed until this limit is reached, and will put any other pending jobs in the execution queue. If a thread becomes available, the scheduler will choose a job from the execution queue to run on that thread according to some scheduling policy. In the current implementation, a simple first-in-first-out (FIFO) algorithm is supported, and improved schedulers is a work in progress.

The `Retention` module is responsible for implementing data retention policies. For example, a maximum size may be defined for each base table and view, and if their sizes grow larger than maximum, the oldest data will be deleted. This module is critical since data are continuously imported into DBStream and the available storage may fill up quickly. Since each base table and view is partitioned by time, deleting old data is simple: it suffices to drop the oldest partition(s).

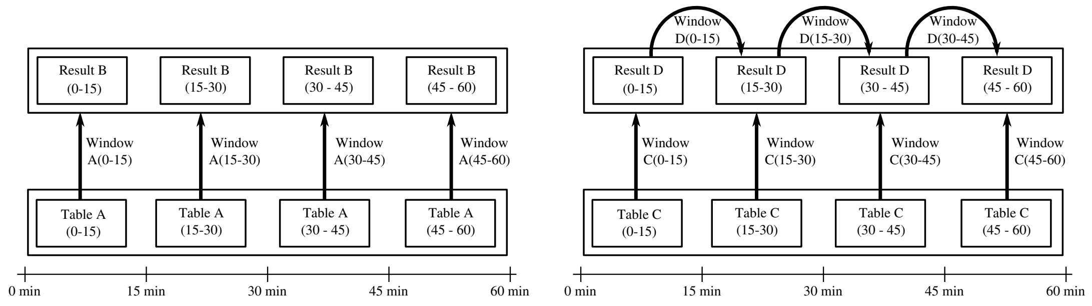
The DBStream system is operated by an application server process called `hydra`, which reads the DBStream configuration file, starts all modules, and monitors them over time. Status information is fetched from those modules and made available in a centralized location. In case a module crashes, it is restarted automatically after a predefined waiting period. This mechanism is important to ensure that modules, which might depend on processes possibly running on remote machines, continue to work even after reboots or temporary failures of those remote machines. All DBStream modules are implemented such that they can be stopped and started at any point in time, always leaving the system in a recoverable state.

The inter-process communication in DBStream, e.g., between `hydra` and other modules as well as between the `Scheduler` and the `View Generation`, is implemented using remote procedure calls over HTTP. Modules can be placed on separate machines, and external programs can connect to DBStream modules by issuing simple HTTP requests.

3.2.2 Continuous Analytics Language

This section describes the user and application interface to DBStream, which exploits the declarative database query language, SQL, to define incremental updates to materialized views. We provide a high-level overview of the language using examples from the networking domain.

Many continuous and streaming query languages have been proposed [12, 7, 5, 8], but they assume that data are not stored persistently and that queries can only refer to temporary state (e.g., a current window of time). On the other hand, many DBMSs support defining materialized views using



(a) A continuous table being transformed into another.

(b) Complex data processing flow for an incremental query.

Figure 5: Data flow of two example jobs.

SQL queries over large historical tables, but incremental view maintenance over time is restricted to simple types of queries such as filters and joins. In contrast, DBStream enables users and applications to declaratively specify, using arbitrary SQL, exactly how to update a view when a new batch of data is inserted into its source table(s). These specifications may refer to previously generated results that are stored in the view, which, to the best of our knowledge, is not *declaratively* supported by any other system.

Fig. 5 shows two abstract examples of how views may be updated upon insertion of new data into their source tables; we will show concrete queries corresponding to these two cases in the next section. Fig. 5a illustrates a base table A that is updated every 15 minutes, i.e., it is partitioned into 15-minute slices. A view B is also maintained, which is the result of some query over A, e.g., A may store raw traffic data and B may store aggregated traffic volume for each source-destination pair over every 15-minute window. When a new window of data is added to A, the SQL query corresponding to B runs over the contents of the new window, and the result is inserted into the most recent window of B. Four such updates are shown in Fig. 5a, corresponding to the processing done over one hour. This example corresponds to the simple non-incremental case, without accessing history, that can be handled by DBStream as well as existing stream processing systems.

Fig. 5b illustrates incremental processing by accessing previously computed results. Here, C is a base table and D is a view, both of which are partitioned into 15-minute windows. The first (leftmost) partition of D is created by running D's SQL query over the first partition of C. However, the next partition of D is computed by running D's query over the next partition of C *and* the previous partition of D that has been computed 15 minutes ago. For instance, D could store all the active sessions in any given 15-minute window, in which case computing the active sessions at any time amounts to taking the active sessions as of 15-minutes ago, adding newly active sessions from the past 15 minutes, and removing sessions that have ended in the last 15 minutes.

Without the ability to refer to the previous result of D, a default way of maintaining D would be to read the entire base table C and extract all the sessions that are currently active (or at least read as far back as the longest possible connection, so that we can capture its connection-start record in C). This would be far less efficient than the above incremental technique. Since network monitoring involves many tasks that are computed incrementally over time, the incremental view maintenance approach of DBStream makes it an ideal candidate for the repository layer of mPlane.

3.2.3 Job Definition Examples

Let us now return to the two examples from Fig. 5 and show how such views can be set up in DB-Stream. Each view requires a user-supplied job definition that includes, among other things, the windows (partitions) that are necessary to compute a new partition of the view, and the SQL query that will transform the input windows into a result window. For Fig. 5a, an example job definition is as follows:

```
<job inputs="A (window 15min primary)"
     output="B (window 15min)"
     schema="serial_time int4,
             device_class int4,
             count int4"
     query="select serial_time, device_class,
            count(*) from A
            group by serial_time, device_class"/>
```

The `inputs` attribute defines the input window(s) and the `output` attribute defines the destination for the result. Here, B is the resulting view, partitioned by 15 minutes, each of whose partitions is computed from the corresponding partition of the base table A. The ``primary'' keyword denotes the leading input partition - when created, it triggers an update of the view. A job can have multiple inputs.

The schema of the output view is defined using the `schema` attribute; in addition to the attributes returned by the query, a timestamp attribute corresponding to the window end time is also included by default. The `query` attribute specifies the SQL query which is executed for every primary input window; DBStream supports all SQL queries that are supported by the underlying DBMS (PostgreSQL). Here, the query counts the number of packets for each combination of serial time and device class for each 15-minute window. The query includes a ``from A'' statement, which does not actually read all of A, only the window of A that was specified in the inputs statement (i.e., the most recent 15-minute window). In addition, the `index` attribute can be used to define indexes over the view.

For Fig. 5b, the following job definition maintains a view D with the currently active TCP sessions; assume that C stores passive probe data and contains a record whenever a connection is started or stopped.

```
<job inputs="C (window 15min primary),
             D (window 15min delay 15min)"
     output="D (window 15min)"
     schema="id int4"
     query="
   select id from D
         where not exists (
           select * from C
             where D.id=C.id and C.state='disconnect'
           ) union
       select id from C
         where state='connect'"/>
```

Note the ``delay'' keyword, which specifies a delayed version of D (more specifically, the previous result of D from 15 minutes ago) as one of the inputs, along with a new window of C, which is the primary window that triggers the creation of a new partition of D. The SQL query selects session ids from D that were active 15 minutes ago (i.e., from the previous partition of D from 15 minutes ago, as specified in the inputs statement), but only for those for which there is no disconnect record in the newest partition of C. Then the result is merged (unioned) with the newly started connections from (the newest partition of) C.

4 Data Access

This chapter describes how repositories enable access to their data for mPlane clients, supervisors and reasoners. The chapter is split into three parts, reflected in three sections. Section 4.1 defines and describes the verbs an mPlane repository should implement and report as capabilities to other mPlane components. The second Section 4.2 gives a description of other useful verbs, which might be implemented in mPlane repositories, but are not part of every repository. The last section, Section 4.3 explains how access control for mPlane repositories is handled.

4.1 General Repository Access Mechanisms

A mPlane repository is a physical or logical entity offering the functionality specified in this section. In general, mPlane repositories store data, keep it for a certain amount of time and make it available to other components of the mPlane architecture, like clients, supervisors and reasoners. The following list gives an overview of mPlane information model elements, so called verbs, which should be implemented and reported as capabilities by an mPlane repository.

collect is used to import data into an mPlane repository. Its main purpose is to initiate the import of data exported by an mPlane probe using an asynchronous export mechanism. It is optional to implement the `collect` verb if the `store` verb is implemented.

store is used to store single items of data in an mPlane repository. It is optional to implement the `store` verb, if the `collect` verb is implemented.

query is used to retrieve data from an mPlane repository. The verb `query` can not be used to run selective queries or queries which support any kind of processing inside a repository, but only the retrieval of stored data. A mPlane repository might implement an extended version of `query` offering additional functionality.

It is important to notice, that all data stored in mPlane repositories has to either expose a event time with every stored record, or the repository has to be programmed to infer a event time for each record it will store depending on the given stream type. In addition, the general verbs `collect` and `store` do not specify the handling of out of order events. Therefore, a repository might decide to discard out-of-order events.

In some cases there might be repositories which neither implement `collect` nor `store`, but only `query`. Those repositories are used in cases, where the underlying monitoring or probe infrastructure is not part of the mPlane or the probe directly exposes an mPlane repository interface. In this case, mPlane components can still access data stored in the repository using the `query` verb, but the mPlane is not controlling the import of data into the repository.

4.1.1 `collect` - Continuous Data Storage

The `collect` verb is used to import data continuously from mPlane probes. For this purpose, the repository reports one capability for each pre-programmed data type it is able to store. Therefore, the following capabilities will be reported for `collect`:

```
capability: collect
  parameters:
    stream.name: string
    async-format: string
    async-target: url
```

In this example, `stream.name` is a predefined name for an mPlane data format the repository is able to store, e.g. log_tcp_v01. The field `async-format` contains the format in which the data might be sent to the given repository. The format might be one of the mPlane protocol message representations e.g. JSON, YAML, XML or CSV. In the `async-target` the repository specifies a URL to which the exporter should send data to, as well as the transmission protocol. The `async-target` uses a export URL scheme defined in D1.3 and might look like this: `mplane-http://repo1.ict-mplae.eu`.

Some repositories might use one of the import mechanisms described in Section 2.3 instead of async-target.

4.1.2 `store` - Single Value Storage

The `store` verb is used to store single data values. It is not intended to use this verb for large data transfers towards repositories, but rather for the storage of single values e.g. produced by reasoners as the result of a machine learning algorithm. Therefore, the capabilities reported by the repository are rather flexible and depend highly on the preconfigured data format the repository is able to store.

Whereas, the general capability template for `store` looks like this:

```
capability: store
  parameters:
    stream.name: string
    start.s:     -inf..+inf
    [list of attributes]
```

The example of a reported capability for storing a IP address along with the measured link load might look like this:

```
capability: store
  parameters:
    stream.name: probe_linkload
    start.s:     *
    server.ip4:  *
    load.link:    *
```

The actual data is then send by e.g. the reasoner in the specification statement.

4.1.3 `query` - Data Retrieval

The `query` verb is used to retrieve data from an mPlane repository. The reference implementation of the `query` verb does neither support selective queries nor queries supporting any data processing. It is only used for the retrieval of data which was either directly imported before, using `collect` or `store`, or is the result of in repository processing. The template of capabilities exposed for `query` by the repository looks like this:

```
capability: query
  parameters:
    stream.name: *
    start.s: -inf..+inf
    end.s: -inf..+inf
  results:
   - (fields of the data stream)
```

In this example, `stream.name` is the name of the stream the data was stored with, `start.s` and `end.s` refer to the event times which where provided while storing the queried data. The results section will contain the names and types of the format of the stream. Therefore, a concrete example of a query verb might look like this:

```
capability: query
  parameters:
    stream.name: probe_linkload
    start.s: -inf
    end.s: 2013-10-31 23:59:59
  results:
   - start.s
   - server.ip4
   - load.link
```

4.2 Extending Repository Access

In general, every mPlane repository can report verbs in addition to the default implementation. Those additional verbs can be used to reflect functionalities, which are special to that particular repository, but are not reflected by the general mPlane architecture. Since many different underlying systems might be used as mPlane repositories many different capabilities will be reported and used by the mPlane.

4.2.1 DBStream Extensions

This section gives an example how the mPlane repository access layer can be extended by a specific repository. In this section, it is shown what other verbs will be implemented in DBStream. Other repository might implement their own verbs or also implement the verbs proposed here.

One new verb of DBStream will be an extended version of `query`, which also allows to filter and sort results. Therefore, the query verb will have an additional field to express a filter condition which is similar to a SQL WHERE clause. The returned result will be the subset of results fulfilling the filter condition. Another additional field, sort can contain SQL order by statements. Those can be used to sort the returned result set using a stable sort algorithm. The values provided by those two fields will be passed to the underlying SQL execution engine, which will execute the filtering and sorting of the results. The extended version of `query` will also provide other ways to return results, which are optimized for high data rates.

Another additional verb is an extended version of `collect`. `collect` is used in cases, where the import/export mechanism is not part of the general mPlane architecture but a more direct coupling of the repository to the probe. In those cases, the repository might use one of the mechanisms described in Section 2.3. Per example to import data from Tstat probes DBStream might import data which arrives at the database server using the log_sync mechanism. In this case the `collect` verb is only used to switch this import/export mechanism on and off.

Since one of the main purposes of DBStream is to process imported data and generate aggregated statistics, a special verb `process` will be implemented. In DBStream multiple processing algorithms might be pre-programmed and then activated and deactivated, as well as parametrized using the process verb. This mechanism is useful in cases where it is e.g. too computationally extensive to compute a certain learning algorithm for all subnets/hosts/IP addresses/etc. Therefore, the reasoner might activate a certain analysis e.g. only for one subnet. In this case the reasoner sends the repository a process specification along with the parameter defining the e.g. specific subnet which should be analyzed. In other cases, the parameter given to the process verb might be the result of a machine learning or data mining algorithm, and therefore change over time.

4.3 Repository Access Control

According to the architecture overview exposed in chapter 2.1 of the deliverable D1.3, the repository is involved in two types of interactions:

- the "synchronous data request", in which the Repository R publishes its capabilities to the Supervisor S, then S sends a specification message to R requesting for a capability, and finally R replies with a data flow that is the result for that specification.
- the "asynchronous data export", in which the probe P exports the collected data directly to R, creating a short circuit between the two components.

Regarding the interactions between Repositories and Supervisors, there is no need to implement the authorization procedures at the Repository, since all the Access Control mechanisms are implemented at the Supervisor level: the Supervisor checks if the specifications coming from the Clients are legitimate, and if so it forwards them to the appropriate component. The Probes and the Repositories only receive legitimate specification messages. For this reason, the Repository component only needs an Authentication layer, in order to check the identity of the Supervisor and accept its specification messages.

In the "asynchronous data export" case the Probe needs to establish a channel with the Repository, and to push data into it. The Repository must expose a public interface, and a capability expressing the ability to accept data from that specific Probe. This interface must be strictly controlled, in order to avoid unauthorized accesses or illegal requests.

A viable solution could be to delegate to the supervisor all the authorization controls on the data export, for example:

- P publishes a capability to S that indicates it can export a given measurement (perhaps explicitly to R)
- R publishes a capability to S that indicates it can collect the same measurement (perhaps explicitly from P)
- S publishes a capability based on these two
- Client C sends a specification to S asking to use this capability
- S checks C's credentials and the specification to see if it is authorized. If so:
- S sends a specification over a mutually authenticated channel to R to begin collection and awaits a receipt.
- S sends a specification over a mutually authenticated channel to P to begin export.
- P exports collected data over a mutually authenticated channel to R.

With this approach all the Authorization controls are enforced in the Supervisor, that is already responsible for the whole mPlane architecture security controls, relieving the other components from the implementation of those mechanisms.

References

[1] http://www.team-cymru.org/.

[2] http://www.ece.gatech.edu/research/labs/MANIACS/as_taxonomy/,.

[3] Tophat website. http://www.top-hat.info/.

[4] Apache Flume. http://flume.apache.org.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006.

[6] T. Bourgeau, J. Augé, and T. Friedman. TopHat: supporting experiments through measurement infrastructure federation. In *Proceedings of TridentCom'2010*, Berlin, Germany, 18-20 May 2010.

[7] Q. Chen and M. Hsu. Experience in extending query engine for continuous analytics. In *DaWaK 2010*.

[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD 2003*.

[9] S. Fdida, T. Friedman, and T. Parmentelat. OneLab: An open federated facility for experimentally driven future internet research. In *Proc. Workshop on New Architectures for Future Internet*, 2009.

[10] A. Finamore, M. Mellia, M. Meo, M. Munafò, and D. Rossi. Experiences of internet traffic monitoring with tstat. *IEEE Network*, 25(3):8--14, 2011.

[11] FireLog project. http://firelog.eurecom.fr/.

[12] L. Golab, T. Johnson, S. Sen, and J. Yates. A sequence-oriented stream warehouse paradigm for network monitoring applications. In *PAM 2012*.

[13] IP2LOCATION Home Page. http://www.ip2location.com/databases/db8-ip-country-region-city-latitude-longitude-isp-domain.

[14] IPligence Home Page. http://www.ipligence.com/products.

[15] Maxmind GeoIP databases and web services. http://www.maxmind.com/en/geolocation_landing.

[16] mPlane consortium. Public Deliverables. http://www.ict-mplane.eu/public/public-deliverables.

[17] Network Diagnostic Test (NDT) - Home page. http://www.measurementlab.net/tools/ndt.

[18] Neustar Home Page. http://www.neustar.biz/enterprise/ip-intelligence/ip-intelligence-packages#.UkXqPhYmzdk.

[19] Tstat Home Page. http://tstat.polito.it.

[20] WhatIs MyIPAddress Home Page. http://whatismyipaddress.com/geolocation-providers.

[21] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[22] WipMania WorldIP - IP Geolocation. http://www.wipmania.com/en/api/.