mPlane

an Intelligent Measurement Plane for Future Network and Application Management

ICT FP7-318627

Selection of Existing Probes and Datasets

Author(s):	ALBELL	D. Papadimitriou
	ENST	D. Rossi, YiXi Gong
	ETH	B. Trammell
	EURECOM	M. Milanesio, E. Biersack
	FHA	R. Winter
	FUB	F. Matera
	NEC	M. Dusi
	NETVISOR	B. Szabó, T. Szemethy
	POLITO	A. Finamore, M.Mellia
	TI	A. Capello, F. Invernizzi, O. Jabr
	TID	I. Leontiadis
	ULG	B. Donnet

Document Number:	D2.1
Revision:	1.0
Revision Date:	31 July 2013
Deliverable Type:	RTD
Due Date of Delivery:	31 July 2013
Actual Date of Delivery:	10 August 2013
Nature of the Deliverable:	(R)eport
Dissemination Level:	Private

Abstract:

The mPlane architecture has been designed to include the possibility to interface with existing systems and platforms. While most measurement platforms in existence target a very specific measurement use case (e.g., the discovery of the Internet's router-level topology, the continuous measurement of the RTT among host pairs, the exporting via SNMP of network state, etc.), there are platforms that have a large deployed base, with lot of data being at disposal, and/or continuously collecting data. It would be a waste of resources to merely reproduce this effort within mPlane. Instead, mPlane aims at directly interfacing with existing systems and re-using their capabilities and data to feed measurement results to the mPlane intelligence. This document lists selected existing systems that are important for mPlane either for theoretical, conceptual or practical reasons, and that are part of the background of mPlane partners. A sub-set of these systems will be eventually incorporated into mPlane by developing the necessary interfaces. Others could be integrated by the means of proxy probes, i.e., the conceptual component responsible for such interfacing. The main focus of this document is to elaborate the concept of proxy probes, enumerate the systems that will be possibly considered for interface (proxy probe) development, and to give high level descriptions of the proxy probe design for these systems. The following list enumerates the systems that the consortium has chosen to include:

- QoF - a TCP-aware IPFIX flow meter
- Cisco Ping and SLA Agents - commercial availability and basic network parameter agents
- Tracebox - a tool for middlebox detection and identification
- Scamper - a sophisticated active probing tool
- MERLIN - a router-level topology discovery tool
- TopHat - a configurable measurement system on top of PlanetLab
- Tstat - a passive network monitoring tool
- BlockMon - a flexible network monitoring and analysis tool
- MisuraInternet - a QoS measurement system
- Firelog - a Firefox plugin to measure HTTP QoE
- Pytomo - an end-host-based video QoE measurement tool
- DATI - a high performance deep packet inspector
- MobiPerf - a tool for monitoring smartphone performance

Keywords: existing probes, proxy probes

SEVENTH FRAMEWORK PROGRAMME

Disclaimer

The information, documentation and figures available in this deliverable are written by the mPlane Consortium partners under EC co-financing (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission.
The information in this document is provided ``as is'', and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

SEVENTH FRAMEWORK PROGRAMME

Contents

Executive Summary

This document provides a list of existing systems that classify as *probes* under mPlane's taxonomy. The list - and this document - serves the following purposes:

- Enumerate and discuss a comprehensive list of important existing systems implementing mPlane's *probe* concept. The mPlane consortium has in-depth background knowledge about the systems selected, so incorporating these systems into mPlane can be carried out with reasonable effort. The systems selected represent all major probe development directions, thus iterfacing them with mPlane shows how to handle other systems in the future.
- Provide input material for the formalization effort in WP1 (mPlane architecture). Systems enumerated in this document (more precisely their input and output formats, configuration options and specifications, and most importantly measurement and data representation) need to be considered by the architecture work package when formalizing these concepts within mPlane, a general purpose measurement framework
- Finally, this document introduces the concept of *proxy probes* within mPlane as the main vehicle for interfacing with existing systems within the scope of WP2. Proxy probes operate as ``translators'' between legacy and deployed systems and mPlane's control and data acquisition mechanisms.

1 Introduction

The mPlane architecture, as outlined in D1.1, introduced three principle classes of components:

- probes: measurement instances
- repositories: instances that correlate, store, and analyze measurment data
- a supervisor: intelligent controlling instance

The mPlane architecture derived from these components is an interface-centric model of interactions amongst them. The data and control flows between the mPlane components is depicted in Fig. **??**.

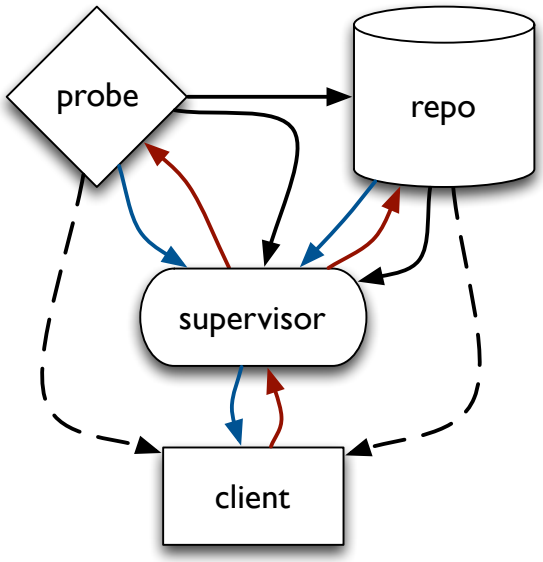


Figure 1: General schema of data and control flows in mPlane (data in black, control in red, capability in blue; uncommon data flows dashed)

This general approach allows an integration of external measurement results and data sets and probes using the exact same meachanisms, provided that the necessary - syntactic and semantic - translation takes place between the external data and mPlane. Taking advantage of legacy monitoring solutions and existing infrastructure is essential for quickly growing the coverage and reach of mPlane. Interfacing to existing monitoring solutions and their collected data is achived in two different ways in mPlane:

a) legacy systems that store their results in a centralized data storage allowing bulk operations (e.g. a measurement database containing results from multiple sources)

b) exising systems that can be classified as *probes* using the mPlane architecture (see Deliverable 1.1 chapter 5) are interfaced using the *proxy* concept

From the above list of mPlane component classes, the repository is responsible for interfacing with external bulk data sources of the kind a). The mPlane architecture defines the following capabilities for such repositories:

- allow storage and retrieval of measurement data received from probes or other repositories
- provide retrieval of externally sourced data
- perform analyses on stored or collected measurement data
- merge/fuse data from multiple sources.

There is a broad set of measurement data available in existing (pre-mPlane) repositories that can (and should) be utilized by mPlane, as repeating measurement data collection campaigns is not feasible nor reasonable. Thus, interfacing with external data source is an important task for the mPlane Work Package focusing in repositories (WP3). This interfacing is carried out by WP3 in Task 3.3 (*"Access to Analytic and External Data"*).

However, it is also necessary to interface with external systems of kind b) ("external probes"), because in many cases the data sought cannot be drawn from external repositories for various reasons such as:

- the data is stale for a given measurement (e.g. average latency to a particular host or subnet)
- there's no data for a particular target (path asymetry in a certain autonomous system)
- repositories might aggregate data received from probes (e.g. only provide averages) which is not what is needed

In these and similar situations probes need to be contacted directly to perform measurements, if they allow it (e.g. through re-configuration) or specific measurement results can be read from external probes directly instead of going through a repository. The conceptual component that is responsible for interfacing with external probes and "translating" such probe data is the proxy probe, and this document focuses on this concept.

2 Proxy Probes within mPlane

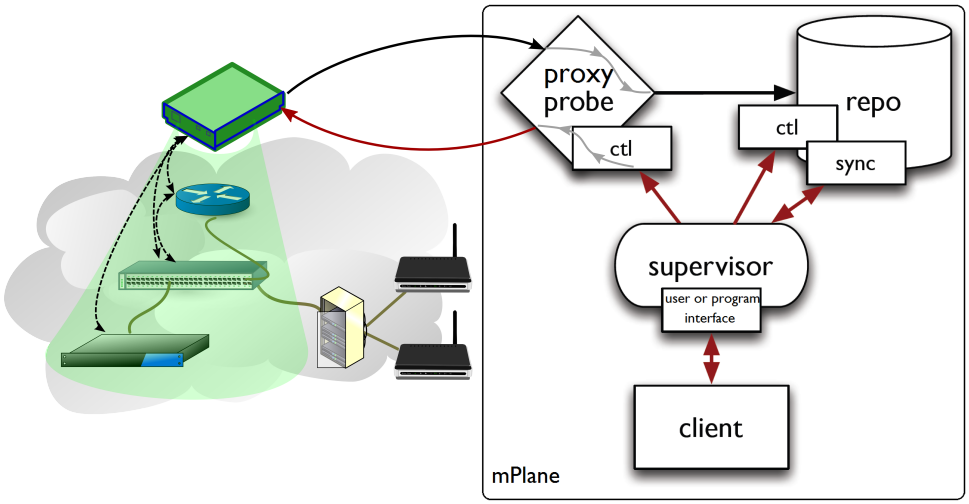


Figure 2: Proxying and external probe in mPlane (control in red, data in black, internal proxy probe processing in grey, measurement coverage in green)

A proxy probe is a special module implemented for the mPlane platform which acts as a "translator" between the external system and mPlane: mPlane control commands are translated into specific configuration instructions (if the external systems permits re-configuration), and, more importantly, measurement data is read from the external system and translated into a format usable by mPlane. For example, a proxy module would allow the mPlane Supervisor to run RTT measurements from Planetlab nodes distributed across the world, or to retrieve RTT measurements already available from other projects, e.g., iPlane.

2.1 Proxy Probe Functionality

Architecturally speaking, the proxy probes aims at delivering the following functionality:

Abstraction: the proxy probe obtains the data (including the capabilities, configuration and spatial/-topological location as much as possible) of the legacy probes it is proxying. The main architectural consideration consists in defining the actual representation of this information by mPlane's (Abstract) Data Types and Capabilities (to be defined in D1.3 mPlane Architectural Specification, see D1.1 chap. 5 for a preliminary discussion).

Syntactic translation: recognizing that many probes exist in the field nowadays and more diversity is to be expected in the future, performing syntactic translation of the exchanges between the probes and their clients (e.g. supervisors) is a fundamental functionality. The reasons to keep a $1 : n$ relationship between mPlane language and the native probe languages are:

i) it enables to keep a generic and unique language between the different mPlane modules that would remain unaffected by the addition or the removal of new probes (assuming the corresponding words/expressions are already covered by the syntactic rules of the mPlane language),

ii) it prevents introducing race conditions/concurrency between different instructions that would not be directly detectable if all probes would be driven by different languages processed independently (this role is also associated to the scheduling role described here below),

iii) the third reason is more practical in the sense that whereas the project aims at defining a probe language many probes would still be made in the future without being fully compliant in their first phase

Control/coordination: in order to ensure consistency of the measurement data, either the control logic of the external system must strictly follow the control logic of the mPlane platform, or the translation of results needs to take external control into account (e.g. for external systems controlled independently of mPlane, but willing to share measurement results). As both mPlane and external systems are expected to grow organically, the problem can be schematized as follows: either the supervisor keeps a master relationship with each probe through direct control enforced by the proxy, or the proxy probes translate and present measurements and capabilities data towards mPlane indicating non-controllability.

Additionally, as the types of external probes can be large and their number of probes of each type can itself be quite large, the number of parallel communication channels between the proxy probe and the probes may soon become a limiting factor of the scaling properties of the mPlane platform. For this purpose, as the proxy probe will act as a control (service interworking) gateway, it is also advisable to enable proxy probes to form a hierarchical structure. Such structure would be rooted at a few "main" proxy probes topologically close to the supervisor until the bottom level where many probes would have to be controlled (configured /parametrized and operated) while spatially local.

The measurement data gathered by the probes flowing upstream towards the supervisor or repositories will yield a bidirectional, tree-like structure (the control forms a proper tree rooted at the supervisor, while the reverse direction may also be rooted by the repositories). The depth of the tree would be in most cases limited to 2 or 3 levels with numerous leaves.

2.2 Proxy Probe Implementation

Note: in the following text, the terms *external*, *existing* and *legacy* systems are referring to the same concept: measurement systems that were developed before mPlane, (thus *legacy*), they obviously do not implement mPlane's interfaces (thus *external*) and yet, they provide important data mPlane needs to rely on (thus *existing*).

The concept of a proxy probe is illustrated in Fig. **??**. The green device symbolizes the existing measurement collector (*legacy probe*), the green area shows the network area it can access, and dashed lines symbolize some sort of measurements it can perform. For this discussion, it is irrelevant whether these are active or passive measurements.

A proxy probe, or more precisely a *proxy probe instance* is a software module (process), that is responsible for *proxying* one or more external probes towards mPlane. One proxy is implemented separately for each external system, and - depending on the technical pecularities of the external system and on the decisions of the proxy developer - one running proxy *instance* may represent one or many external probes (of the same kind). For example, one *tstat-proxy* instance would be run for each *tstat* system interfaced (because for technical reasons the tstat proxy has to reside on the same computer where tstat is running). On the other hand, one *CiscoPing* proxy instance may represent many CiscoPing agents scattered over the network, due to the way CiscoPing agents are configured and the measurement collected.

As shown in the figure, the mPlane *proxy probe* has to perform the following tasks:

1. fully implement mPlane's *probe interfaces*, thus it is able to communicate with the Supervisor, and upload data to the Repositories. Actually, the Supervisor is unable to tell whether the probe is communicating with is a proxy or a "real" probe, as this difference is not relevant thus not articulated in mPlane. The same holds for the probe $\rightarrow$ Repository communication.

   The proxy probe $\leftrightarrow$ mPlane communication:

   - describing the external system's capabilities towards mPlane, thus the Supervisor can learn about the probe's availability, metrics supported, accessible network segments, and all other characteristics and limitations of the system that is necessary to schedule measurements or interpret results.
   - (if the proxied system is configurable) accepting control from the Supervisor
   - making measurement results available to mPlane by implementing one or both of:
     - synhronous or asynchoronous results retrieval for mPlane clients (e.g. Supervisors, users)
     - assisting bulk results upload towards mPlane Repositories

2. implementing the external probe's *control interface* (if any), and mapping ("translating") the Supervisor's control messages (sent using mPlane's control protocol) into native control actions (shown by grey arrows in Fig. **??**)

3. implementing the external probe's *data interface* (always exist), and mapping ("translating" - shown by grey arrows in Fig. **??**) the results into:

   (a) mPlane result records using native mPlane data types and/or

   (b) result messages using mPlane result retrieval or bulk upload protocols

   (c) setting up (if supported) a direct external probe $\rightarrow$ mPlane Repository results upload

The very last action is possible because bulk Repository upload operations are not standardized by mPlane, thus it might be possible that a mPlane respository accepts the result upload protocol (esp. if it is a well-known standardized one like IPFIX) implemented by an existing probe. In this case the proxy probe is still necessary to orchestrate (set up) this data transfer, although the data is not actually passed though the proxy.

3 Selected Probes

In this chapter, a selection of existing systems (probes) is enumerated and related to mPlane. Each system is discussed in the following structure:

Overview: the first section describes the purpose, motivation, important concepts and capabilities of the system.

Configuration: this section describes the *input* for the existing system: how is it configured and what kind of data formats and standards are used.

Results: this section describes the *output* of the system, i.e. the kind(s) of measurement readings it support, and the ways to retrieve them.

Proxy Design: finally, this section relates the input and output descriptions above to mPlane concepts described in D1.1 (chapters 5. and 6.). A rudymentary design is sketched - as much as possible with the current state of mPlane interface specifications - for a *proxy probe* translating between mPlane components (Repository and Supervisor) and the external system.

The systems listed below were chosen so that the mPlane consortium has the necessary in-depth knowledge and reasonable resources for developing proxies for them. Notwithstanding this limitation, we still tried to select systems so that most major directions in probe development has at least one representative on the list, for example:

- passive sniffers, flow analyzers, DPI inspectors (*tstat, QoF, DATI*, including packet processing framework (*BlockMon*)
- active testers from simple ICMP queriers (*CiscoPing*) to protocol-level testers of known hosts (*CiscoIPSLA, MisuraInternet*), or client-side QoS analyzers (*Firelog, PyTomo*)
- path-analyzer and topology-discovery tools (*tracebox*)
- large-scale multi-probe and measurement orchestrators (*Scamper, MERLIN*)
- mobile network performance analyzers (*MobiPerf*)

3.1 QoF

3.1.1 Overview

QoF is a TCP-aware IPFIX flow meter that runs on UNIX systems, tailored to the measurement of flows for TCP performance analysis purposes on intermediate-size network links (1G - 10G network edge, depending on capture hardware).

3.1.2 Configuration

QoF observes packets, takes an IPFIX template definition and flow table configuration parameters, and exports IPFIX. The flow meter is presently under active development; in-progress documentation is presently available at `http://github.com/britram/qof/wiki`.

Most of the configuration parameters for QoF are internal to the operation of the flow meter's TCP performance measurement features, and will be configured statically per site. Dynamic configuration includes the destination for IPFIX export and the idle and active flow timeouts.

3.1.3 Results

QoF produces extended flow records in IPFIX: n-tuples described by IPFIX templates definable at runtime, using any combination of a set of about fifty information elements, including timestamps, the 5-tuple flow key, counters for packets and octets at layers 3 and 4, progress through sequence number space, RTT information, retransmission and out-of-order transmission information. The TCP-specific Information Elements allow QoF to be used to generate flow records relevant to TCP performance measurement.

3.1.4 Proxy Design

QoF's configuration language uses YAML. The selection of features to be activated at runtime in QoF is driven directly by the set of Information Elements which are selected to be exported. For example, latency measurement algorithms are activated only if RTT-relevant Information Elements are selected in the list of Information Elements apprearing in its configuration template definition.

This is a natural match for the proposed mPlane interfaces, as these also define measurements in terms of the data types produced.

The mPlane proxy therefore need only support three operations:

1. Advertisement of Capabilities to create flows according to a set of profiles (templates) given parameters (IPFIX export target, flow timeouts).
2. Extraction of template Information Elements (including mPlane Element to IPFIX Information Element translation), timeout values, and export target information from Specifications, leading to an invocation of QoF which can be controlled with information in the associated Receipt.
3. Control of a running QoF instance by extraction of information from a cancellation Specification derived from this Receipt.

3.2 Cisco PING Agents

3.2.1 Overview

Cisco devices offer an active probing capability to measure IPv4 availability using ICMP echo messages. The device sends a preconfigured number of ping messages and calculates round-trip-time (RTT) statistics based on the received packets. Cisco devices are widely used, so if mPlane has a proxy probe for Cisco PING measurements then the reasoner can use it for example to measure availability and RTT on an on-demand basis to detect, and possibly isolate a network problem.

The main limitation of Cisco PING is that it cannot run measurements constantly, because each test has to be configured separately.

3.2.2 Configuration

The measurements can be configured using SNMP and the results can also be queried using SNMP. The configuration parameters supported/handled by the proxy are:

- SerialNumber: this is the unique identifier which identifies the measurement
- Protocol: the protocol for the measurement
- Address: the remote address which the device has to ping
- PacketCount: number of ping packets
- PacketSize: size of the packets
- PacketTimeout: the timeout of each ping packets in msec
- Delay: the minimum amount of time to wait before sending the next packet
- VrfName: the name of the VPN (optional parameter)

The proxy will automatically assign a new serial number for each test and send back a receipt to the requester with the serial number (among other parameters). The proxy will only support IPv4 ping measurements, so that although the requester will be able to specify the protocol, initially only the IPv4 will be supported.

3.2.3 Results

The proxy will be able to send back the measurement results for a particular receipt, but only once the whole test has finished. The returned parameters are:

- SentPackets: the number of packets sent out
- ReceivedPackets: the number of result packets received
- Availability: the availability in percentage, calculated as 100 * ReceivedPackets / SentPackets

- MinRTT: the calculated minimum RTT, if at least one packet was received
- AvgRTT: the calculated average RTT, if at least one packet was received
- MaxRtt: the calculated maximum RTT, if at least one packet was received

3.2.4 Proxy Design

The proxy will work in the "Delayed data access" mode, see deliverable D1.1, Figure 6.3 (c)

The proxy will be able to handle more than just one Cisco device. This means that the proxy will advertise the management IP addresses and names of those Cisco devices which it can use to run the measurements. The proxy will also advertise for each device the available protocol, however as mentioned above, initially the proxy will only advertise the IPv4 protocol.

The proxy will receive specifications for each measurements with the specified parameters mentioned in the Configuration section, except

- the IP address of the Cisco device has to be specified.
- it will not require the serial number to be specified. If it is filled in then it will try to use that serial number, otherwise it will automatically look for an unused number

If the proxy knows the specified Cisco device and it can configure the required measurement then it sends back a receipt.

The proxy will also receive receipt requests and send back the results if the specified measurement exists on the specified Cisco device and it has finished the measurements.

3.3 Cisco IP SLA Agents

3.3.1 Overview

Some Cisco devices have an active probing agent with capabilities beyond simple IPv4 availability measurements. This agent, called Cisco IP SLA (formerly known as Cisco SAA) can perform different tests, but the tests available on a particular device depends on the device type and its IOS version. The tests are performed periodically until a given time or indefinitely and usually they measure success rate and response time parameters. Some of the available measurements are:

- dhcp: the agent tries to get an IP address from a designated DHCP server
- dns: the agent queries from a designated DNS server an IP address or a hostname
- ipIcmpEcho: the agent sends periodically IPv4 ICMP echo messages to an IP address
- ftp: the agent uses FTP GET operation to download a file specified by an URL either in passive or in active mode
- http: the agent uses HTTP GET or RAW operation to download a file specified by an URL and measures success rate and response time also separately for DNS, TCP connection and HTTP operation
- jitter: the agent sends UDP packets to a destination and measures also the source->destination and destination->source jitter values
- pathEcho: the agent discovers the path to a given IP address using the traceroute facility and then measures hop-by-hop response time along that path using ICMP ECHO. Obviously the path can change between tests so the agent is able to record a certain number of previously seen paths
- pathJitter: this test is similar to the pathEcho, except that in this case the agent uses the ICMP ECHO messages to approximate the two-way jitter values for each hop
- tcpConnect: the agent opens a TCP connection using the given IP address and port. Once the connection is established the agent can end the test but it can also perform a TCP echo operation
- udpEcho: the agent sends an UDP echo message to the given given IP address and port

Cisco devices are widely used, so if mPlane has a proxy probe for Cisco IP SLA measurements then the reasoner can use it for example to measure availability and RTT using different protocols, measure jitter values or determine the path to a given IP address. In this section we choose the pathEcho test to demonstrate the interface between the mPlane proxy and an IP SLA capable device.

3.3.2 Configuration

The pathEcho measurements can be configured using SNMP and the results can also be queried using SNMP. The configuration parameters supported/handled by the proxy:

- EntryNumber: this is the unique identifier which identifies the measurement
- Type: the type of the measurement, in this case this is fixed to pathEcho

- DestinationAddress: the remote IP address which the device has to ping
- SourceAddress: the optional source IP address for the operation
- Frequency: number of ping packets
- RequestDataSize: size of the packets
- Timeout: the timeout of each ping packets in msec
- VrfName: the name of the VPN (optional parameter)

The proxy will automatically assign a new entry number for each test and send back a receipt to the requester with the serial number (among other parameters).

3.3.3 Results

The proxy will send the results periodically to a designated mPlane repository in an asynchronous mode. The results will contain one or more paths which were detected since the last result was sent and for each path it will contain the one or more hops. The returned parameters for each hop:

- Address: the IP address of the hop
- CompletedTests: the number of tests completed
- SuccessfulTests: the number of successful tests
- Availability: the availability in percentage, calculated as 100 * SuccessfulTests / CompletedTests
- AvgRTT: the calculated average RTT, if at least one packet was received

3.3.4 Proxy Design

The proxy will be able to handle more than just one Cisco device. This means that the proxy will advertise the management IP addresses and names of those Cisco devices which it can use to run the measurements. The proxy will also advertise for each device the available test types, however as mentioned above, initially the proxy will only advertise the pathEcho test type.

The proxy will receive specifications for each measurements with the specified parameters mentioned in the Configuration section, except

- it will not require the entry number to be specified. If it is filled in then it will try to use that serial number, otherwise it will automatically look for an unused number
- Additional parameters needed for the configuration:
  - ConfigurationType: an enumeration type, either Create or Delete: if it is Create then the proxy will configure the IP SLA operation, otherwise it will delete the operation
  - RepositoryAddress: the IP address of the mPlane repository where the probe will send the results of the operation

- CollectionInterval: how often should the probe send the results, in seconds
- the IP address of the Cisco device has to be specified.

If the proxy knows the specified Cisco device and it can configure the required measurement then it sends back a receipt.

3.4 Tracebox

3.4.1 Overview

The TCP/IP architecture was designed to follow the end-to-end principle. A network is assumed to contain hosts implementing the transport and application protocols, routers implementing the network layer and processing packets, switches operating in the datalink layer, etc. This textbook description does not apply anymore to a wide range of networks. Entreprise networks, WiFi hotspots, and cellular networks often include various types of middleboxes in addition to traditional routers and switches [**?**]. A *middlebox*, defined as ``any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host'' [**?**], manipulates traffic for purposes other than simple packet forwarding. Middleboxes are often deployed for performance or security reasons. Typical middleboxes include Network Address Translators, firewalls, Deep Packet Inspection boxes, transparent proxies, Intrusion Prevention/Detection Systems, WAN optimizers, etc.

Recent papers have shed light on the deployment of those middleboxes. For instance, Sherry et al. [**?**] obtained configurations from 57 entreprise networks and revealed that they can contain as many middleboxes as routers. Wang et al. [**?**] surveyed 107 cellular networks and found that 82 of them used NATs. Although these middleboxes are supposed to be transparent to the end-user, experience shows that they have a negative impact on the evolvability of the TCP/IP protocol suite [**?**]. For example, after more that ten years of existence, SCTP [**?**] is still not widely deployed, partially because many firewalls and NAT may consider SCTP as an unknown protocol and block the corresponding packets. Middleboxes have also heavily influenced the design of Multipath TCP [**?**, **?**].

Despite of their growing importance in handling operational traffic, middleboxes are notoriously difficult and complex to manage [**?**]. One of the causes of this complexity is the lack of debugging tools that enable operators to understand where and how middleboxes interfere with packets. Many operators rely on `ping`, `traceroute`, and various types of `show` commands to monitor their networks.

`tracebox` is a `traceroute` [**?**] successor that enables network operators to detect which middleboxes modify packets on almost any path. `tracebox` allows one to identify various types of packet modifications and can be used to pinpoint where a given modification takes place.

`tracebox` is similar to `traceroute` but also very different at the same time. To detect middleboxes, `tracebox` uses the same incremental approach as `traceroute`, i.e., sending probes with increasing TTL values and waiting for ICMP `time-exceeded` replies. While `traceroute` uses this information to detect intermediate routers, `tracebox` uses it to infer the modification applied on a probe by an intermediate middlebox.

`tracebox` brings two important features.

- **Middleboxes Detection**. `tracebox` allows one to easily and precisely control all probe packets sent (IP header, TCP or UDP header, TCP options, payload, etc.). Further, `tracebox` keeps track of each transmitted packet. This permits to compare the quoted packet sent back, in an ICMP `time-exceeded` by an intermediate router, with the original one. By correlating the different modifications, `tracebox` is able to infer the presence of middleboxes.
- **Middleboxes Location**. Using an iterative technique (in the fashion of `traceroute`) to discover middleboxes also allows `tracebox` to approximately locate, on the path, where modifications occurred and so the approximate position of middleboxes.

When an IPv4 router receives an IPv4 packet whose TTL is going to expire, it returns an ICMPv4 `time-exceeded` packet that contains the offending packet. According to RFC792, the returned ICMP packet should quote the IP header of the original packet and the first 64 bits of the payload of this packet [**?**]. When the packet contains a TCP segment, these first 64 bits correspond to the source and destination ports and the sequence number. RFC1812 [**?**] recommended to quote the entire IP packet in the returned ICMP, but this recommendation has only been recently implemented on several major vendors' routers. Discussions with network operators showed that recent routers from Cisco (running IOX), Alcatel Lucent, HP, Linux, and PaloAlto firewalls return the full IP packet. In the remainder of this section, the term *Full ICMP* is used to indicate an ICMP message quoting the entire IP packet. The term RFC1812-compliant router is used to indicate a router that returns a *Full ICMP*.

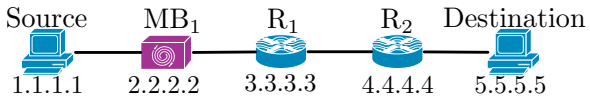
By analyzing the returned quoted packet, `tracebox` is able to detect various modifications performed by middleboxes and routers. This includes changes in the Differentiated Service field and/or the Explicit Congestion Notification bits in the IP header, changes in the IP identification field, packet fragmentation, and changes in the TCP sequence numbers. Further, when `tracebox` receives a *Full ICMP*, it is able to detect more changes such as the TCP acknowledgement number, TCP window, removal/addition of TCP options, payload modifications, etc.

`tracebox` also allows for more complex probing techniques requiring multiple probes to be sent, e.g., to detect segment coalescing/splitting, Application-level Gateways, etc. In this case `tracebox` works in two phases: the *detection* and the *probing* phases. During the detection phase, `tracebox` sends probes by iteratively increasing the TTL until it reaches the destination. This phase allows `tracebox` to identify RFC1812-compliant routers. During the probing phase, `tracebox` sends additional probes with TTL values corresponding to the previously discovered RFC1812-compliant routers. This strategy allows `tracebox` to reduce its overhead by limiting the number of probes sent.

`tracebox` can also be used to detect TCP proxies. To be able to detect a TCP proxy, `tracebox` must be able to send TCP segments that are intercepted by the proxy and other packets that are forwarded beyond it. HTTP proxies are frequently used in cellular and enterprise networks. Some of them are configured to transparently proxy all TCP connections on port 80. To test the ability of detecting proxies with `tracebox`, a script that sends a SYN probe to port 80 was used and, then, to port 21. If there is an HTTP proxy on the path, it should intercept the SYN probe on port 80 while ignoring the SYN on port 21. After that, the returned ICMP messages are analyzed.

NATs are probably the most widely deployed middleboxes. Detecting them by using `tracebox` would likely be useful for network operators. However, in addition to changing addresses and port numbers of the packets that they forward, NATs also change back the returned ICMP message and the quoted packet. This implies that when inspecting the received ICMP message, `tracebox` would not be able to detect a visible change.

This does not prevent `tracebox` from detecting many NATs. Indeed, most NATs implement *Application-level Gateways* (ALGs) [**?**] for protocols such as FTP. Such an ALG modifies the payload of forwarded packets that contain the `PORT` command on the `ftp-control` connection. `tracebox` can detect these ALGs by noting that they do not translate the quoted packet in the returned ICMP messages. Detecting such middleboxes works by sending a SYN for the FTP port number and, then, waiting for the SYN+ACK. After that, a valid segment with the `PORT` command and the encoded IP address and port number as payload is sent. `tracebox` then compares the transmitted packet with the quoted packet returned inside an ICMP message by an RFC1812-compliant router. If the payload has changed, it means there exists an ALG on the path.



(a) topology

```
# tracebox -o mss -n 5.5.5.5
tracebox to 5.5.5.5 (5.5.5.5): 30 hops max
1: 2.2.2.2 [Reply ICMP full pkt]
2: 3.3.3.3 [TCP seq changed]
3: 4.4.4.4 [TCP opt removed/changed] [Reply ICMP full pkt]
4: 5.5.5.5
```

(b) output

Figure 3: `tracebox` example

3.4.2 Configuration

Fig. **??** shows a simple network, where MB_1 is a middlebox that changes the TCP sequence number and the MSS size in the TCP MSS option. R_1 is an old router while R_2 is a RFC1812-compliant router. The output of running `tracebox` between ``Source'' and ``Destination'' is given by Fig. **??**. The output shows that `tracebox` is able to effectively detect the middlebox interference but it may occur at a downstream hop. Indeed, as R_1 does not reply with a *Full ICMP*, `tracebox` can only detect the TCP sequence change when analyzing the reply of R_1. Nevertheless, when receiving the *Full ICMP* message from R_2, that contains the complete IP and TCP header, `tracebox` is able to detect that a TCP option has been changed upstream of R_2.

`tracebox` is implemented in C in about 2,000 lines of code and provides python and LUA [**?**] bindings to ease the development of more complex measurement scripts. More that ten python scripts are currently available which allow to detect various types of middleboxes from Application-level Gateways to HTTP proxies and fragments reassembly middleboxes.

`tracebox` is a command-line tool that can be used as followed:

```
sudo tracebox [ -hn ] [ OPTIONS ] host
```

`host` is the target of `tracebox`. This is mandatory. Options are:

- `-h` displays the help and exit;
- `-n` does not resolve IP addresses;
- `-i device` specifies a network interface to work with;
- `-m hops_max` sets the maximum number of hops (maximum TTL to be reached). Default is 30;
- `-o option` defines the TCP option to put in the SYN segment. Default is none. Available options are mptcp, mss, wscale, ts, sack;
- `-O file` uses `file` to dump the sent and received packets;
- `-p port` specifies the destination port to use when generating probes. Default is 80;
- `-f flag1[,flag2[,flag3...]]` specifies the TCP flags to use. Values are: syn ack, fin, rst, push, urg, ece, cwr. Default is syn;
- `-M mss` specifies the MSS to use when generating the TCP MSS option. Default is 9140.

3.4.3 Results

Output provided by `tracebox` is given in Fig. **??**. This is a standard text output, quite similar to the one provided by standard `traceroute`.

In addition, packets sent/received by `tracebox` can be saved in binary format (`pcap`) for off-line analysis.

3.4.4 Proxy Design

The proxy will receive a file containing a list of target IP addresses (possibly, the file may contain a single target), with one target per line. In addition, the proxy will receive a configuration file for setting up `tracebox`. Elements of interest for this configuration file are the TCP option to put in the SYN segment (should $\in \{$mptcp, mss, wscale, ts, sack$\}$), the name of the dump file, the list of TCP flags to use (should $\in \{$syn ack, fin, rst, push, urg, ece, cwr$\}$) and, optionnally, the MSS to use when generating the TCP MSS option.

3.5 Scamper

3.5.1 Overview

`scamper` is a parallelised packet-prober capable of large-scale Internet measurement using many different measurement techniques. Briefly, `scamper` obtains a sequence of measurement tasks from the input sources and probes each in parallel as needed to meet a packets-per-second rate specified on the command line. Tasks currently being probed are held centrally by `scamper` in a set of queues the probe queue if the task is ready to probe, the wait queue if it is waiting for time to elapse, and the done queue if the task has completed and is ready to be written out to disk. Each measurement technique is implemented in a separate module that includes the logic for conducting the measurement as well as the input/output routines for reading and writing measurement results, allowing measurement techniques to be implemented independently of each other. When a new measurement task is instantiated, the task attaches a set of callback routines to itself that `scamper` then uses to direct the measurement as events occur, such as when it is time to probe, when a response is received, or when a time-out elapses. Sockets required as part of a measurement are held centrally by `scamper` in order to share them amongst tasks where possible so that resource requirements are reduced. Finally, `scamper` centrally maintains a collection of output files where completed measurements are written.

`scamper` comes with `traceroute` utilities. The `traceroute` included in `scamper` is feature-rich: it supports IPv4 and IPv6; probe methods based on UDP, ICMP, and TCP, including Paris traceroute [**?**], path MTU discovery (PMTUD) to infer the presence of tunnels [**?**]; a method to infer the hops in a path that do not send an ICMP packet too big message which is required for PMTUD to work [**?**]; and Doubletree [**?**] to reduce redundant probing.

`ping` is useful to measure end-to-end delay and loss, search for responsive IP addresses, and classify the behaviour of hosts by examining how they respond to probes. In addition to the traditional ICMP echo method, `scamper` supports UDP, TCP, and TTL-limited probing, which can be used if directed ICMP echo probes do not obtain a response. `scamper` includes the ability to spoof the source address of probes, as well as include IP options for record route and timestamps. These features are useful for implementing reverse traceroute [**?**].

`scamper` implements a multipath detection algorithm [**?**] to infer all interfaces visited between a source and destination in a per-flow load- balanced Internet path. It does this by deliberately varying the flow-identifier that a router may compute when load balancing. Probes with different flow-identifiers may take different paths and thus reveal different parts of the forward IP path. In addition to the ICMP and UDP methods originally implemented by Augustin et. al that vary the ICMP checksum and UDP destination port values, `scamper` implements a UDP method that varies the source port instead of the destination port so that the probes do not appear to be a port scan. This method also provides the ability to probe past a firewall that blocks UDP probes to ports above the usual range used by traceroute [**?**]. `scamper` also implements TCP methods that vary the flow-id by changing the source or destination port, depending on the user's choice.

`scamper` implements four techniques for inferring which IP addresses observed in the Internet topology are aliases:

1. Mercator probing [**?**];
2. Ally probing [**?**];
3. RadarGun probing [**?**];
4. Prefix scan probing. This method infers a router's outging interface by findin an alias in the same subnet as the next interface in the forward path.

`scamper` implements Savage's [**?**] TCP-based algorithm to infer one-way loss by making use of algorithms used by TCP receivers when they receive out-of-sequence packets.

Finally, `scamper` implements two of the techniques described in [**?**]: measurement of behaviour in response to an ICMP packet too big message, and measurement of behaviour in response to ECN negotiation and notification.

Note that `tracebox` is expected to being implemented in `scamper` in the very near future.

3.5.2 Configuration

`scamper` is a command-line tool that can be launched as follows:

```
scamper -c -p -w -M -o -O -f
```

Options are the following:

- `-c command` specifies the command to use for `scamper`. Possible choices are delias, neighbourdisc, ping, trace, tracelb, sniff, sting, and tbit (see Sec. **??** for details);
- `-p pps` specifies the target packets-per-second rate for `scamper` to reach. By default, this value is 20;
- `-w window` specifies the maximum number of tasks that may be probed in parallel. A value of zero places no upper limit. By default, zero is used;
- `-M monitorname` specifies the canonical name of machine where `scamper` is run. This value is used when recording the output in a `warts` output file;

- `-o outputfile` specifies the default output file to write measurement results to. By default, `stdout` is used;
- `-O options` allows `scamper`'s behaviour to be further tailored. The only relevant choice here (for mplane) is `warts` (it ouputs results to a `warts` binary file);
- `-f listfile` specifies the input file to read for target addresses, one per line, and uses the command specified with the -c option on each.

3.5.3 Results

`scamper` provides two output file formats:

1. ASCII text option;
2. binary file format know as `warts`.

The text option produces low-fidelity output similar to `ping` and `traceroute` and is suitable for interactive use. The binary option is an extensible format designed for use by researchers because of its ability to record detail and provide archival features.`scamper` includes a library that allows its binary output files to be easily read, and CAIDA have created a Ruby library [**?**] allowing researchers to develop analysis programs in Ruby. `scamper` supplies an API to assist a researcher implementing a new measurement technique to create a record which can then be stored in the binary file format. To promote precision and discourage researchers from recording results in the text option, the library provides no ability to read results from a text file.

3.5.4 Proxy Design

The proxy will receive a file containing a list of target IP addresses (possibly, the file may contain a single target), with one target per line. In addition, the proxy will receive a configuration file for setting up `scamper`. Elements of interest for this configuration file are the command to use for `scamper` (should $\in \{$delias, neighbourdisc, ping, trace, tracelb, sniff, sting, tbit$\}$) and, finally, the name of the output file.

3.6 MERLIN

3.6.1 Overview

Recently, `mrinfo`, a multicast management tool, has been used for topology discovery [**?**, **?**]. `mrinfo` comes with the strong advantage of listing all multicast interfaces of a router and its multicast links towards others using a single probe. `mrinfo` offers, by design, a router-level view of the topology: it does not suffer from the same shortcomings resulting from combining traceroute and alias resolution techniques. However, its view is limited to multicast components and, in the same way that ICMP messages may be rate limited or filtered for traceroute probing, IGMP messages can be filtered by some ISPs [**?**].

Here, we discuss a novel client/server platform called MERLIN (MEasure the Router Level of the INternet) efficiently mixing three active probing tools: IGMP probing using an improved version of `mrinfo` [**?**],

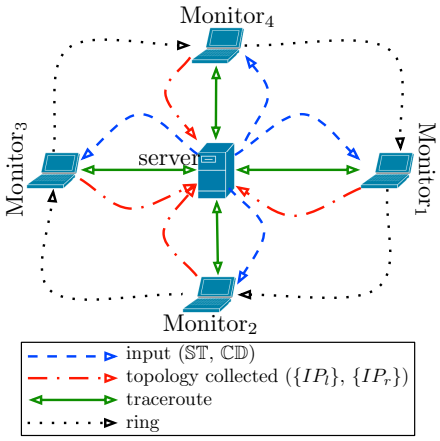


Figure 4: MERLIN global overview

ICMP probing with Paris Traceroute [**?**], and alias resolution with *Ally* [**?**]. MERLIN makes use of IGMP probes to natively discover ISPs at the router level, while Paris Traceroute and Ally are used to overcome `mrinfo` limitations (i.e., filtering). The proposed methodology does not depend on those two last specific tools, the platform can be deployed with any other probing mechanisms.

3.6.2 Configuration

3.6.2.1 General Overview

MERLIN is based on a centralized client-server architecture and runs on Unix platforms. A central server, written in Python and C++, controls the vantage points, called *monitors*, written in C. The MERLIN monitors are potentially spread all over the world and logically organized as a ring, as illustrated in Fig. **??**. Only one monitor actively probes the targeted domain at a given time. This probing period by a single monitor is called *run*. There are as many runs as monitors in the system per probing *cycle*. A *cycle* is completed when all monitors have probed the targeted domain during a given round around the monitor ring. Note that, for a given targeted AS, several cycles can be required per IGMP probing *stage*, i.e., the number of cycles required to take advantage of all current seeds including those recursively found during previous runs. An IGMP probing stage might be made of an incomplete number of cycles: a stage stops when no more topological data can be collected using the current seed set. In order to increase the coverage of an AS probing *campaign*, between each stage, MERLIN launches internal Paris traceroute campaigns to collect new fresh seeds and better understand the topology lacks. Thus, for a given AS, a complete MERLIN campaign consists of several stages of IGMP probing (made of several cycles) seeded by external measurements and internal `traceroute` campaigns.

The input sent by the server to each monitor is composed of a list of IP addresses called *CurrentSeed* (CD in Fig. **??** - see Sec. **??** for its exact description). This set may contain seeds coming from several kinds of input. At the first stage, the seed list is initially built by the server from various external dataset while, for subsequent stages, the list is initially and dynamically built by the server based on intermediate internal `traceroute` measurements. During each stage the current seed set is recursively updated using neighbor IP collected by each monitor (the set $\{IP_r\}$ in Fig. **??**). The initial seeds list used during the

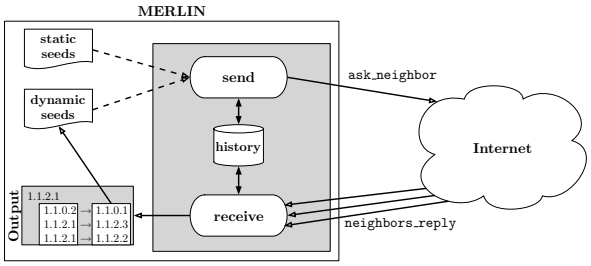


Figure 5: MERLIN monitor architecture

first stage is based on three inputs:

- Previously collected `mrinfo-rec` data is used as part of the seeds.
- ``External''[1] `traceroute` data collected by Archipelago [**?**].
- ``Initial''[1] `traceroute` campaign. As already stated by Spring et al. [**?**], it is possible to extract on an AS basis a set of *reachable prefixes*. Those prefixes are used as destinations for the initial internal traceroute campaign. This campaign is launched from all vantage points supporting the Paris traceroute tool.

Each input list is subject to a standard IP-to-AS mapping filtering process so that only IP addresses mapped to the targeted AS are used. Each monitor probes the network recursively (see Sec. **??** for details). For ``controlling'' this recursion and then avoiding *inter-monitor* redundancy, the server computes a list of IP addresses, called the *Stop Set* (ST in Fig. **??** - see Sec. **??**). This list contains IP addresses that have been previously discovered by MERLIN monitors and, thus, do not have to be probed again. The purpose is to avoid redundancy at two levels: *intra-monitor* (i.e., a given monitor probing several times the same router - see Sec. **??** for details) and *inter-monitor* (i.e., a monitor re-probes a router already discovered by another monitor). Initially (i.e., for the first monitor in the first stage), the Stop Set is empty. Then, it is cumulatively populated based on the topological data brought back to the server: each local IP address of a discovered router ($\{IP_l\}$ in Fig. **??**) is added to the Stop Set so that the next monitor in the ring will not probe it again.

3.6.2.2 MERLIN Monitor

Fig. **??** depicts the architecture of the MERLIN monitor. The monitor is composed of two parallel processes: *send*, in charge of sending probes to the network, and *receive*, in charge of processing the replies returned back by the routers. To minimize redundancy, the sending process never probes an IP address previously discovered: the ``history'' box in Fig. **??** is initialized with the Stop Set and is maintained up-to-date by the monitor, avoiding thus intra- and inter-monitor redundancy.

The *send* process is fed by both a static IP address list and a dynamic IP address list collected from replies. This dynamic list is used for recursion. When the monitor starts, the *send* process operates in a static mode using so IP addresses from the static list. Once replies are collected from the *receive* process, the dynamic list is built based on publicly routable neighbor IP addresses: the recursion is engaged. The

[1] Here the terms external and internal respectively describe the use of data previously collected by others or through our own probing campaigns.

send process enters in the dynamic mode and gives priority to targets of the dynamic list. Each time the dynamic list becomes empty (i.e., the current recursion is finished), the *send* process is again fed with remaining IP addresses from the static list. This recursion first scheme was a design choice that has been made in order to minimize the probability of reprobing a given router.

During the dynamic mode, the probe inter-departure time is fixed to a given value α in the order of a second. In contrast, during the static mode, the inter-departure parameter is fixed to a lower value β with $\beta << \alpha$ in order to fasten the probing. The success rate of the static mode being much more lower than the one of the dynamic mode, the reprobing risk is then really lower. To summarize, the *send* process prioritizes its tasks as follows: (1) if a new router has been discovered, it marks all its local IP interfaces as already seen, (2) if there are neighbor IP addresses to probe, it elapses the probing with the timer α, (3) otherwise it uses the static list and elapses probes with the timer β. Finally, note that the recursion stops at routers that have no interfaces mapped to the AS of interest thanks to the use of a *patricia tree* populated by the server.

3.6.2.3 MERLIN Server

The main server task is to keep track of the entire probing process by collecting the reassembled replies coming from each monitor and avoiding the injection of useless probes: while probing the same destination from different vantage points using `traceroute` is likely to give new information, once a router has responded to an IGMP query from a vantage point, it is useless to query it from another one (it is also necessary to stop the monitor recursion when discovering an already known router. Between two monitor probing runs, the server computes a new list of $CurrentSeed$ and its associated Stop Set based on the union of the already collected data. These two lists are then sent to the next monitor in order to increase the probing coverage while remaining as network friendly as possible: MERLIN minimizes the inter-monitor probing redundancy. Indeed, a MERLIN client stops its recursive probing exploration at an IP address belonging to the Stop Set. The IP interface list of a given router returned by `mrinfo` may be incomplete such that, despite efforts to avoid redundancy, MERLIN may probe the same router several times. Fortunately, this risk does not imply the reprobing of an entire connected component: the monitor stops its recursion at the first duplicated probe thanks to the $StopSet$ containing all the IP interfaces belonging to already collected routers.

In order to improve the global coverage and offer a better view of the probed network, MERLIN also launches dynamic `traceroute` campaigns when the list of seeds given as input to a monitor becomes empty. In practice, the end of an IGMP probing stage occurs when all current seeds respond or have been probed by all monitors. Those inter-stage `traceroute` campaigns are performed using a specific hitlist of IP addresses. For this purpose, the server constructs a set called $BorderIP$ consisting of the IP addresses located at the ``border'' of connected components. Those addresses correspond to the ones that have been collected as neighbor IPs but that do not reply directly to MERLIN. This list is then used by all monitors to select destinations for inter-stage `traceroute` campaigns. The objective is twofold: produce new seeds and improve the data returned by MERLIN replies. It allows for obtaining missing unicast information.

A current view of the $BorderIP$ set is maintained between each run to take benefit of the recursion across monitors. As long as new potential seeds are discovered (recursively -- *intra stage* -- and based on internal `traceroute` measurements -- *inter stage*), the server continues to ``feed'' monitors. The server sequentially sends to monitors an updated list of IP addresses corresponding to new seeds to probe: $CurrentSeed$. This set consists of IP addresses not responding to other monitors, new neighbor IP addresses discovered during previous runs, or IP addresses collected through Paris traceroute campaign

between two IGMP probing stages. However, the $CurrentSeed$ set excludes interfaces belonging to already collected routers and also the ones that have been already probed without success by the next monitor in the ring.

When the discovery probing phase of MERLIN stops, MERLIN enters in a post-processing phase dedicated to the router level graph refinement. After having removed potential duplicates due to pure unicast IP addresses[2] and applied a router-to-AS election [**?**] focusing on routers belonging to the targeted AS, MERLIN performs a recursive hybrid reconnection mechanism based on `traceroute` and alias resolution (able to keep the native router level view of MERLIN) for merging isolated IGMP components into a larger one

3.6.3 Results

Output provided by MERLIN is a standard text output. Here is an example of MERLIN output for a given router in AS137:

```
193.206.130.6 |137| - 58 - (15.0) - rx1-na1-ru-unina-l1.na1.garr.net {42.8333;12.8333}:
 143.225.190.229 -> 143.225.190.230 [1/0/p] - (- -> -) - |137 -> 137|
 143.225.190.58 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |137 -> 0|
 143.225.190.17 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |137 -> 0|
 143.225.190.193 -> 143.225.190.194 [1/0/p] - (- -> -) - |137 -> 137|
 143.225.190.1 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |137 -> 0|
 143.225.190.93 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |137 -> 0|
 193.206.130.10 -> 193.206.130.9 [1/0/p/q] - (- -> -) - |137 -> 137|
 0.0.0.0 -> 0.0.0.0 [1/0/p/d/o/l] - (- -> -) - |0 -> 0|
 0.0.0.0 -> 0.0.0.0 [1/0/p/d/o/l] - (- -> -) - |0 -> 0|
 193.206.130.6 -> 193.206.130.5 [1/0/p/q] - (rx1-na1-ru-unina-l1.na1.garr.net -> ru-unina-l1-rx1-na1.na1.garr.net) - |137 -> 137|
 0.0.0.0 -> 0.0.0.0 [1/0/p/d/o/l] - (- -> -) - |0 -> 0|
 0.0.0.0 -> 0.0.0.0 [1/0/p/d/o/l] - (- -> -) - |0 -> 0|
 0.0.0.0 -> 0.0.0.0 [1/0/p/d/o/l] - (- -> -) - |0 -> 0|
 143.225.190.41 -> 143.225.190.42 [1/0/p] - (- -> dis.r190.unina.it) - |137 -> 137|
 143.225.190.177 -> 143.225.190.178 [1/0/p] - (- -> mbonegrid.r190.unina.it) - |137 -> 137|
 143.225.190.169 -> 143.225.190.170 [1/0/p] - (- -> mbonedis.r190.unina.it) - |137 -> 137|
 143.225.190.98 -> 143.225.190.97 [1/0/p/q] - (- -> -) - |137 -> 137|
 172.21.190.5 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |0 -> 0|
 172.21.190.1 -> 0.0.0.0 [1/0/p/q/l] - (- -> -) - |0 -> 0|
 143.225.190.54 -> 143.225.190.53 [1/0/p/q] - (- -> -) - |137 -> 137|
 143.225.190.82 -> 143.225.190.81 [1/0/p/q] - (- -> -) - |137 -> 137|
 143.225.190.22 -> 143.225.190.21 [1/0/p/q] - (- -> -) - |137 -> 137|
```

3.6.4 Proxy Design

The proxy may actually be the various MERLIN monitors. Indeed, as the MERLIN server is performing some computing actions beyond measurements themselves, the MERLIN server should be placed outside the measurement proxy.

Each MERLIN monitor will receive a list of target IP addresses, as well as two file names: one for the MERLIN output, one for the logs.

[2] This lack is due to multicast routers responding with a unicast IP address not present in the list of multicast reported interfaces. Such an address is then added to the router as a local interface.

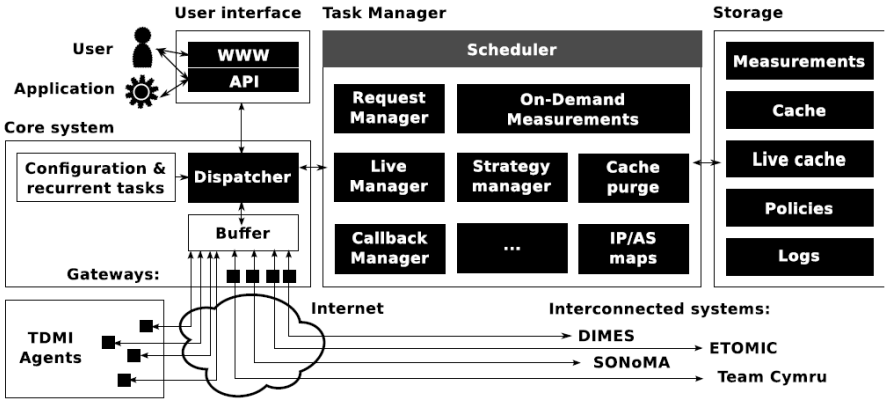


Figure 6: TopHat architecture

3.7 TopHat

Researchers use the PlanetLab testbed for its ability to host experimental applications in realistic conditions over the public best-effort Internet. Due to the scale of the machines involved, and of the diversity of measurement points, PlanetLab is an ideal candidate for Internet experiments involving active measurements.

TopHat is a system that fills a special niche, as it is designed to support the entire lifecycle of an experiment: from setup, through run time, to retrospective analysis. TopHat has been developed as the *active measurement* component of PlanetLab Europe, the flagship testbed of the OneLab experimental facility.

Due to the scale of the PlanetLab infrastructure, and of the spirit of the TopHat system, this makes it a natural candidate to be integrated in the mPlane infrastructure.

3.7.1 Overview

The TopHat architecture is described at [**?**]. In this section, we provide the mPlane point of view about TopHat.

TopHat architecture is depicted in Fig.**??** (taken from [**?**] with authors permission). TopHat agents can be instructed via XML-RPC API (at the command line, or over a TCP socket). On the bottom left are TopHat's own measurement agents, which are deployed within a slice on PlanetLab nodes. This slice is the TopHat Dedicated Measurement Infrastructure (TDMI), supplying measurements when no other system can do so. Other measurement infrastructures can be inter-connected to TopHat via specialized gateways (bottom right). Mediating between the user and application requests and the measurement infrastructures is the core system (at center left). The core system dispatches requests and measurements to the task manager, (top center). Data are stored in the storage subsystem (top right).

mPlane active measurement tools using TopHat would be integrated in a similar way through gateways to control the experimental loop (XML-RPC API) or to access the results of the experiment.

Query:

```
path_list = [('planet2.elte.hu', 'planetlab-europe-02.ipv6.lip6.fr'),
             ('ape.onelab.elte.hu', 'planetlab-europe-02.ipv6.lip6.fr')]
print TopHat.Get(auth, 'traceroute', 'now', path_list,
   ['src_ip', 'dst_ip', 'hops.ttl', 'hops.ip', 'hops.hostname', 'platform_name'])
```

Figure 7: TopHat configuration example: XML-RPC API for TDMI agent

3.7.2 Configuration

As TopHat configuration is rather generic, configuration should be dwelved for each specialized system. The one provided by TopHat is able to run traceroutes. The configuration is somewhat similar to that presented in Fig.**??**, (where access to a python shell is supposed). Alternatively, a TCP socket should be opened toward the TopHat agent, with instructions encoded in XML format; these instructions would be dispatched to the agent, triggering execution of commands specified in the XML configuration.

Of mPlane participants, ENST is using TopHat to execute a proof-of-concept agent capable of large-scale horizontal ICMP scans. The agent is providing a simple, reusable architecture, that can e.g., perform high-frequency sampling of large target sets for prolonged durations (e.g., sending 1 packet/second to 10^6 targets for a week), or Internet-scale low-frequency sampling (e.g., a periodic scan of a significant fraction of Internet hosts every hour). This differ for instance from Scamper, that is able to run a larger set of probes, but that is significantly less scalable.

In the case of ICMP scans, configuration input is:

- one or more IP address ranges (i.e., the set of target hosts in a compact form)
- a policy to share/dispatch the ranges (i.e., should a single target be probed by 1 or more or all agents)
- a policy to loop over the ranges (i.e., random, round-robin; this may have consequences of the number of ICMP messages seen by each subnetwork)
- a ping frequency (i.e., time between two consecutive probes toward the same host).

When configuring the agent, special care should be taken to avoid (even accidental) mis-using the agent, as allowing too large address ranges, or too high frequencies could lead to Denial of Service (see later).

3.7.3 Results

Taking the default TopHat TDMI agent as an example, the output resembles to Fig.**??** (taken from [**?**] with authors permission). Basically, TopHat fetches, aggregates and consolidates the output of all probing processes. At the current stage TopHat allows FTP access to the data, that is possibly periodically synchronized to a remote FTP server.

At ENST, during our preliminary investigation of TopHat, we already have (slightly) modified the TopHat architecture, allowing the negotiation of non standard FTP ports to get around firewall limitations. We are confident that further access to data can be provided depending on the type of results. A number of the statistics below are being implemented in TopHat by ENST.

Result:

```
[{'src_ip': '157.181.175.248',  'dst_ip': '132.227.62.19',
    'hops': [ {'ttl': '1', 'ip': '157.181.175.254',  'hostname': None},
              {'ttl': '2', 'ip': '157.181.126.45',  'hostname': 'taurus.taurus-leo.elte.hu'}, ...],
    'platform_name': 'TDMI'},
 {'src_ip': '157.181.175.247', 'dst_ip': '132.227.62.19',
    'hops': [ ... ],
    'platform_name': 'SONoMA'}
]
```

Figure 8: TopHat configuration example: XML-RPC API for TDMI agent

In the case of ICMP scans, output could range in:

- binary host reachability list (with timestamp)
  - 1 means reachable
  - 0 means unreachable (optionally augmented by ICMP error type)
- per-host RTT delay statistics have multiple interpretation
  - min RTT correlates with geographical distance, and could be used for geolocation (when multiple agents ping the same set)
  - variability implies path stability, or correlates with TCP performance over that link;
  - raw time series, as well as per-host statistics (e.g., mean, stdev, as well as percentiles) can be computed over arbitrary time windows.
- per-host queuing delay (RTT- min RTT) statistics correlates with user QoE
  - RTT variation correlates with user QoE
  - raw time series, as well as per-host statistics (e.g., mean, stdev, as well as percentiles)
- per-host TTL statistics correlates with path variability
  - small varation correlates with load balancing; large variations with routing changes.
  - raw time series, as well as per-host statistics (e.g., mean, stdev, as well as percentiles)

3.7.4 Proxy Design

The mPlane proxy will be pre-configured with the address and authentication information of the TopHat agent(s) it is proxying, and will provide the list of measurement names (in mPlane-standardized format), the IP subnets the agent can reach and probe, and various other attributes (such as preformance or other limitations) as mPlane *probe capabilities*. Upon receiving instructions from an mPlane probe client (e.g. supervisor), the proxy will perform the following steps:

Configuration: decode the mPlane-encoded active measurement instructions (i.e. extract $a)$ the actual measurement to take and $b)$ the targets, and $c)$ the instructions regarding results upload). Based on these, the proxy formulates the XML-encoded TopHat-specific measurement descriptor. Then, using the above-mentioned XML-RPC API, the proxy sends the TopHat XML descriptor over a TCP connection to the agent.

Results collection: based on the mPlane instruction received (see D1.1 section 5.3.2), the proxy will:

- on *synchronous*-mode or *delayed*-mode mPlane client requests: collects the results from the TopHat agent using FTP file transfer, parses the results, encapsulates them in mPlane data format, and sends them to the mPlane client. In the synchronous case the proxy will wait for the TopHat agent to finish and retrieves the results. In the delayed-access case, the proxy will just issue the receipt for the client and retrieves the results later.
- on requests setting up *bulk repository upload* (D1.1 sec. 5.3.2.3): when TopHat results are available, the proxy will retrieve them using FTP, and - after the necessary data format conversion - it uploads them directly to the mPlane repository assigned by the request.

Due to the long durations and large data volumes of typical TopHat measurements, the proxy will typically use the second method.

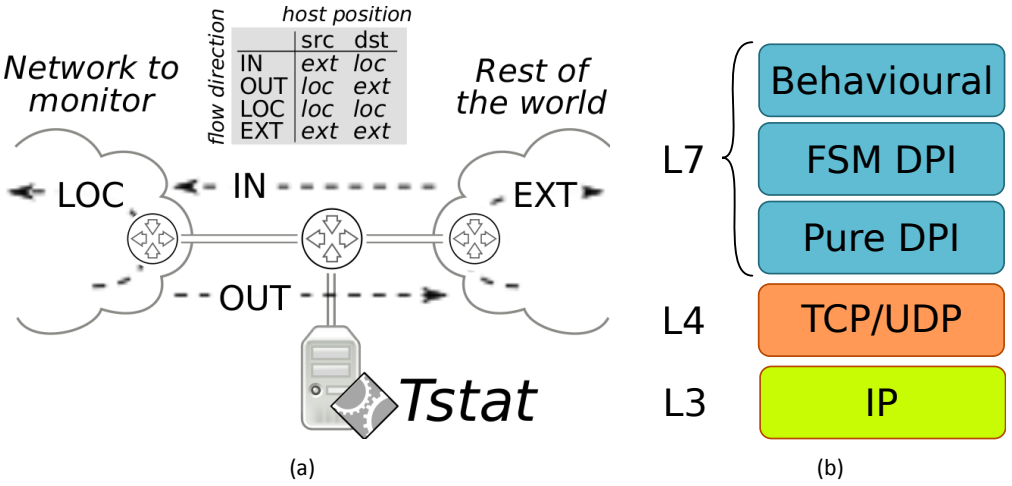


Figure 9: Tstat monitoring probe setup (a) and analysis workflow (b).

3.8 Tstat

3.8.1 Overview

Tstat (TCP STatistic and Analysis Tool) is an Open Source passive monitoring tool developed by the Telecommunication Network Group of Politecnico di Torino [**?**, **?**]. Tstat was born as a branch of tcptrace[**?**] with the purpose to perform TCP traffic analysis. Today, after more than 10 years of development, it is a full-fledged solution for IP networks traffic monitoring.

Tstat is able to process aggregates of traffic and to perform several statistics. Input traffic can be in the form of a *pcap* file [**?**] but more commonly Tstat is used to process live traffic directly collected from a network interface as depicted in Fig. **??**(a). In this case, Tstat is used to create a *passive monitoring probe*, i.e., it is installed on an off-the-shelf hardware PC deployed on the network. The PC receives all the traffic generated from and destinated to the local network which is being monitored. Tstat captures the traffic and inspects the aggregate to rebuild TCP/UDP connections and to perform several statistics from simple counters such as the number of packets sent/received up to more advanced traffic classification labeling to identify the application which has generated each connection.

Each observed packet is processed by Tstat using the sequence of analysis modules depicted Fig. **??**(b). After the frame de-encapsulation, the first analysis is operated at network-layer (L3) where IP packet headers are processed to extract statistics such as bitrate, packet length, etc., for both single IP addresses and subnets. Going up to the transport-layer (L4), a set of common statistics for both TCP and UDP are maintained on each connection, e.g., total packet and byte, round trip time (RTT), connection start/end time, etc. Finally, at the application-layer (L7), the main goal is to perform traffic classification, i.e., to identify the application that generated the traffic observed. As known, traffic classification is prone to fallacies and each classification methodology has its own peculiarities [**?**]. For this reason Tstat contains three different classification engines:

Pure Deep Packet Inspection (PDPI) uniquely identifies applications on a per-packet base by matching signatures in the packet payload. All the application signatures are collected in a dictionary. In case a packet matches on a signature, the connection related to the packet is labeled with the matching application name. In case no match is found after testing a maximum number of consec-

utive packets, the connection is labeled as ``unknown''. This engine is used to identify P2P/P2PTV applications, standard email and chat protocols, etc.

Finite Stat Machine Deep Packet Inspection (FSMDPI) extends the PDPI methodology to inspect more than one packet of a connection and possibly correlate the two traffic directions. This engine is useful to identify HTTP, RTP/RTCP, DNS etc. traffic

Behavioural techniques allow to classify traffic when encryption/obfuscation mechanisms are adopted to mask the packet payload. More in details, Tstat exploits statistical properties such as the packet size and the inter arrival time to obtain a characterization of the transport-level ``behaviour'' of the applications. This engine is useful to identify VoIP and P2P applications using obfuscation.

3.8.2 Configuration

Different levels of configurations are provided to adapt internal workflow for different needs. The following section provides a general description of these configuration mechanisms. For more details, please refer to `http://tstat.tlc.polito.it/HOWTO/HOWTO.html`.

Hard coded parameters represent the mechanisms to define specific properties at compile time. For example, most of the traffic classification and application characterization engines are included in the analysis workflow only if the associated compilation flags is specified (see `Makefile.conf`).

Other parameters control the data collection size (e.g., `MAX_TCP_PAIRS` corresponding to the maximum number of concurrent TCP connection that can be tracked) or characteristics of the analysis to be run (e.g., `UDP_IDLE_TIME` defining the amount of time after which a UDP connection is considered ended if no traffic is observed). All these parameters correspond to a very fine grained control of the software. As such, their values are hard coded in constants (see `param.h`) that can be defined only at compile time.

Command line options are used to express runtime configuration. This includes the output directories where to collect the statics computed, the input NIC or the set of pcap to process, and all the other configurations needed to configure the analysis modules. For example, by default all the IPs observed by Tstat are considered as external, i.e., the software able to understand each connection is between two host A and B and there are two traffic directions (A→B and B→A), but it is not able to label the host A and/or B as belonging to the network we are interested to monitor. For such purpose a list of the local IP addresses has to be provided (see the option `-N`).

Runtime engine options corresponds to a INI text file which, if provided at run time, can be used to tweak the configuration of Tstat without the need of restarting it. This configuration file is divided into sections, each containing a set of key-value pair, one couple per line (see `http://tstat.tlc.polito.it/HOWTO/HOWTO.html#runtime_module`).

Tstat can be compiled also as a library, namely *libTstat* (see `http://tstat.tlc.polito.it/HOWTO/HOWTO.html#libtstat_library`). This allows to integrate Tstat capabilities in other traffic analysis tools. All the configuration options listed above are still valid for libTstat. In particular, the library runtime configuration is defined through the same set of options available on the command line. The only difference with respect to the normal command line execution is that the options need to be collected in a text and passed to the library using an initialization API call.

3.8.3 Results

Tstat outputs statistics at different granularities:

Histograms: these data collections correspond to empirical frequency distributions obtained from flows aggregate. Tstat can track an extensive set of metrics and can be configured to periodically output their empirical distribution $F(X)$. Consider for example the case of monitoring a metric X such as the TCP packet size. Packet sizes $X = \{1\,byte, 2\,bytes, \ldots, 1500\,bytes, > 1500\,bytes\}$ can be assigned into different *bins* and counted how many packets fall in each bin over a window of time (e.g., 5 minutes). In this way, for each time window a new distribution is computed and saved into a text file having two columns, one to express the binning and the other for the collected frequencies $F(X)$. For an extensive description of the Histogram supported, please refer to `http://tstat.polito.it/measure.shtml#histo`

Round Robin Database: this output format corresponds to a different output representation of Histograms. A Round Robin Database (RRD) collects a set of histograms using a database in binary format [**?**]. The advantage of this format is that it handles historical data with different granularities. More in details, newer samples are stored with higher frequencies (e.g., every 5 minutes), while older data are averaged into coarser time scales to obtain different different granularities (e.g., 30 minutes, 4 hours, 1 day, etc.). This dramatically reduces disk space requirements (a priori configurable). Moreover, thanks to the tools provided by the RRD technology, it is possible to visually inspect the results. For example, RRD data collected by a Tstat probe can be queried in real time using a simple web interface [**?**], and plots of historical measurements over multiple sites can be shown.

Notice that, from the measurement point of view Tstat does not make differences between Histograms and RRD. This means that the same set of metrics available as Histograms can be saved in a RRD as well. The only difference is the output format (i.e., text files instead of binary files).

Connection logs: at an intermediate level of granularity, Connection logs are text files providing detailed information for each monitored connection. A log file is arranged as a simple table where each column is associated to a specific information and each line reports the two direction of a connection. Several logs are available to distinguish among different protocol and applications. For example, general statistics about TCP and UDP traffic are reported in separated logs. Separate logs are used to track VoIP, video streaming, web traffic, etc. The log information is a summary of connection properties. For example, the starting time of a YouTube video stream, its duration, total amount of bytes and packets downloaded, metadata related to the video downloaded (e.g., videoID, encoding format, duration and size), just to name a few. Besides computing statistics, Tstat labels each connection through a set of classification engines to identify the application which has generated the connection. For detailed descriptions of the logs format, please refer to the documentation provided on the Tstat SVN repository `http://tstat.polito.it/viewvc/software/tstat/trunk/doc/`.

Pcap traces: at the finest level of granularity, Tstat can dump pcap packet traces [**?**]. This output format, besides being useful to debug or pinpoint fine grained information related to the applications dynamics, is extremely valuable when coupled with Tstat classification capabilities. Indeed, packets can be dumped separated per application in different files based on the application they are related. For example, it is possible to instruct Tstat only to dump packets generated by Skype and BitTorrent applications, while discarding all other packets.

3.8.4 Proxy Design

Through the combination of the configuration described, it is possible to precisely define the Tstat analysis workflow. When the software is used in a passive monitoring probe, it is typically configured once by the system administrator and left running for long period of times (e.g., several months) without any further modification. Conversely, it is not so unfrequent that the runtime engine options are changed to enable/disable some outputs. In other words, the software is built once to provide the core functionalities and define the main set of output. The runtime engine option instead provides a way to enable/disable some functionalities. For example, the software can be compiled to include the HTTP analysis engine while the collection of the logs related to HTTP traffic can be colleted only when needed. This design choice is related to the fact that Tstat is expected to run on probes in operative networks. This means that frequent restart can be harmful for the continuity of the collected statistics. Moreover, configuration changes are restricted only to system administrators.

These considerations need to be taken into account to design the mPlane proxy for Tstat. In particular, there is no need for the proxy to expose configurations related to compiling options (e.g., the TCP/UDP idle time for the connections). Conversely, the proxy should be used to access to the runtime configuration option, i.e., the set of options that can be changed at runtime. In this way, the system administrator of the network probe is still in charge of defining the details of the software setup, i.e., the hard-coded configuration. However, through the proxy he can dynamically change some of the runtime parameters, i.e., the soft-coded configuration.

Overall, the Tstat proxy corresponds to a different way to access to the runtime configuration options. As such, in the context of the project, the set of options provided will be further expanded where needed to provide more tuning functionalities. It is however important to stress that these functionalities will be available under restricted access, i.e., only the system administrator and other allowed roles to act on the probes can interact with the Tstat proxy.

For what concerns the output, the typical setup is related to the collection of logs which can easily corresponds to massive amounts of data over time. This means that probes need to be equipped with a delayed mechanisms to consolidate the output on a separate repository. This process needs to know the Tstat configuration (e.g., connections logs are created each hour and saved to a specific directory) but it corresponds to a separate tool and managed independently from Tstat.

3.9 BlockMon

3.9.1 Overview

BlockMon is a modular system, implemented in C++11, for *flexible, high-performance traffic monitoring and passive analysis*. Its development started within the EU FP7 project *DEMONS* [**?**], and it is available open-source under the BSD license [**?**].

At a high-level, BlockMon provides a set of units called *blocks*, each of which performs a certain discrete processing action, for instance parsing a DNS response, or counting the number of distinct VoIP users on a link. The blocks communicate with each other by passing *messages* via *gates*; one block's output gates are connected to the input gates of other blocks, which allows runtime indirection of messages. A set of inter-connected blocks implementing a measurement application is called a *composition*.

In its original shape, users can develop efficient monitoring applications by interconnecting blocks that capture traffic from high-speed NICs with blocks that run algorithms for packet analysis on a single node. BlockMon supports dynamic reconfiguration to cope with the change of the environment in which the analysis runs. Therefore, BlockMon allows also to add, update and remove blocks from a running composition without the need of stopping it and without losing per-block state. Note that in order to add a block, this must be already present and compiled on the probe itself. At reconfiguration time, messages flowing through a composition are kept in flight until the reconfiguration is complete, then processed before any new messages.

Block Name	Description
PcapSource	Captures packets from a local interface or pcap trace files.
PFQSource	Captures packets using PFQ. Supports multi-queue NICs.
ComboSource	Captures packets from an INVEA-TECH COMBO card.
IPFIXExporter	Transcodes messages to IPFIX records, and exports them.
IPFIXCollector	Collects data via IPFIX and generates messages for appropriate records.
SerExporter	Serialize messages and exports them through TCP connection.
SerSource	Deserialize messages received through a TCP connection and generates messages to inject into the next blocks.
PacketFilter	Filters packets based on packet header fields.
PacketPrinter	Prints packets for debugging purposes.
PacketCounter	Counts received packets for debugging purposes.
IPAnonymizer	Anonymizes the source and destination IP addresses of a packet.
FlowMeter	Assembles packets into flows with natural lifetime export.
PeriodFlowMeter	Assembles packets into flows with periodic export.
FlowCounter	Periodically logs the number of messages it has received.
FlowFilter	Selects packets based on conditions specified over the 5-tuple.

Figure 10: Sample of provided blocks.

Blocks can implement a wide range of functionality including packet capture and filtering, monitoring, anomaly detection algorithms and export capabilities. A sample of the blocks available in the base Blockmon distribution as of this writing are described in figure **??**, and the list is always growing.

In the BlockMon architecture, blocks implement at least two methods: `configure`, which receives XML representing the block's configuration parameters, and `receive_msg` which is called when a message arrives at the block. Blocks can also be invoked on periodic or one-shot timers via the `handle_timer` method, and can perform high-frequency but non-periodic asynchronous work in the `do_async` method; this last method is mainly provided for source blocks (e.g., packet capture or message import via IPFIX),

which send messages but do not receive them. A block can use the `send_out` method to en-queue messages on the output gates and pass along the messages to the next block in the composition.

BlockMon supports the ability to extend a single composition across a set of nodes. This allows additional scalability and flexibility of deployment. CPU-intensive processing can be split among multiple hosts, and BlockMon instances at widely separated observation points can cooperate in decentralized monitoring and correlation of events in larger networks. This decentralization is implemented either by TCP or IPFIX [**?**] exporting and collecting processes. In the former case, TCP is used as binary transport protocol to exchange serialized messages. In the latter case, flow message can be imported from any IPFIX unidirectional flow, thus we can leverage data produced by existing IPFIX devices (e.g., IPFIX-enabled routers or standalone flow meters) to apply BlockMon as a flow analysis tool, in addition to its native packet analysis capabilities.

3.9.2 Configuration

To allow high-rate monitoring and data analysis, BlockMon provides the following software-based packet capture blocks: a standard pcap-based *PcapSource* block that can capture packets from a network interface or a packet trace; and a *PFQSource* block, which implements an adapter for the PFQ engine [**?**], which better leverages multi-core architectures. Hardware-accelerated capture using INVEA-TECH Combo cards is supported by a third *ComboSource* block. Examples of configuration parameters of these capturing blocks are the network interface or the subset of its associated hardware queues to capture from, and the packet filtering expression in Berkley Packet Filtering (BPF) format.

The capturing blocks turn the raw packets into BlockMon messages, which can be passed to the processing blocks. Configuration parameters are passed to each processing block through the XML composition: and example of how the XML composition looks like is provided in figure **??**. In the example, a source block captures packets on the given interface and passes them to a flow meter, which keeps a table of per-flow statistics, such as number of packets and bytes being exchanged by the flow and, depending on the configuration, also the packets making up that flow. Flow timeout can be configured as a parameter.

```
<composition id="example">
  <block id="source" type="PcapSource">
      <params>
      <source type="live" name="eth0"/>
    </params>
  </block>

  <block id="meter" type="Flowmeter">
      <params>
          <active_timeout ms="500"/>
          <idle_timeout ms="100"/>
    </params>
  </block>

  <block id="export" type="IPFIXExporter">
    <params>
        <domain id="1"/>
        <export host="collector.example.net" port="4739"
            transport="tcp"/>
        <datatype name="ipv4flow"/>
    </params>
  </block>

  <connection src_blk="source" src_gate="source_out"
              dst_block="meter" dst_gate="in_pkt"/>
  <connection src_blk="meter" src_gate="out_flow"
              dst_block="export" dst_gate="in_msg"/>
</composition>
```

Figure 11: Example of XML composition.

3.9.3 Results

BlockMon can export the measurement results of the composition running on top of it synchronously (for every processed message) or asynchronously (periodically), so measurements can continuously run and results are available without having to wait for the whole measurement campaign to finish. To allow to export information for each message, the last block of the composition needs to invoke the `send_out` method, whereas for periodically exporting, statistics are collected and exported every time the `handle_timer` method is invoked.

Results can be exported through one of the methods mentioned in section **??**, i.e., by using TCP as binary transport protocol or via IPFIX.

Depending on the composition run on top of BlockMon, several outputs are possible. In particular, any combination of the following information elements can be exported:

- source and destination IPv4 address;
- source and destination Transport port;
- flow start and end time;
- bytes and packets exchanged by the flow;
- flows with the highest number of packets or bytes.

3.9.4 Proxy Design

The proxy will work in the "Delayed data access" mode, see deliverable D1.1, Figure 6.3 (c). It will be able to handle more than just one BlockMon probe and will advertise the management IP addresses and names of those probes available to run the measurements. The proxy will also advertise for each probe the available basic processing units, or the set of compositions that can be run on the probe itself. In details, based on the capabilities of the BlockMon nodes (the basic processing units and compositions that nodes can run), the proxy will generate a list of corresponding mPlane-compatible measurements and advertise them to the mPlane devices, such as the supervisors. The proxy will maintain an internal mapping between the measurement names as specified within mPlane devices, and the corresponding BlockMon compositions which can carry out such measurements.

The proxy will receive specifications for each measurement with the specified parameters as mentioned in section **??** and based on its internal mapping it will generate the composition to run. If the proxy knows the specified BlockMon probe and it can configure the required measurement then it sends back a receipt.

The proxy will also receive receipt requests and send back the results if the specified measurement exists on the specified probe, if the instantiation has been successful and the timeout to export the results has expired or the measurement has finished. In this case, the proxy will take care of making the results provided by the BlockMon probes in a format which is compatible to the mPlane devices.

3.10 Misurainternet

3.10.1 Overview

MisuraInternet is the Italian method of monitoring the QoS (Quality of Service) provided by Internet Service Providers (ISPs) to check the user SLA. This project is based on AGCOM n.244/08/CSP Resolution and the technical aspects comply with pertinent ETSI standard. It is based on *active probes*. Measurements estimate the throughput of a client-server connection, the latency between the same client-server couple and the packet loss. The analyses of the measurements are made according to the methods that are proposed by ETSI in [**?**]. The measurement systems are:

- ISP measurements of throughput and QoS, carried out by means of a monitoring network consisting of 20 test points, adopting *active probes*, distributed across Italy and specifically placed in major Italian cities (i.e., one test point for each region). This monitoring network operates all day every day.
- End user measurements, concerning throughput and QoS, that are carried out directly at the user's home, using an open source software (*software agent*) `Ne.Me.Sys` (Network Measurement System) that is developed specifically for this project by Fondazione Ugo Bordoni.

The aim of the MisuraInternet is the creation of a generic and efficient instrument for the consumer. Every aspect of the project is public and every activity is well detailed on the dedicated web site (www.misurainternet.it). Several details were already given in D1.1 and D.3.1.

The monitoring system structure is designed to implement FTP based tests between two hosts located in relevant network sections.

3.10.2 Configuration

Active probes are distributed over Italian territory adopting access networks that can be considered representative of the average conditions in terms of congestion (number of customers belonging to the same DSLAM) and physical parameters (such as attenuation or distance from the central office).

FTP servers are located in IXPs, which are big Internet gateways for ISP networks. They are physical infrastructures through which ISPs exchange Internet traffic between their networks (autonomous systems). By means of a measurement server located in an IXP, it is possible to ensure that each client will measure only the performance related to its own ISP network. This approach has been chosen since it allows comparable and reproducible results.

To guarantee the reliability of the measurement Ne.Me.Sys adopted the following conditions:

1. only the user PC, that is involved to measure, is allowed to be connected to the LAN, so the other hosts have to be disconnected during the measurement;
2. traffic, that does not belong to the measurement traffic is denied;
3. traffic over a Wi-Fi connection is not allowed;
4. the CPU usage has to be under a predetermined threshold (< 85%);

5. the RAM usage has to be under a predetermined threshold (< 95%) and the RAM of used PC has to be at least 12 MBytes.

In the second release of the software, many difficulties that occurred during user measurements have been overcome, and some constraints have been relaxed. One of the most stringent constraint concerning the alien traffic; indeed, in the first release, the user web-surfing was forbidden, but sometimes some applications automatically connected themselves to the Internet, without the user's permission. In these cases Ne.Me.Sys. did not allow the measurement. In the second release a small amount of alien traffic is permitted; in this way the end user can complete more easily the measurement cycle and the measurements are always valid. In particular a threshold has been introduced: the alien traffic has to be less than 10% of the measurement traffic.

3.10.3 Results

According to AGCOM recommendation we should distinguish between wireline and wireless measurements. Here we report the wireless case since it is more general. Probes outputs are:

1. download-upload throughput. It is defined as the ratio between the file dimension and the time transmission duration. The transmission duration is the period that starts from the instant when the wireless network has received all the information necessary to start the download/upload transmission up to the last bit of the test file is received.
2. Transmission data failure (down/upload). Percentage among data transmission failure and the transmission data attempts. Failure can occur when the test file is either not totally transmitted or without error within a time interval.
3. Packet loss.
4. Transmission delay in terms of Round Trip Time. It is obtained by means of Echo Request/Reply (Ping) in agreement with protocol ICMP (RFC 792: "Internet Control Message Protocol")
5. Delay variation (jitter).

More details on the probe outputs are reported in D3.1 Sect. 2.8.

3.10.4 Proxy Design

MisuraInternet active probes were carried out to work both as agents and devices. From a comparison between Misurainternet and MPLANE architecture (described in D1.1) we can confirm that the FUB probes are compatible with the scheme reported in Fig. 1 of this deliverable. The only mean difference is the fact that FUB probes, in current version, do not use to communicate their configuration to the supervisor. However such a requirement can be upgraded in a simple way. We point out that the FUB probes, based on Python, are simply programmable. Data can be elaborated on scales of days, weeks, months and years. Data transfer can be synchronous, asynchronous and delayed. Each probe communicates its IP address, location and timestamp (Network Time Protocol). Looking at the probe requirements reported in D1.1, pag. 43, we can summarize that FUB probes currently satisfy the characteristics described in points: 1, 2, 3, 4, 5, 6, 7 and they can be upgraded to satisfy 8, 9, 10, 11, 12, 13, 14, 21, especially to operate in a passive environment. More investigation would be necessary to satisfy the remaining points.

3.11 Firelog

3.11.1 Overview

FireLog [**?**] is an active probe composed by two different parts: client side engine for measurement and server repository for analysis. The client-side part is a plugin to Firefox, which can be easily integrated into the end users' browser. While users are surfing their web pages, the plugin will record a list of metrics (described later) and periodically transfer them to the server repository. To protect user privacy, all URLs and server host names can be also hashed before the transfer.

The final aim of this distributed probe is to give information on the possible root cause of the poor user experience in a web session, to identify whether the cause resides in the home network, the remote server or in the ``wild'' internet (see Fig. **??**).

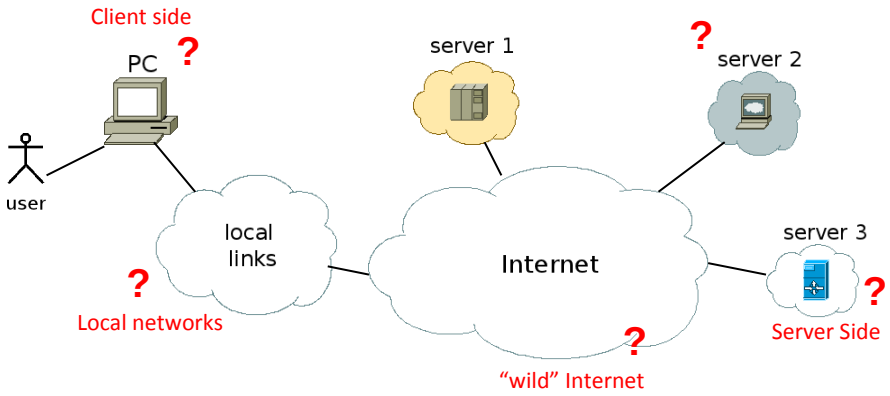


Figure 12: Firelog overview

3.11.2 Configuration

The plugin comes as a Firefox extension (XPI file), that needs to be manually installed[3]. After this step, a real user can browse desired web pages, or a manually set list of web pages to be visited can be given to the probe in form of a text file. At the server side, a PostGreSQL server must be set up. A customizable set of predefined stored procedures to analyze the data collected from the plugin.

3.11.3 Results

The typical procedure for loading one object is shown in Fig **??**. The time elapsed can be broken down into several metrics, such as, for example, (a) the time elapsed between the first DNS query and first corresponding response with valid IP address(es); (b) the time between the first SYN packet sent by the client and its corresponding first SYN-ACK packet received from the server; and (c) the time between the first data packet (normally carrying a GET request) sent by the client and its first corresponding ACK. Additionally, (d) when the ACK packet from the server in response to the data packet that carried the GET request does not contain any payload, there will be a time-gap between this ACK and the first data

[3] `http://firelog.eurecom.fr/page/links/instruction.html`

packet from the server. Since this gap between client request and data reception is caused a delay due to server processing, this gap is referred to as service offset. Finally, (e) the time between the reception of the first and last data packet containing the data of a Web object.

The results can be either: a set of the aforementioned performance metrics (timings) for a specific web browsing session on a single Firelog instance; or the aggregated result computed via a distributed clustering on several Firelog instances. That is, each Firelog probe can run as a single instance, returning the set of metrics (see next section) for a pre-determined set of web sessions, or it can cooperate with other Firelog instances to compute the centroids of the actual performance results for further processing of the aggregated results.

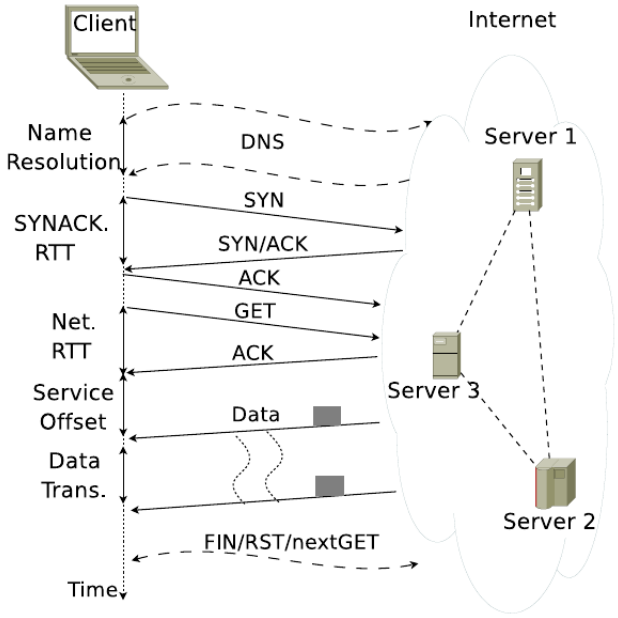


Figure 13: Metrics for downloading one web object

3.11.4 Logged metrics

ff_version	current firefox version
method	HTTP query method, such as GET or POST
host	host/server name for current object
request_event_ts	request event starting time at the browser
content_type	content type of current object (e.g. png, flv, html....)
content_lengthcontent	length declared by the HTTP header (declared length, NOT real bytes)
acceptencodingAcceptEncode	HTTP header, set by the browser
contentencoding	ContentEncode HTTP header, e.g. zip, gzip, ...
content_load_ts	timestamp of browser event of end of parsing current page
load_ts	timestamp of page fully load
session_abort_ts	if current page is aborted by the user before page is fully loaded, then, record this time
server_cnxs	HTTP Connection header (to check if persistent connection is DISABLED by the server)
server_os	server declared operating system at HTTP header
server_http_v	server HTTP version
http_id	random number inserted in the HTTP header to couple with the packet trace
session_start	starting time of the whole browsing session that current object belonging to
session_url	page URL of the whole browsing session that current object belonging to
if_complete_cache	cache status of current object (0:not cached, 1:cached, but need to validate by the server, e.g. HTTP response = 304, 2: completely cached)
localaddress	client IP for current connection, if from cache, then 0.0.0.0
localport	client port for current connection, if from cache, then 0
remoteaddress	server IP for current connection
remoteport	server port for current connection (normally 80)
response_code	HTTP response code from the server for current request (normally 200)
http_request_bytes	http request size in bytes
http_header_bytes	http response header size in bytes
http_body_bytes	http response data body size
http_cache_bytes	if object is from cache, this is the object size
dns_start	DNS event start timestamp
dns_end	DNS event end timestamp
dns_time	dns duration delay (e.g. end time - start time)
syn_start	TCP handshake event start timestamp (e.g. SYN packet is sent)
syn_end	TCP handshake event end timestamp (e.g. SYN/ACK packet is received)
tcp_cnxting	tcp connecting duration (tcp end time - tcp start time)
send_ts	timestamp of sending HTTP request
send	time duration to send out an http request
get_sent	timestamp of HTTP request that is already sent out
first_bytes	timestamp of receiving the 1st bytes from the server side
app_rtt	time duration between HTTP.sent and receiving 1st byte
endtime	ending timestamp of current HTTP transfer
data_trans	time duration of the whole data download.(e.g.endtime-1st.byte)
full_load_timepage	full load time in milliseconds for the whole web page that current object belonging to
content_load_time	page parsing time in milliseconds for the whole web page that current objects belonging to
abort_time	page abandoned time
first_annoy	duration between current web page beginning to the 1st clicking by the user
nr_annoying	number of clicking by user for the current web page
tabid	a random id corresponding to a tab in firefox
client_ip	client local ip address
referer	http Referer header
curr_wifi_quality	signal strength in percentage
cpu_idle	current idle CPU resource
cpu_percent_ffx	CPU percentage of the whole firefox
mem_free	total free memory of client PC
mem_used	total memory in use of client PC
mem_percent_ffx	percentage of memory used by firefox
ping_gw	ping delay to default gateway (if icmp response enabled)
ping_dns	ping delay to default DNS server
ping_google	ping delay to www.google.com host (or www.google.fr server)
obj_aborted	whether current object is stopped by the user before finishing downloading all the data
location	HTTP Location header (e.g. if http response code shows a Redirection)
if_google_host	boolean value indicating whether a "google" key word exists in the host name
session_domain	domain of the browsed web page
uri	Unique Resource Identifier of current object
client_os	client operating system info

3.11.5 Proxy Design

In the case of Firelog, collecting measurements from users' random web browsing, external control is obviously not possible. The proxy probe will connect to Firelog's native repository sever, and serve measurement results from there upon query. Thus, one proxy instance serves for all the Firelog agents submitting data for the native repository.

The proxy will implement mPlane's synchronous data query interface (see Deliverable D1.1, 5.3.2.1, also Figure 6.3(b) in D1.1). On receiving the result query from a mPlane client (user or supervisor), the proxy immediately queries the native Firelog repository and (after performing data type conversions if needed), returns results synchronously.

3.12 Pytomo

3.12.1 Overview

Pytomo [**?**] is a Python based tomographic tool[4] to precisely evaluate the quality of experience (QoE) as perceived by the user when viewing a video from Youtube. Pytomo emulates the video consumption process as it would be experienced by a ``real'' end user. For the videos set in an initial list the Pytomo tool first finds the IP address of the cache servers from which these videos are downloaded. The cache server is pinged to obtain the RTT times. Then a download for a limited amount of time is started to calculate the different statistics of the download, enriching the main QoE metric, i.e., the number of *video stalls*, with many network measurements and use multiple DNS servers to understand the main factors that impact QoS and QoE.

Pytomo is an active probe, as the perturbations between the probe and the end-user are not taken into account in the passive monitoring analysis over a large set of users. Moreover, as the video data transfered during HTTP video streaming can become huge, large scale passive monitoring would need too much processing. Therefore, Pytomo monitors the HTTP video streams directly from the end-user's computer.

3.12.2 Configuration

The initial list of videos need to be provided. Then the configuration is built by setting the following parameters.

MAX_ROUNDS	Maximum number of crawl rounds to performed
MAX_CRAWLED_URLS	Max number of urls to be visited
MAX_PER_URL	Max number of related videos to be selected from each url
MAX_PER_PAGE	Max number of related videos to be considered for selection from each page
EXTRA_NAME_SERVERS	A list containing the name of the resolver and its IP address.
PING_PACKETS	Number of ping packets to be sent
DOWNLOAD_TIME	The duration for which the video must be downloaded
BUFFERING_VIDEO_DURATION	The duration for which the video is to be buffered
MIN_PLAYOUT_BUFFER_SIZE	The size of the buffer for the video stream
RESULT_DIR	The directory to store the text results
RESULT_FILE	The file to store the text results
DATABASE_DIR	The directory to store the result database
DATABASE	The name of the result database
TABLE	The name of the result table
LOG_DIR	The directory to store the log files
LOG_FILE	The file to store the logs
LOG_LEVEL	Parameter to set the log level(choose from DEBUG, INFO, WARNING, ERROR and CRITICAL)
PROXIES	The HTTP Proxy to be used (Set from command-line at run-time by user)

3.12.3 Results

Pytomo provides the following APIs.

- **compute_stats(url)** Returns a list of the statistics related to the url. The contents of the list are : (url, cache_url, current_stats) where current_stats is a list containing:
  - Ping_times;
  - download_statistics;
  - DNS resolver used;

[4] `http://code.google.com/p/pytomo/`

- **format_stats(stats)** Functions used to format the stats obtained from compute_stats function so that they can be inserted into the sqlite3 database. The stats are converted into a tuple. The arguments to this function is the list returned by compute_stats().
- **md5_sum(input_file)** Function to generate the standard md5 of the file. Done to cope with the large file values taken from Python distribution.
- **check_out_files(filepattern, directory, timestamp)** Returns a full path of the file used for the output. It checks if the path exists, if not then the file is created in the path if possible else it is created in the default user directory.
- **do_crawl(result_stream=sys.stdout, db_file=None, timestamp=None)** Crawls the urls given by the urlfile.txt(present in the package). The crawl is performed upto MAX_ROUNDS or MAX_VISITED_URLS. The statistical results obtained are inserted into the db_file.

Results are then stored in a database:

Timestamp	A timestamp indicating the time of inserting the row
Service	The website on which the analysis is performed.
Url	The url of the webpage
CacheUrl	The Url of the cache server hosting the video
IP	The IP address of the cache server from which the video is downloaded
Resolver	The DNS resolver used to get obtain the IP address of the cache server.Example Google DNS, Local DNS
DownloadTime	The Time taken to download the video sample (We do not download the entire video but only for a limited download time)
VideoType	The format of the video that was downloaded
VideoDuration	The actual duration of the complete video
VideoLength	The length(in bytes) of the complete video
EncodingRate	The encoding rate of the video
DownloadBytes	The length of the video sample(in bytes)
DownloadInterruptions	Nb of interruptions experienced during the download
BufferingDuration	Accumulate time spend in buffering state
PlaybackDuration	Accumulate time spend in playing state
BufferDurationAtEnd	The buffer length at the end of download
PingMin	The minimum recorded ping time to the resolved IP address of the cache server
PingAvg	The average recorded ping time to the resolved IP address of the cache server
PingMax	The maximum recorded ping time to the resolved IP address of the cache server
RedirectUrl	In case of HTTP redirect while downloading the video from the cache server. The address of the URL to which the the request has been redirected is stored in this field

3.12.4 Proxy Design

The Pytomo proxy can be implemented as a "wrapper" around the Pytomo instance deployed on a server. (Consequently, one proxy instance will represent one Pytomo instance for mPlane.) As Pytomo offers no remote control and results retrieval interface, the proxy instance needs to be installed on the same computer. Then, upon receiving a mPlane measurement command, the proxy will:

- configure Pytomo with the URL and other parameters (based on the information encoded in the mPlane request)
- launch and manage the Pytomo process locally while it is performing the measurement
- upon completion, reading the Pytomo results database, extract the fields requested, and encode the reply in mPlane-compatible manner

As even a single measurement round might take significant amount of time, the proxy implements mPlane's delayed data access mode (See Deliverable D1.1, Figure 6.3.(c)).

3.13 DATI

3.13.1 Overview

DATI (Deep Application Traffic Inspection) is a passive traffic analysis system developed by Telecom Italia.

The system, build upon FreeBSD technologies [**?**], can recognize traffic flows from static BPF signatures and collect state information (volume, packets and duration) as well as application-level data based on an XML description of the protocol stack.

The last release of the software has been tested to work on a double 10Gbe Intel interface with off-the-shelf hardware.

3.13.2 Architecture

The architecture of the DATI system is depicted in the picture below.

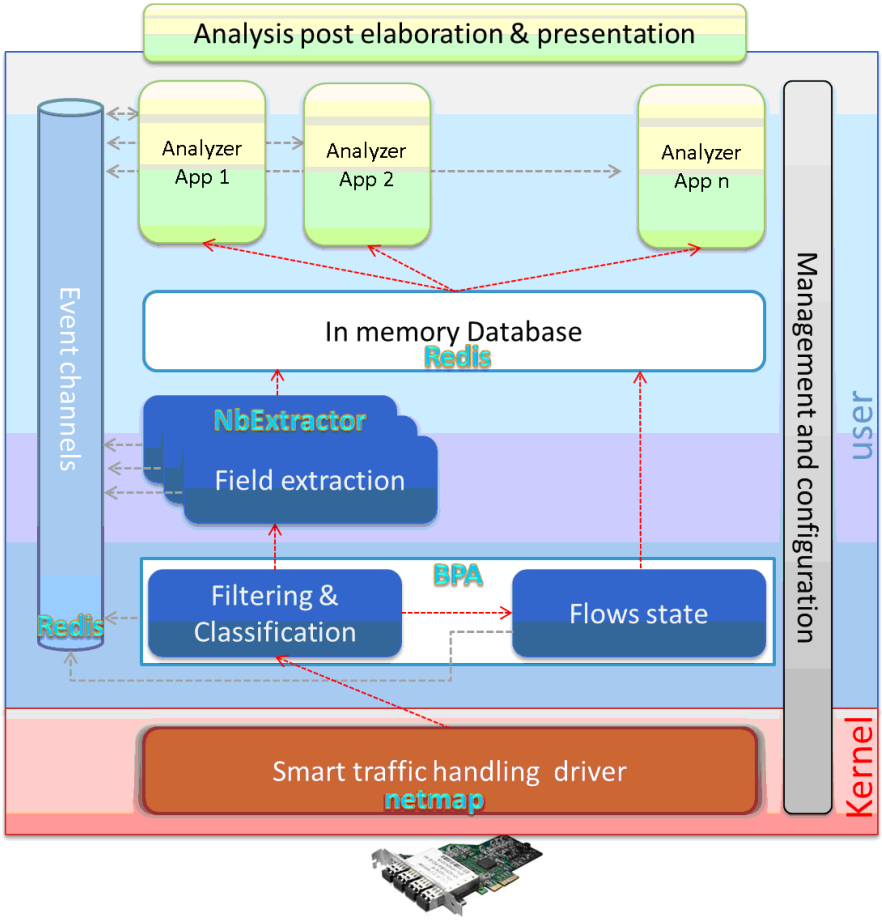


Figure 14: DATI Architecture

The network card driver (NetMAP [**?**]) is the only part of the system located in kernel space, responsible for sending of traffic flows directly to user space, minimizing the impact of kernel performance draw-

backs.
At user space, the BPA module is responsible of two basic functions:

- filtering and classification, extracting only the packets matching a pre-configured signature, aggregating them as macroflows and sending the traffic through dedicated channels
- flow state, keeping track of informations about each traffic flow (represented by a tuple of src and dst ip address, layer 4 protocol and src and dst port), like flow duration and volume, start timestamp, etc

BPA Module sends the traffic that matches each signature towards "tun interfaces", representing emulated network interfaces on which NbExtractor [**?**] operates, a tool implemented in cooperation with Politecnico of Turin, that, based on a XML definition of the structure of the protocol stack, parses the traffic and extracts from the payload only the fields the analyzer app needs.

Nbextractor outputs the fields directly to an in-memory db, Redis [**?**], that stores the record, signaling the presence of it through dedicated channels. Redis DB offers a publish/subscribe mechanism, useful to keep a separation between macroflows: on top of this mechanism, analyzer apps can subscribe to a specific channel to process the informations, that resides on each record they are listening to, in a near real time basis, fetching and processing the record without the need of storing it on file system.

At the highest level of the system a management and configuration module controls and configures each low level module.

3.13.2.1 NETMAP

NETMAP [**?**] is a smart traffic handling driver designed for FreeBSD and Intel network interfaces, developed by Luigi Rizzo (University of Pisa): the main purpose of the approach that netmap implements is to expose the traffic flowing through the interface directly to userspace, copying only pointer of the memory where are located the packets. Techniques used by netmap are:

- memory-mapping of network interface buffers
- I/O batching, to aggregate groups of system calls
- fixed size ring buffers

The performance the netmap driver can achieve are much higher than kernel based drivers, potentially reaching up to 90Gbs traffic with 800bytes packets [**?**].

3.13.2.2 BPA

BPA module accesses traffic flows directly using netmap driver, classifies flows on a static signatures basis, keeps flows state and send a copy of matching packets towards a tun interface. BPA functional steps are the setting up of the traffic capture on the selected network interfaces, bringing the network interface on netmap mode and putting it in promiscuous mode. A load balancer splits all the packets through different threads, calculating a hash using a balancing formula that provides a fair distribution between the threads, that puts the packets belonging to the same flow on the same thread, on both directions. Each processing thread extracts the pointer of each packet and stores or updates the info

associated to the flow. Once the update step is completed, the packets are processed by each active bpf filter: if the matching occurs the packets are copied in a memory slot dedicated to the thread and finally they are sent to the tun interface.

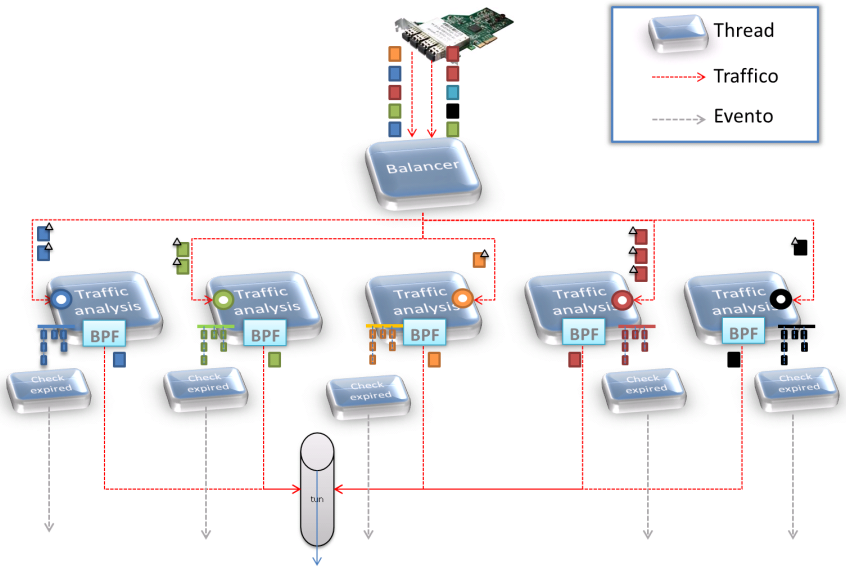


Figure 15: BPA multithreading

Other threads, one for each processing thread, implement the cleaning of the flow state tables in order to remove the items that are not active anymore. If a flow is not updated since a configurable time interval, the thread sets it as expired (if the processing thread that operates on the same flow table sees that expired flag, it removes the item). If the thread finds an expired flow, it also send a message on a dedicated channel publishing the information regarding the expired flow, such as IPs, ports, volumes, duration and start timestamp.

3.13.2.3 NBEXTRACTOR

NbExtractor is a tool developed by Politecnico of Turin based on a library, for the analysis of network traffic, called NetBEE [**?**]. The function it implements is the extraction of the fields located inside the packets captured by BPA module and sent to the tun interfaces. The tun interface is a virtual interface seen by nbextractor as a real physical interface: nbextractor listens on tun and performs the field extractions needed, using an XML configuration file that defines the structure of the protocol stack, the fields format and the way the field is presented at output. NbExtractor can send the fields directly to standard output, to text files or to sqlite databases. In the DATI system, TI developed an extension of NbExtractor that allows to output the fields to the in-memory Redis databases, storing the information as key-value pairs.

NbExtractor is a single process tool. In the DATI system, each analyzer app has its own NbExtractor process that listens on the tun interface dedicated to the app.

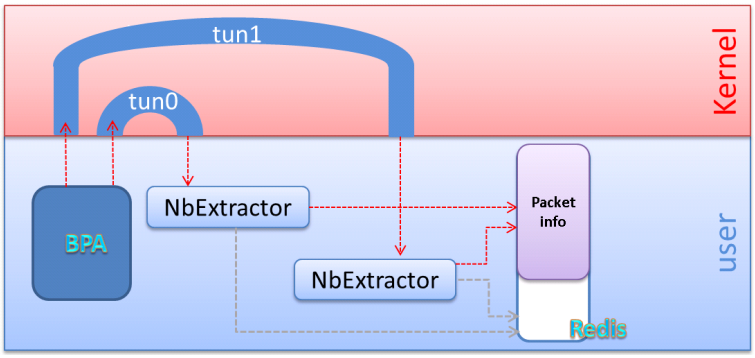


Figure 16: NbExtractor

3.13.2.4 REDIS

DATI uses the in-memory database Redis [**?**] to avoid copying raw traffic data on file system before the analysis made by the analyzer app. Older versions of DATI stored the raw data on MySQL or sqlite databases, on which processing scripts operated to get the statistics. These intermediate steps, that introduced a large amount of disk usage and very expensive processing time, are now bypassed allowing an analyzer app to operate in a near real-time basis.

Preliminar tests, made to evaluate Redis performances, have demonstrated that, with 500 byte data record, with a sequence of 6 atomic operations, the Redis database supports up to 15 thousands sequences per second per producer, with negligible CPU load (less than 10%). The system can also scale using:

- more than one producer
- different indipendent databases

3.13.2.5 ANALYZER APP

Analyzer apps are the consumers of data extracted by DATI system: these apps are designed to manage specific data flows; as an example, the main analyzer app in DATI is the video QoE manager: receiving the useful fields located inside video traffic packets, the app processes these data and extracts the statistics in a human readable form, giving the possibility to render it as a chart or table. Many analyzer apps can be developed, each app listening on one or more redis channels to get the informations needed.
Early versions of DATI were focused on P2P traffic, with analyzerapps dedicated to eMule and BitTorrent protocols, extracting statistics like the popularity of contents, the number of users using that protocol and the volume that each user generated, the traffic load aggregated by authonomous system and much more statistics.
Analyzer apps are developed using different programming languages, such as javascript, perl or python. The output of each app is a set of sqlite databases, one for each configured statistic interval.

3.13.3 Configuration

DATI can be configured by means of a text file defining the BPF filter and the XML protocol definitions to be applied.

There is a configuration file for each module involved in the system:

- bpa.conf, contains the definitions of the bpf filters used to capture traffic and the tun interface the traffic is sent to
- nbextractor.xml, defines the protocol specification used to parse the traffic and to extract the fields from the payload; from command line, the nbextractor tool chooses the redis database and the redis channel to which the data are to be sent
- redis.conf, defines the location of the in memory db
- analyzer.app.conf, each consumer can have a specific configuration file

3.13.4 Results

DATI can produce a set of measures and analysis related to the quality of experience of video traffic. Some of the current implemented measures are:

- Youtube QoE: 4 different QoE scores are proposed, one is related to video set up latency, the second is related to buffer stall events, one is related to packet loss events and the last is a weighted average of the first three scores. It is possible to get a QoE scores differentiated by access nodes or have a whole network score
- HTTP traffic distribution: it is possible to get the location of the server from which video content is downloaded, giving a perspective about how the content provider manage user requests, how volume of traffic is splitted between different server location
- Video traffic popularity
- Video formats and content types

3.13.5 Proxy Design

DATI system, as a passive probe, can be viewed as an independent probe that collects statistics without the need to be configured from mPlane actors. As DATI exposes no remote interfaces, the mPlane proxy is implemented as a process on the DATI server itself, thus each mPlane proxy represents a single DATI instance. Currently it is not planned to expose any control interface towards mPlane, although this may change in future versions (on-off control and the on-demand re-composition of textual configuration files such as BPF expression and the XML descriptor files are natural candidates for this).

As for other passive probes, the proxy will advertise the configuration of DATI as mPlane *capabilities* and *measurements* (see D1.1 sec. 5.2.2.1 and 5.2.2.2, respectively).

The main task of the proxy is the implementation of mPlane's data interface (see D1.1 sec. 5.3.2), where mPlane result queries are mapped onto DATI internal storage queries, and results are mapped onto

mPlane data types. DATI results databases can be queried for results detailed in the previous section, filtered by various conditions (e.g. user, access node, time interval etc.). The proxy probe will also act as an intermediate for asynchronous data export (see D1.1. 5.3.2.3), issuing bulk queries for the various DATI modules locally, and transporting the resulting data though the network to be stored in a mPlane repository.

3.14 MobiPerf: Mobile Network Measurement System

3.14.1 Overview

MobiPerf [**?**] is an open source framework for measuring network performance on mobile platforms. It consists of two major components: a client application running on mobile devices and a server suite running on a Linux server. With server support, the client application executes a list of network measurements and reports the results to the user and the server. It is an active measurement system.

The experiments include

i) available bandwidth,

ii) LDNS lookup tests,

iii) LDNS Coverage tests,

iv) Landmark tests and

v) Reachability Tests.

3.14.2 Configuration

The mobiPerf platform requires instrumentation at the user's devices. This is done by downloading an app through an App store or by shipping the client with the device. Currently an analysis is triggered manually by the users.

The server is typically used for bandwidth end-to-end tests and requires minimum configuration.

3.14.3 Results

A number of results are generated:

Device statistics
The mobile app collects device information such as the carrier name (e.g., Movistar), network type (e.g., UMTS, CDMA), cell ID, Location Area Code (LAC), signal strength, local IP address, IP address seen by the remote server, GPS (latitude and longitude).

Downlink/Uplink Performance Test
To test downlink and uplink performance, the client and server exchange bulk data for 16 seconds. Packet traces are collected at the server side. The downlink and uplink server are running on port 5001 and 5002.

LDNS Lookup Test
A list of 80 popular mobile domains is used to to test the DNS lookup time of the configured DNS servers. In order to make sure that LDNS server has already cached these domains, each domain is queried twice.

LDNS Coverage Test
In order to understand how local DNS servers are assigned to different users a uniquely host is queried (generated just for the experiment). Since the domain request issued by the client is unique the local

DNS server in the cellular network is expected to make a DNS request for the unique domain. Therefore, a mapping of users GPS locations and their assigned LDNS server IPs is generated.

Landmark Test
The latency to a list of landmark 20 servers is measured. The IP is used directly for these measurements.

Reachability Test
MobiPerf measures the port blocking behavior for by scanning a list of preselected ports to test reachability. For each of these ports, there is a corresponding TCP server that echos any contents back to its client and reports the blocking and the blocking stage (connect, send, receive).

3.14.4 Proxy Design

In this case, the the proxy is designed the same way as for Firelog (sec. **??**). As the measurements are activated by the users at random, no control action is accomodated. The proxy connects to MobiPerf's central measurement server, and serves mPlane result queries by translating mPlane queries to native MobiPerf ones, and mapping the results onto mPlana data types and encapsulating the result.

With MobiPerf, servers can be deployed at multiple locations and the client device should decide which one to connect to (e.g. to the user's home provider's server, or to the roaming proviers's server, or to an independent server operated by a third-party or a regulator). Server selection is not exposed to mPlane (it is not within mPlane's authority to decide), but the proxy needs to be able to connect to multiple MobiPerf servers and locate the one having the measurement results sought.

3.15 Akamai Web Performance Monitoring Tool

3.15.1 Overview

Similarly to MobiPerf (sec. **??**, MobiTest [**?**] measures page load times on many of today's most popular mobile devices. It offers detailed performance information, ranging from total page load times to individual request headers and timings. It is also designed to capture screenshots during page load, and show a video visualizing the page load as it happened. MobiTest is similar in intent to Pytomo (**??**), the main difference being that MobiTest is optimized for mobile platform measurements, while Pytomo (it it's current form) cannot be executed on mobile devices.

Mobitest is an open source project and Akamai also offers a free hosted service at www.akamai.com/mobitest. A client for iOS, Android and Blackberry devices is available enabling users to measure on their own devices and within their internal networks, in addition to the free community service.

3.15.2 Configuration

The MobiTest platform requires instrumentation at the user's devices. This is done by downloading an app through an App store or by shipping the client with the device.

The Web Server can be supported by any OS that supports Apache and PHP (Linux and Windows have both been tested). Apache 2.x+ with PHP 5.3.0 or later should be supported.

The configuration is quite complex but it is fully described here [**?**]

3.15.3 Results

A number of results are generated about loading and rendering each component of a webpage. These are saved into a raw CSV log file that can be parsed by the proxy. Furthermore, aggregated results are also generated.

Please refer here [**?**] for an extensive information about the results.

3.15.4 Proxy Design

As with MobiPerf, the MobiTest can easily integrated with mPlane to diagnose any issues related with web-page loading. The client can be deployed as part of the user-side mobile probes of Mplane to perform active measurements. The servers can be either in the location of the supervisor or another probe distributed within the network (e.g,. the user's router or the ISP).

4 Conclusions

In this document, a number of existing measurement systems, external to mPlane were described. The Consortium will consider these systems for inclusion into the mPlane ecosystem by the project through developing specific proxy probes. The informal design and interfaces of such proxy probes were described, as much as possible at this stage of the project. The selection of systems spans a wide variety of aspects that characterize measurement systems in general. E.g. passive (e.g. Tstat) and active (e.g. Tracebox) systems were chosen. Commercial (e.g. Cisco's Ping Agent), regulatory (MisuraInternet) as well as research and open source systems were included in the list. Systems that run on the public Internet as well as a system running on PlanetLab will be included. Also a wide variety of metrics are collected by these systems ranging from topology data (such as provided by MERLIN) to QoE metrics (e.g. Pytomo). Finally, different networks and network segments are covered such as fixed and mobile networks. Given this wide range in a number of dimensions, the mPlane consortium believes to cover enough characteristics of measurement systems to make sensible choices in the proxy probe design and to demonstrate the feasibility and effectiveness of including existing systems into the mPlane this way.