mPlane

an Intelligent Measurement Plane for Future Network and Application Management

ICT FP7-318627

mPlane Architecture Specification

Author(s):	ETH	B. Trammell (ed.)
	POLITO	M. Mellia, A. Finamore, S. Traverso
	NETVISOR	T. Szemethy, B. Szabó
	FHA	R. Winter, M. Faath
	ENST	D. Rossi
	ULG	B. Donnet
	TI	F. Invernizzi
	ALBL	D. Papadimitriou

Document Number:	D1.3
Revision:	1.0
Revision Date:	31 Oct 2013
Deliverable Type:	RTD
Due Date of Delivery:	31 Oct 2013
Actual Date of Delivery:	31 Oct 2013
Nature of the Deliverable:	(R)eport
Dissemination Level:	Public

Abstract:

This document defines the mPlane architecture and the interface provided between mPlane clients and components. The protocol is divided into layers: an information model for mPlane messages -- measurement capabilities, specifications, results, and event notifications; serialization representations using JSON, YAML, and XML, and session protocol bindings using SSH and HTTP over TLS.

Keywords: architecture, use case, scenario, measurement, platform

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.

Contents

Disclaimer 3

Document change record 7

1 Introduction 8
1.1 Architectural principles 9

2 Architecture Terminology and Specification 10
2.1 Components 10
2.1.1 Probe 10
2.1.2 Repository 10
2.1.3 Client 10
2.1.4 Supervisor 11
2.1.5 Reasoner 11
2.2 Information Model Terminology 12
2.2.1 Element and Primitive 12
2.2.2 Schema 13
2.2.3 Capability 13
2.2.4 Specification 13
2.2.5 Result 13
2.2.6 Verb 14
2.2.7 Notification 14

3 mPlane Protocol Information Model Specification 15
3.1 Statements 15
3.1.1 Statement Type and Verb 15
3.1.2 Parameters section 16
3.1.3 Results and Resultvalues sections 17
3.1.4 Metadata section 17
3.1.5 Link 17
3.1.6 Support for indirect export 18
3.2 Notifications 18
3.2.1 Notification type and verb 18
3.2.2 Receipt 19
3.2.3 Redemption 19

3.2.4 Indirection 19
3.2.5 Withdrawal 19
3.2.6 Interrupt 20
3.2.7 Exception 20
3.3 Reference Message Sequences 20
3.3.1 Direct and Reverse Query 21
3.3.2 Delayed Query 21
3.3.3 Initiating Indirect Export 21
3.3.4 Canceling Indirect Export 22
3.3.5 Notes on Message Flow Initiation and Capability Withdrawal 23
3.3.6 Notes on Error Reporting and Recovery 24

4 mPlane Protocol Message Representations 25
4.1 JSON 25
4.2 YAML 25
4.3 XML 26
4.4 Text-CSV 26
4.5 Representing Element Values 26
4.6 Representing Parameter Constraints 27

5 mPlane Session Protocol Bindings 28
5.1 Hypertext Transfer Protocol (HTTP) over Transport Layer Security (TLS) 28
5.1.1 Capability push, Specification pull 28
5.1.2 Capability pull, Specification push 29
5.1.3 Capability push, Specification push 30
5.2 Secure Shell (SSH) 30
5.3 Component and Client Discovery 30

6 Core Type System Specification 31
6.1 Primitives 31
6.2 Element naming and matching rules 31
6.3 External element mappings 32
6.4 Core elements 32
6.5 Elements supporting reference implementation 33
6.6 Link section URL schemes 33
6.7 Export section URL schemes 33

7 Conclusions..... 35

A Notes on Interoperability with LMAP..... 36

B Reference Implementation Requirements..... 37
B.1 Component Reference Implementation Requirements..... 37
B.2 Additional Supervisor Reference Implementation Requirements..... 38

Document change record

Version	Date	Author(s)	Description
0.1	29 May 2013	B. Trammell (ETH) ed.	define structure
0.2	27 Sep 2013	B. Trammell (ETH) ed.	first complete revision
0.3	18 Oct 2013	B. Trammell (ETH) ed.	frontmatter completion
0.5	22 Oct 2013	B. Trammell (ETH) ed.	comments from Barcelona plenary
1.0	31 Oct 2013	B. Trammell (ETH) ed.	comments post-Barcelona plenary

1 Introduction

This document defines the initial revision of the mPlane architecture for coordination of heterogeneous network measurement components: probes and repositories that measure, analyze, and store aspects of the network. The architecture is defined in terms of a single protocol, described in this document, used between **clients** (which request measurements and analyses) and **components** (which perform them). Sets of components are organized into measurement infrastructures by association with a **supervisor**, which acts as both a client (to the components it supervises) and a component (to the clients it serves). This arrangement is shown in figure 1.1 and further described in the rest of the document; the ``capability -- specification -- result'' cycle in this diagram comprises the mPlane protocol.

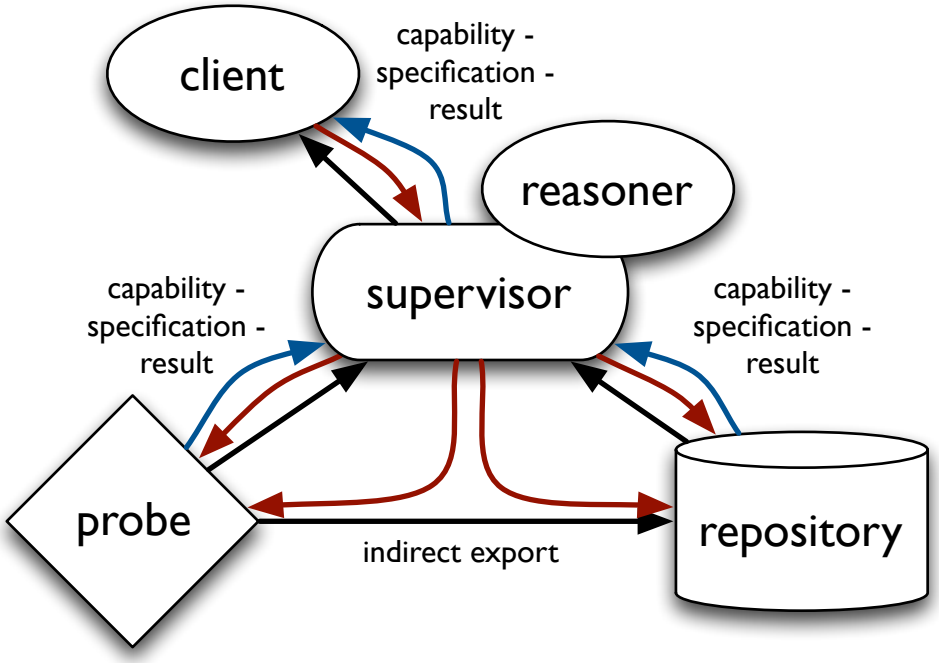

Figure 1.1: Architecture overview

This architecture is drawn from the set of first principles described and elaborated in the next section, and from the requirements and scenarios considered in D1.1.

Since the mPlane protocol is essentially modular, consisting of an information model, separate message representations, and separate session protocol bindings, this deliverable builds the description of the mPlane protocol from the bottom up. First, section 2 defines the terminology used to define the architecture. Section 3 defines the information model for the core mPlane message types: statements and notification. Section 4 defines representations of this infomodel in JSON, YAML, and XML. Section 5 defines bindings to HTTP over TLS and SSH as session protocols for transporting mPlane messages.

These sections, taken together, define the mechanics of the mPlane protocol. The key to measurement interoperability using mPlane is not this protocol so much as it is the types used to define the

measurements and the results. The set of elements from which these types can be built is described in section 6. The complete definition of the type system is out of scope for this document, drawing from workpackages 2, 3, and 4, and will be maintained by the project as a live registry.

Note that this deliverable reflects the status of the mPlane architecture and protocol as of the publication date. Future developments (e.g. changes to support unforeseen requirements arising during pilot integration, modifications for interoperability with existing measurement platforms and emerging standards such as LMAP (see Appendix A)) may result in revisions to the architecture and protocol specification and the reference implementation thereof. Up-to-date references to the reference implementation and protocol specification will be available on the mPlane website.

1.1 Architectural principles

In elaborating the architectural principles described in D1.1, a key realization we came to in early discussions about the architecture was that when components advertise their capabilities, distinctions among those components at an architectural level are somewhat artificial. We have maintained the distinction between probes and repositories in this document where it is informative, and to align with initial architectural guidance given in the description of work, but a key thing to realize about the mPlane architecture is that **everything is a component**.

Given the heterogeneity of the measurement tools and techniques applied, and the heterogeneity of component management, especially in large-scale measurement infrastructures, reliably stateful management and control would imply significant overhead at the supervisors and/or significant measurement control overhead on the wire to maintain connectivity among components and to resynchronize the system after a partial disconnection event or component failure.

A second architectural principle is thereby **state distribution**: by explicitly acknowledging that each control interaction is best-effort in any case, and keeping explicit information about each measurement in all messages relevant to that measurement, the state of the measurements is in effect distributed among all components, and resynchronization happens implicitly as part of message exchange. The failure of a component during a large scale measurement can therefore be accounted for after the fact.

This emphasis on distributed state and heterogeneity, along with the flexibility of the representations and session protocols used with the platform, makes the mPlane protocol applicable to a wide range of scales, from resource- and connectivity-limited probes such as smartphones and customer-premises equipment (CPE) like home routers up to large-scale backbone measurement devices and repositories backed by database and compute clusters.

mPlane defines a self-describing, error- and delay-tolerant remote procedure call protocol: each capability exposes an entry point in the API provided by the component; each statement embodies an API call; and each result returns the results of an API call. The final key principle in the mPlane architecture, which allows it to be applied to the problem of heterogeneous measurement interoperability, is **type primacy**. A measurement is completely described by the type of data it produces, in terms of schemas composed of elements. The key to measurement interoperability in mPlane is therefore the definition of a type registry, the core elements of which are defined in this deliverable.

2 Architecture Terminology and Specification

The following terms are used to describe the elements of the mPlane architecture.

2.1 Components

A *component* is any entity which implements the mPlane protocol specified within this document, i.e., advertises its *capabilities* and accepts *specifications* which request the use of those capabilities. The measurements, analyses, storage facilities and other services provided by a component are completely defined by its capabilities.

In the following subsections, we describe the characteristics of specific kinds of mPlane components; arrangements among types of components are shown in Figure 1.1.

2.1.1 Probe

A probe is a component that performs measurements. It either returns the results of these measurements directly to the component requesting them, or forwards them indirectly via a specified second protocol to another component or collector.

2.1.2 Repository

A repository is a component that provides access to a data source. It may provide read-only access (e.g. as in interface to an external data source, such as the global domain name system or BGP information from routing looking glasses), or read-write access (storing data received from Probes, and providing retrieval and analysis of that stored data.)

Note that some measurement components, especially those based on passive measurement and employing a local data store, share qualities of both probes and repositories.

2.1.3 Client

A client is anything which uses the mPlane protocol to consume services provided by one or more components. Within the architecture, this term has two distinct meanings. In any given interaction over the mPlane protocol, the client is the entity which receives capabilities from, and sends specifications to, a component. In an infrastructure of mPlane components managed by a a supervisor (see below), the client is the entity on whose behalf the supervisor acts: it receives capabilities from, and sends specifications to, the supervisor.

Unless otherwise noted, ``client'' in this specification refers to the first sense of this definition.

2.1.4 Supervisor

A supervisor collects capabilities from a set of components, and provides capabilities based on these to its clients. Algorithms at the supervisor aggregate the lower-level capabilities provided by these components into higher-level capabilities exposed to its clients. A supervisor is, in effect, a combination of a client and a component: it acts as the client for the components in its domain, acts as a component to its clients. With respect to a component, a supervisor is simply a type of client.

The set of components which respond to specifications from a single supervisor is referred to as an mPlane *domain*. Interdomain measurement is supported by federation: a local supervisor delegates measurements in a remote domain to that domain's supervisor. This arrangement, shown in figure 2.1, greatly simplifies access control and aggregation.

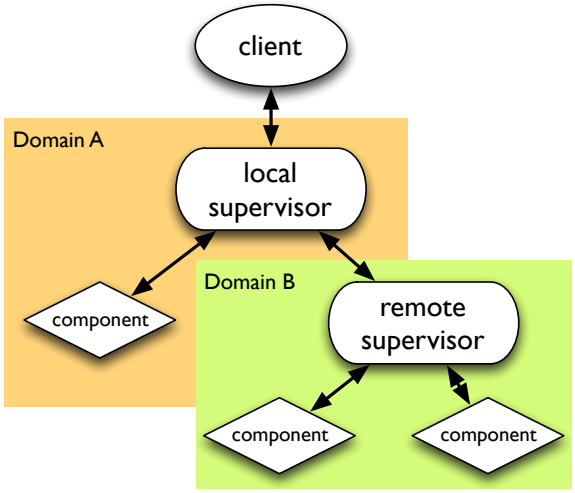

Figure 2.1: Interdomain delegation in mPlane

The access control specified in section 6 of D1.2 is applied to specifications received at the supervisor; it is the supervisor's responsibility to decide whether a received specification is authorized before passing the resulting lower-level specifications down to the components in its domain.

2.1.5 Reasoner

A reasoner is a client which supports iterative measurement by learning the best subsequent measurements to perform in order to drill down to the root cause of a specified problem. For reasons of implementation efficiency, a reasoner may be colocated with a supervisor, and may cooperate with the supervisor to provide reasoner-enhanced capabilities to the supervisor's clients.

2.2 Information Model Terminology

Components interact with each other by exchanging *statements*, which are the unit of messaging in the mPlane protocol. Statements define what a component can or should do, or what the results of an action were. The *schemas* defining the measurement described by a statement are expressed in terms of *elements*.

In the following subsections, we define these terms in more detail. The relationships among information model entities are shown in Figure 2.2.

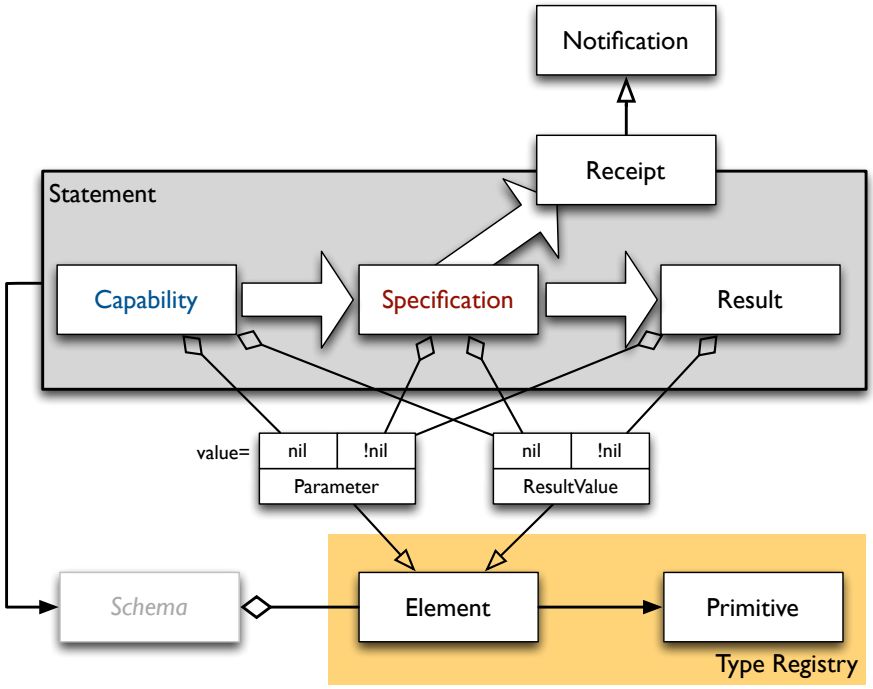

Figure 2.2: Information model: Statements, Elements, and Primitives

2.2.1 Element and Primitive

An element is a name for a particular type of data with a specific semantic meaning; it is analogous to an IPFIX Information Element[2], or a named column in a relational database. Elements have a primitive type, which defines the values the element can take and the representation thereof. Element primitive types supported by mPlane include the following:

- `string`: a sequence of UTF-8 encoded characters
- `natural`: an unsigned integer
- `real`: a real, floating point number
- `bool`: a true or false (boolean) value

- `time`: a timestamp, expressed in terms of UTC
- `address`: an identifier of a network-level entity
- `url`: a uniform resource locator

2.2.2 Schema

A schema is a collection of elements defining a measurement; it is analogous to an IPFIX Template[3] or a table definition in a relational database. The schema defining a measurement uniquely identifies the measurement.

In order to allow for experimental measurement to be tested in the mPlane architecture without disrupting its operation, an *experimental schema* can be defined by prepending an `x-` prefix to the measurement name. As a byproduct, experimental schema also provides the ability to lexicographically scope the different schema, that can be useful to ``hardwire'' the resolution from a schema to a metric (e.g., in particular cases, it could be desirable to select a specific implementation among the available ones).

2.2.3 Capability

A capability is a statement of a component's ability to perform a specific operation, conveyed from a component to a client (or supervisor). It does represent a guarantee that the specific operation can or will be performed.

A capability has a set of *parameters*, elements for which a value is required in order to perform the operation, which may optionally have *constraints* on the acceptable values. A capability also has a set of *results*, elements for which values will be returned when the capability is invoked. The parameters and results together comprise the schema of the measurement.

2.2.4 Specification

A specification is a statement that a component should perform a specific operation, conveyed from a client (or supervisor) to a component. It can be conceptually viewed as a capability whose parameters have been filled in with values.

2.2.5 Result

A result is a statement produced by a component that a particular measurement was taken and the given values were observed, or that a particular operation or analysis was performed and a the given values were produced. It can be conceptually viewed as a specification whose results have been filled in with values.

Note that, in keeping with the stateless nature of the mPlane protocol, a result contains the full set of parameters from which it was derived.

2.2.6 Verb

Statements have verbs, which specify the operation they describe. The special verb `measure` describes a measurement, as provided by a probe, noting that the identification of the measurement is implicit in the statement's schema. The special verb `query` likewise describes a query with an implicit identification, as `measure` but without the implication of starting a measurement activity.

The special verb `collect` specifies that a repository can accept data of a given type indirectly via a specified protocol. The special verb `store` specifies a repository can directly accept single data items given as parameters via statements.

Other verbs allow components to declare non-measurement analysis operations, where the operation is not necessarily unambiguously described by the associated schema.

2.2.7 Notification

A notification is an asynchronous message in the mPlane protocol; it does not share the structure or workflow of a statement, and can be used in an application-specific way to provide asynchronous indication of an event at a component. Types of notifications used by mPlane itself include Receipts, Redemptions, Indirections, Withdrawals, Interruptions, and Exceptions.

3 mPlane Protocol Information Model Specification

mPlane protocol interactions consist of messages sent between components and clients. There are two kinds of messages: statements, which assert some properties of measurements which can be, should be, or have been performed; and notifications, which are used to send other information and metainformation between components and clients.

3.1 Statements

The structure of an mPlane statement is shown in figure 3.1.

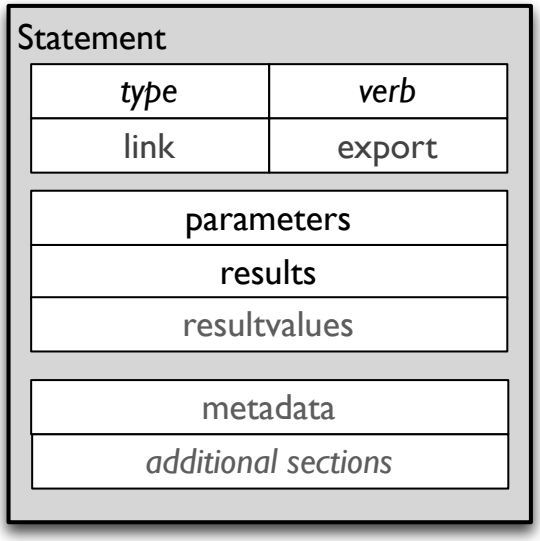

Figure 3.1: Statement Structure

3.1.1 Statement Type and Verb

Every statement has a type (one of *capability*, *specification*, or *result*), which identifies which point in the measurement workflow it represents; and a verb, which identifies the operation to which it pertains.

Verbs supported by the core protocol include:

1. `measure`: take measurements described by the statement
2. `collect`: collect measurements described by the statement
3. `store`: directly store measurements contained in the statement
4. `query`: retrieve data from a repository described by the statement; equivalent to `measure`.

Other verbs are application-specific, and may, for instance, name specific analyses available at a repository.

3.1.2 Parameters section

Statements have parameters: these are the elements provided by the client to specify the specifics of the operation to perform. The *parameters* section is an ordered list of these elements. For capabilities, each parameter has a constraint, which defines the set of acceptable values for the parameter. For specifications and results, each parameter has a single value.

3.1.2.1 Temporal scope

All statements must have a temporal scope, consisting of *start* and *end* parameters. These specify the time interval during which a capability is valid, during which a specification is to be run, or during which a result was collected.

There are five special time values to be used for temporal scope. $-inf$ represents the infinite past, now represents the absolute present, and $+inf$ represents the infinite future.

$once$ is a special end time meaning ``at the natural end time of the operation''. $whenever$ is a special start time meaning ``at an arbitrary time to be chosen by the component''.

The special time interval $[now, +inf]$ in a capability represents a probe which can do on-demand measurement without any local storage; this is the default for probes. The special time interval $[-inf, now]$ in a capability represents a repository which can answer queries about the past but has no probing capability; such repositories are, however, encouraged to expose the time of their first available record in the *start* parameter.

The special time interval $[now, once]$ in a specification means ``immediately run the specified operation once then stop''. The special time interval $[whenever, once]$ in a specification means ``run at your convenience the specified operation once then stop''. $once$ can also be used as an end time with a specific start time in the future.

The special time interval $[now, +inf]$ in a specification means ``run the specified operation until interrupted''; it is only valid for measurements or other operations using indirect export as in section 3.1.6.

Periodic measurements or actions may be scheduled using an additional *period* parameter. If present, this directs the component to perform the specification once every *period* between *start* and *end*.

The use of temporal scope in statements requires a reasonably precise synchronization of the clocks of all the components interacting with each other; an accurate synchronization would also be desirable for correlation with external sources of event data. This synchronization is explicitly out of scope for the mPlane project, as we presume that many network management tasks already require synchronized clocks, and refer to the state of the art in this field.

mPlane components and clients should use an existing clock synchronization approach such as NTP (for sub-second level accuracy) or a satellite-based synchronization system such as GPS (for sub-millisecond level accuracy), according to application requirements. At a minimum, mPlane components should synchronize their clocks to an appropriate NTP server.

3.1.2.2 Topological scope

Network measurements must have a topological scope; for active measurements, this consists of a `source` parameter, and for passive measurements, an `observer` parameter. Probes which have multiple possible vantage points expose these through multiple possible values in constraints in their capabilities.

3.1.3 Results and Resultvalues sections

Statements have result values: these are the elements provided by the component containing the results of the measurement or operation performed. In a statement, these are split into two sections.

The *results* section is an ordered list of elements the statement supports as results; the elements listed here together with those in the parameters section make up the schema of the measurement described by the statement. The results section is present in all statements.

The *resultvalues* section is only present in results statements, and is an ordered list of ordered lists of values in row-major order. The resultvalues section is separated from the results section for efficiency in multiple representations.

3.1.4 Metadata section

Invariant information about a component (e.g., the algorithms it uses for measurement, revision numbers of component implementation, etc.) which may be used for later analysis of the data produced by the component, but which is not part of the component's configuration, may be represented in a metadata section. The metadata section is an ordered list of elements with values.

The *metadata* section is optional in all statement types, but typically appears in results and capabilities statements only, as it is a one-way mechanism for a component to share relevant information about itself.

3.1.5 Link

To support callback and indirection of control, capability and specification statements may have a *link* section. The value of the link section is a single URL to which control information may be sent in response to the statement.

To perform an operation advertised in a capability with a link section, a client should connect to the component at the given URL and send a specification. To send results of a measurement resulting from a specification with a link section, the component connect to the client at the given URL and send result statements back.

If no link section is given, then statements should be sent to the same component from which the corresponding capabilities were retrieved, and results returned to the same client, possibly in the same connection (see section 5 for more).

URL schemes supported for link sections are listed in section 6.6.

3.1.6 Support for indirect export

The result statement type was designed for inline return of small amounts of (generally highly aggregated or otherwise processed) result data; it is specifically not optimized for large-scale data transfer. For continuous or bulk data transfer between mPlane components, the platform supports *indirect export* instead.

To support indirect export, capability and specification statements may have an *export* section. The value of the export section is a list of one or more URLs to which results can or will be sent.

For *measure* capabilities (i.e., for measurement services provided by probes) a URL may consist *only* of a scheme, which represents an ability to export to any repository using the named protocol; or it may be a complete URL, if the probe has been administratively configured to use only one or a set of repositories. For *collect* capabilities (i.e., for collectors for a given indirect export protocol at a repository) each URL must be complete, specifying the URL on which the collector will listen.

Export section URLs must be complete for all specifications.

URL schemes supported for indirect export are listed in section 6.7.

3.2 Notifications

The structure of an mPlane notification is shown in figure 3.2.

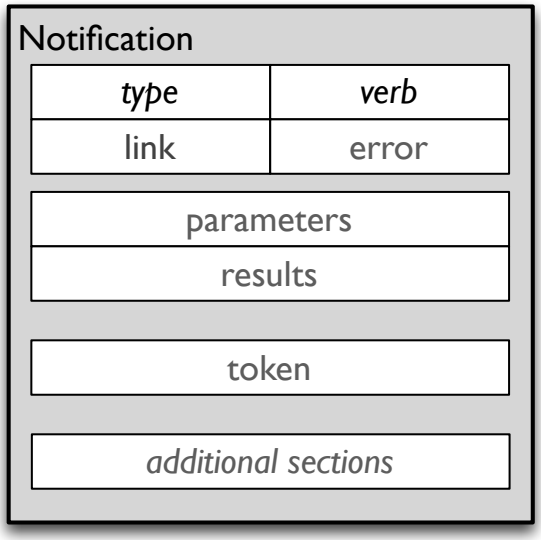

Figure 3.2: Notification Structure

3.2.1 Notification type and verb

Like statements, notifications have a type and a verb.

In subsequent subsections we look at the six types of notification supported by the core system: receipt, redemption, indirection, withdrawal, interruption, and exception.

3.2.2 Receipt

Receipts are returned for specifications which either (1) will never return results, as they started an indirect export connection, or (2) will not return results interactively, as the operation producing the results will have a long run time.

Receipts contain the same sections as the specification they are returned for, with identical values and verb. A component may optionally add a *token* section, which can be used in future redemptions and interruptions by the client. The content of the token is an opaque string generated by the component.

3.2.3 Redemption

A redemption is sent from a client to a component from which it received a receipt for long-running operations. The redemption may contain the fields of the original specification, the token from the receipt, or both, to identify the results it would like to retrieve.

The component responds with either a result (if the results are available) or another receipt (potentially with a new token) if not.

A redemption may contain a *link* section, as in section 3.1.5; if so, the component should return the result, when available, to the given URL.

3.2.4 Indirection

An indirection is sent in a reply to a capability given to a client by a component in a component-initiated capability advertisement via HTTP. It contains a *link* section, as in section 3.1.5, to send a link to a URL from which future Specifications should be retrieved.

An indirection may contain a temporal scope and a period, as well; in this case, the component should contact the client with the specified period during the specified time range.

3.2.5 Withdrawal

A withdrawal is sent from a component to a client to which it has previously sent a capability to cancel the capability. The withdrawal contains the same sections as the capability they withdraw, with identical values and verb. Withdrawals are *not* mandatory, especially in cases where capabilities are passively made available over HTTP. Withdrawals may also be sent in response to specifications for previously but no longer available capabilities.

3.2.6 Interrupt

An interrupt is sent from a client to a component from which it received a receipt for long-running operations in order to cancel the operation. The interruption may contain the fields of the original specification, the token from the receipt, or both, to identify the operation it would like to interrupt.

3.2.7 Exception

Exceptions are returned for specifications which cannot or will not be executed; and may be asynchronously signaled among components to signal other error conditions. By default, components will signal asynchronous exceptions to their supervisor.

Exceptions for non-executed or failed specifications contain all the sections of the failed specification, with identical values and verb, like a receipt.

Exceptions additionally contain a *error* section; the content of the error is a human-readable error message generated by the component. A machine-readable taxonomy of error conditions, which would allow automated recovery and retry, may be developed for a future revision of this specification.

3.3 Reference Message Sequences

The most common sequences of messages in mPlane are queries and indirect export; each of these can operate normally or with an exception. The sequences of messages used in the core interface are shown in figure 3.3, and detailed discussion of key nominal interactions is found in the subsections below. Note in these subsections that ``client'' refers to the client from the component's point of view: in most cases, this will be a supervisor.

The query interfaces are intended for the retrieval and transmission of small relatively volumes of data: instantaneous active measurements from probes and results of data analysis from repositories. The indirect interfaces are intended for bulk data transport.

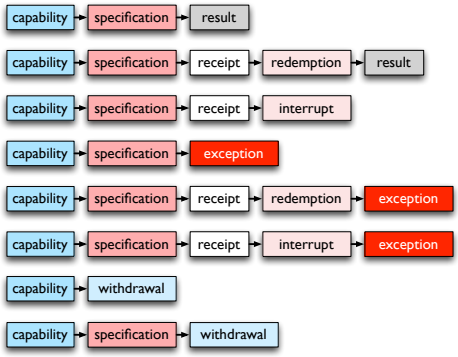

Figure 3.3: Order of messages in mPlane interface

Other message sequences may be supported depending on specific application requirements.

3.3.1 Direct and Reverse Query

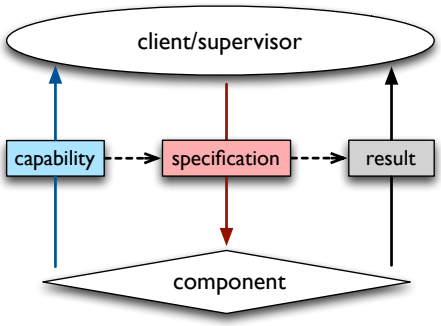

Figure 3.4: Direct Query workflow

In a direct query (figure 3.4), the client receives and stores a capability from the component. At some later time, the client decides to make use of the capability, and sends the component a specification derived therefrom. The component sends back a result immediately as a reply to the specification.

A reverse query follows the same message sequence as a direct query: in this case, however, the component initiates a connection to the client to publish the capability to the client, then later initiaties a connection to the client to retrieve a specification, if any. Once the result of a specification is available at the component sends it back.

3.3.2 Delayed Query

In a delayed query (figure 3.5), the client receives and stores a capability from the component, just as with a direct query. At some later time the client decides to make use of the capability, and sends the component a specification derived therefrom.

Instead of sending back a result in reply, it sends a receipt instead. The client saves the receipt and presents is back to the component as a redemption at a later time. If the result is available, it is sent in reply to the receipt; otherwise, a new receipt may be returned for another redemption attempt.

Clients initiating direct queries must be prepared to handle them as delayed queries, instead.

3.3.3 Initiating Indirect Export

To initiate indirect export (figure 3.6), a client must coordinate with two components, the exporter and the collector; these will typically be a probe and a repository, respectively.

Assuming the client has received capabilities from these components indicating that the exporter can measure and export a type that is compatible with a type that the collector can collect over a compatible protocol, it sends a specification to the collector to begin collection from the exporter, waits for an affirmative receipt (which should be available in the reply), then sends a specification to the exporter to begin export to the collector, and waits for an affirmative receipt. These receipts

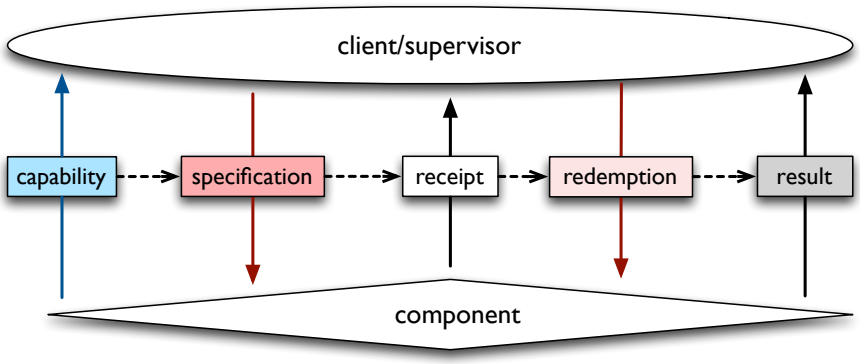

Figure 3.5: Delayed Query workflow

can be used to construct interrupts

The client can then retrieve analyzed or otherwise reduced data from the collector with a subsequent direct query.

If exceptions occur at any point, interrupts are sent to the components to cancel the indirect export, as in the subsection below.

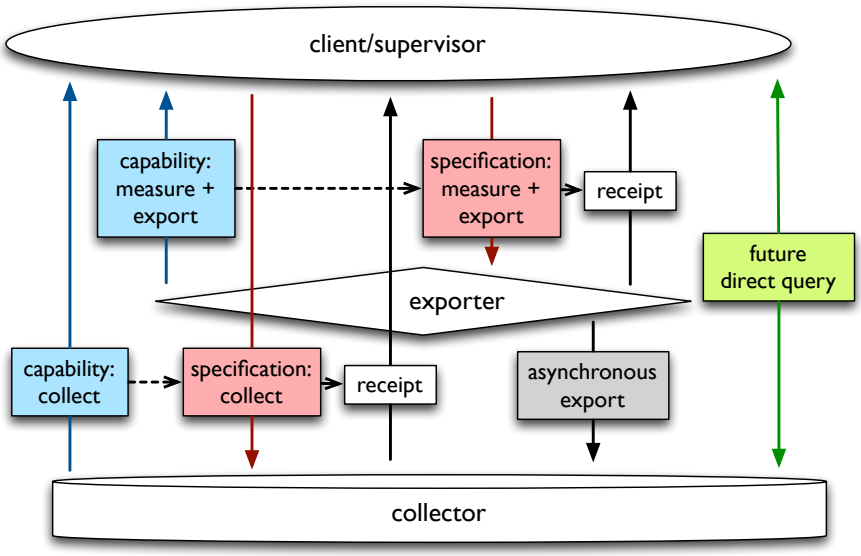

Figure 3.6: Setup of Indirect Export

3.3.4 Canceling Indirect Export

Indirect export should be set up with a specific temporal scope to automatically terminate at some point in time (see section 3.1.2.1) to avoid the problem of zombie exporters and collectors. Alternately, it may be cancelled directly by the client as shown in figure 3.6.

In this case, the interrupts derived from the receipts returned by the components are sent in reverse order.

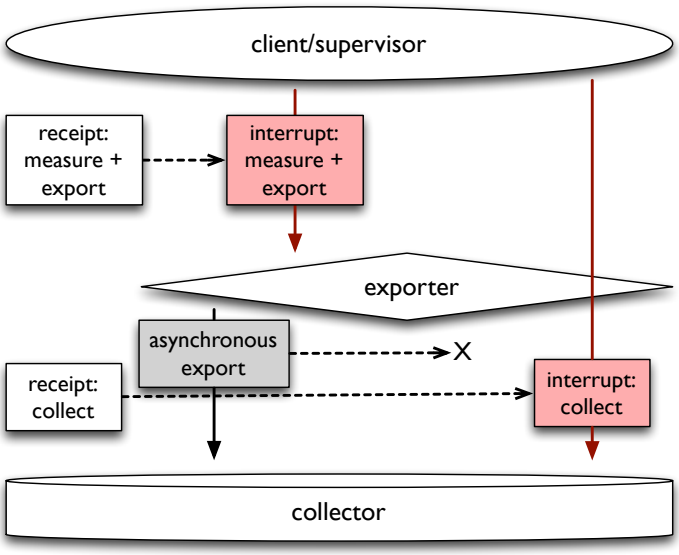

Figure 3.7: Teardown of Indirect Export

3.3.5 Notes on Message Flow Initiation and Capability Withdrawal

When a supervisor or client is directly configured with the location of a component is (e.g., in the case of well-known passive or active probes deployed as part of a network infrastructure, or most repositories), the capabilities can be made available as a static resource, and downloaded on demand by the supervisor; the supervisor can then initiate connections and send specifications to the components based on the capability. In this case, withdrawal notifications are not used: to withdraw a capability, it is simply made unavailable by the component.

On the other hand, in large-scale deployments of heterogeneous components with variable connectivity (e.g., widespread small probes on user devices and networks), the supervisor cannot discover the components. In this case, the components are configured with the location of the supervisor (e.g., though DNS SRV records), and initiate connections to the supervisor to send capabilities and retrieve specifications. In this case, the component may send a withdrawal to the supervisor to indicate a capability is no longer available; this is not mandatory, and supervisors using this message flow pattern must be robust against stale capabilities resulting from components simply disappearing.

Specific bindings to session protocols, with specifications for message flow initiation, are given in section 5.

3.3.6 Notes on Error Reporting and Recovery

Any component may signal an error to its client or supervisor at any time by sending an exception notification, as shown in 3.3. While the taxonomy of error messages is at this time left up to each individual component, exceptions should be used sparingly.

Specifically, components which initiate connections to their supervisors should not use the exception mechanism for expected error conditions (e.g., device losing connectivity for small network-edge probes) -- specifications sent to such components are expected to be best-effort. Exceptions should also not be returned for specifications which would normally not be delayed but are due to high load -- receipts should be used in this case, instead. Likewise, specifications which cannot be fulfilled because they request the use of capabilities that were once available but are no longer should be answered with withdrawals.

Exceptions should be always be returned for specifications sent to repositories which cannot be fulfilled due to a syntactic or semantic error in the specification itself.

4 mPlane Protocol Message Representations

The mPlane protocol information model is explicitly defined in a representation-neutral way. We have defined three serialization representations for turning these data elements into concrete messages for storage and transport. Each is specified for a different use case: JSON is the default representation, chosen for parseability and efficienct; YAML is intended for human readability and writability, including documentation and debugging; and XML is designed for integration with XML-based systems (e.g., a future XMPP-based transport).

4.1 JSON

JSON is the preferred representation of mPlane statements, and should be used unless there is a reason to use another representation. JSON was selected for its widespread implementation support, ease of generation and parsing, and relative efficiency of representation on the wire.

All implementations of the mPlane protocol must support the JSON representation for the purposes of interoperability.

In the JSON representation, each statement is represented as an object. Each section is represented by the section name, with the value being the section contents; additionally, the name of the statement type has the verb as a value.

The parameters section is represented as an object, mapping element name to either the parameter value (for specifications and results) or to a string representing the constraints (for capabilities; see section 4.6). The results section is represented as an array of element names. The resultvalues section, if present, is represented as an array of arrays of values in row-major order.

In addition, the JSON representation of a statement contains one entry with the name being the type of statement and the value being the verb.

Natural and real values are represented in JSON using native JSON representation for numbers. Booleans are represented by the reserved words `true` and `false`. Strings and urls are represented as JSON strings subject to JSON escaping rules. All other mPlane primitive types are represented as in section 4.5 below.

The MIME content-type (used in HTTPS protocol bindings as in section 5.1) for mPlane messages over YAML is `application/x-mplane+json`.

4.2 YAML

A YAML representation may be used in situations where human readability and writability of the statements is important. The YAML representation is directly related to the JSON representation in structure, such that a one-to-one translation between JSON and YAML structures will convert mPlane statements between the two representations.

In the YAML representation, each statement is represented as an mapping from section names to section contents, with an additional entry mapping the statement type to the verb.

The parameters section is represented as an mapping from element names to either the parameter

value (for specifications and results) or to a string representing the constraints (for capabilities; see section 4.6). The results section is represented as an list of element names. The resultvalues section, if present, is represented as an list of lists of values in row-major order.

Natural and real values are represented in YAML using native YAML representations for integers and floats, respectively. Booleans are represented by the reserved words `true` and `false`. Strings and urls are represented as YAML strings subject to YAML escaping rules. All other mPlane primitive types are represented as in section 4.5 below.

The MIME content-type (used in HTTPS protocol bindings as in section 5.1) for mPlane messages over YAML is `application/x-mplane+yaml`.

4.3 XML

An XML representation may be used for protocols designed to transport XML (e.g. XMPP), or for integration with other XML-based technologies.

The XML representation uses top-level elements for each statement type, with the verb as an attribute, containing ``parameters'', ``results'', and ``resultvalues'' sections.

The ``parameters'' element contains ``parameter'' elements, each of which has a ``name'' attribute containing the name of the mPlane element. The content of the ``parameter'' element is a string representation of either the value or the constraint as in section 4.6.

The ``results'' element contains ``result'' elements, each of which has a ``name'' attribute containing the name of the mPlane element, and no content.

The ``resultvalues'' element, if present, contains ``r'' elements; each of which contains one ``v'' element for each ``result'' element in the ``results'' element; the content of each ``v'' element is the value of the corresponding column in the row represented by its containing ``r'' element.

All mPlane types are represented as scalar string content as in section 4.5 below, subject to XML escaping rules.

The MIME content-type (used in HTTPS protocol bindings as in section 5.1) for mPlane messages over XML is `application/x-mplane+xml`.

4.4 Text-CSV

A CSV representation may be used in situations where results are stored and processed as files, and for easy integration with CSV-based tools. As of this writing, this representation remains undefined, as a future work item.

4.5 Representing Element Values

When the enclosing context (JSON, YAML, XML) does not provide a native encoding for a given mPlane primitive type, element values are represented as strings

Addresses are represented as dotted quads for IPv4 addresses as they would be in URLs, and canon-

ical IPv6 textual addresses as in section 2.2 of RFC 4291 as updated by section 4 of RFC 5952. When representing networks, addresses may be suffixed as in CIDR notation, with a `/' character followed by the mask length in bits n, provided that the least significant $32 - n$ or $128 - n$ bits of the address are zero, for IPv4 and IPv6 respectively. See sections 4.9 and 4.10 of [4] for more detail and ABNF notation.

Timestamps are represented in RFC 3339 and ISO 8601, with two important differences. First, all mPlane timestamps are are expressed in terms of UTC, so time zone offsets are neither required nor supported, and are always taken to be 0. Second, fractional seconds are represented with a variable number of digits after an optional decimal point after the fraction. See section 4.8 of [4] for more detail and ABNF notation.

4.6 Representing Parameter Constraints

In order to support the flexible representation of constraints for parameters across all encodings, parameter constraints are represented by strings. A parameter constraint string contains a set of one or more allowable values or ranges of allowable values; sets are separated by commas, and ranges by the string `...'. Ranges are only supported for natural, real, address, and timestamp types. Allowable address ranges may also be represented by CIDR notation as defined in section 4.5.

5 mPlane Session Protocol Bindings

The mPlane protocol information model and representations are designed to be session protocol independent. The services mPlane requires of its session protocol(s) include reliable message transport, mutual authentication, confidentiality, and integrity. Therefore, the mPlane protocol defines bindings to two session-layer protocols for transporting mPlane messages: HTTP and SSH. While HTTP is mandatory to implement and SSH only optional, both HTTP and SSH can be mixed in any given mPlane infrastructure.

5.1 Hypertext Transfer Protocol (HTTP) over Transport Layer Security (TLS)

The default transport protocol for mPlane messages is HTTP over TLS with mutual authentication. An mPlane component initiating a connection with another component acts as a TLS client, and must present a client certificate, which the responder will verify against its allowable peers before proceeding; likewise, the responder acts as a TLS server, and must present a server certificate, which the client will verify against its allowable peers before proceeding.

For components with simple authorization policies (e.g. most ad-hoc active probes exposing a single capability), the ability to establish a connection implies authorization to continue with any capability offered by the component. For components with more complex policies, the identity of the peer's certificate may be mapped to an internal identity on which access control decisions can be made.

Since HTTPS is not a bidirectional protocol -- clients send requests, while servers send responses -- there are various mappings between the reference message sequences (section 3.3) and HTTPS interactions to support the various deployment scenarios envisioned by mPlane. Note that in a given infrastructure of mPlane components and clients, any or all of these mappings may be used.

When sending messages over HTTP, the Content-Type of the message indicates whether the message is JSON, YAML, or XML represented. When sending exception notifications in HTTP response bodies, the response should contain an appropriate 400 (Client Error) or 500 (Server Error) response code.

5.1.1 Capability push, Specification pull

When a component knows the address of its client or supervisor (as is the case with small, ephemeral components attached to a well-known supervisor), capabilities can be POSTed to the client or supervisor, and specifications pulled via GET. The sequence here is as follows:

1. The component connects to the client via HTTPS
2. The component sends a capability statement to the client in an HTTP Request
3. The client replies 200 OK and returns an indirection notification, which includes a link from which specifications for this capability can be retrieved.

Periodically, the component attempts to retrieve specifications from the client, as follows:

1. The component connects to the client via HTTPS.
2. The component GETs the URL on the client specified in the indirection.
3. The client replies 200 OK and returns a specification statement; this may contain a link to which results should be later POSTed.
4. The component begins running the specification if it can, or POSTs an exception to the URL on the client specified in the indirection if it cannot.

If the retrieved specification is not part of an indirect export setup, the component can then later connect to the client to send its results, as follows:

1. The component connects to the client via HTTPS.
2. The component POSTs the result to the URL on the client specified in the indirection or the specification.
3. The client replies 200 OK.

5.1.2 Capability pull, Specification push

On the other hand, when a a client or supervisor knows the address(es) of its component(s) (as is the case with large probes and repositories), capabilities can be retrieved from the component via GET, and specifications POSTed. In this case, the URL(s) from which the capabilities are to be retrieved must be given to the client via a discovery mechanism, as in section 5.3. The sequence here is as follows:

1. The client connects to the component via HTTPS.
2. The client requests the capability or capabilities from the component via GET on a known URL.
3. The component replies 200 OK and returns the capability in the response body; this may contain a link to which specifications should be later POSTed.

At some point in the future, the client decides to use a capability by POSTing a specification to the component as follows:

1. The client connects to the component via HTTPS.
2. The client posts a specification the server via GET on a known URL.
3. The component replies 200 OK and returns either a receipt or a result in the response body.

Delayed query retrieval works similarly to the above, except that a redemption is presented to the component instead of a specification.

Note that since it runs over HTTPS GET, capability pull can be implemented using static resources available at the known URL.

5.1.3 Capability push, Specification push

When both the component and the client know the address of the other, these two mappings can be mixed; i.e. both capabilities can be POSTed from the component to the client, and specifications POSTed from the client to the component.

5.2 Secure Shell (SSH)

For simpler infrastructures -- specifically, to simplify key management -- the Secure Shell protocol (SSH) can also be used to transport mPlane messages. In this case, either the component or the client must have a known address, and the unknown peer initiates a SSH connection to the known peer.

In general, there is a single user associated with the mPlane endpoint on a client or component listening on SSH for mPlane messages; by default, this is `mplane`. To guarantee mututal authentication as with HTTPS over TLS, mPlane components check the presented SSH public key when deciding whether a client is authorized, or when deciding which identity or role to assign to the client; and mPlane clients check the presented SSH host public key when deciding whether a component is authorized.

Once an SSH channel is established between the component and client, either peer may initiate a message exchange with the other; the SSH channel stays connected as long as they component and client are associated, and may be reestblished by either end if it is lost.

Since SSH provides no response code to indicate errors, Exception messages must be sent in reply to any unexpected or unhandled mPlane message.

Since SSH provides no headers to identify the type of content associated with an object, mPlane components and clients may examine the first non-whitespace character in a message to determine whether it is represented in XML (`<'), JSON (`{'), or YAML (anything else.)

5.3 Component and Client Discovery

Bootstrapping an mPlane infrastructure requires a component and client discovery protocol to allow the components and clients/supervisors to find each other.

In the simplest case, components are configured with the address of a supervisor to contact. For environments where the components are expected to use a supervisor provided by the network access provider, the definition of a special DNS name (`supervisor.mplane`) allows simple autoconfiguration.

More complex environments may use DNS-SD [1] or future extensions thereto.

The project will evaluate discovery approaches during the further development of the platform; selection of protocols for component and client discovery will be a topic of the final architecture deliverable.

6 Core Type System Specification

The core type system consists of the primitives and elements used to support the basic interactions among mPlane components and clients, and which must be supported by all mPlane components. Additional Elements are defined within the scope of the scenarios to be supported by the project, and are maintained as part of the mPlane type registry and interface reference implementation, in progress as of this writing, to be drawn from the data types identified in D1.1.

6.1 Primitives

As noted in the information model terminology, mPlane supports the following primitive types, of which the elements are instances.

- string: a sequence of UTF-8 encoded characters
- natural: an unsigned integer
- real: a real, floating point number
- bool: a true or false (boolean) value
- time: a timestamp, expressed in terms of UTC
- address: an identifier of a network-level entity, including an address family
- url: a uniform resource locator

6.2 Element naming and matching rules

Elements are named as dot-separated sequences of name components; name components may contain any alphanumeric character. When mapping element names into contexts in which the dot has a special meaning, underscores may be used in place of dots.

The name components in sequence are:

- experimental tag: the tag is optional, but that must mandatory start with `x-`, the experimental tags allows to lexicographically scope basenames and provide measurement intended to be experimental in nature.
- basename: exactly one, the name of the property the element specifies or measures. All elements with the same basename have equivalent semantic meaning.
- modifier: zero or more, additional information differentiating elements with the same basename from each other. Modifiers may associate the element with a protocol layer, or a particular variety of the property named in the basename. All elements with the same basename and modifiers refer to exactly the same property. An element with the same basename and a subset of the modifiers of the other may be used as a less-specific instance of the property, depending on application requirements.

- units: zero or one, if the quantity can be measured in different units. Units additionally give information about the precision in which a quantity or timestamp is reported.

- aggregation: zero or one, if the element is derived from multiple singleton measurements. Supported aggregations are `min`, `mean`, `max`, `sum`, `NNpct` (where NN is a two-digit number 01 to 99, for percentile).

6.3 External element mappings

To the extent that external protocols define elements that are equivalent to the mPlane elements, the type registry contains mappings between the mPlane elements and the external element definitions. The prime example of an external element mapping is the mapping to the IANA IPFIX Information Element Registry defined by RFC 7012 [2] and to enterprise-specific IPFIX Information Elements required by the project.

6.4 Core elements

The core elements required to support mPlane signaling are listed below:

- `start` (timestamp): Start time of the temporal scope of the capability, specification, or result; may be expressed in units of `s` (seconds), `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds).

- `end` (timestamp): End time of the temporal scope of the capability, specification, or result; may be expressed in units of `s` (seconds), `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds).

- `period` (natural): Period of a periodic measurement; may be expressed in units of `s` (seconds), `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds).

- `source.ip` (address): Topological scope for probes which perform active measurements. Determines where the probe is and from where it can send traffic; for passive measurements, indicates the source address of an IP packet or flow. Has no units.

- `observer.ip` (address): Topological scope for probes which perform passive measurements: determines where the probe is and from which traffic it can observe. Expressed in terms of a routable network address or network prefix for which the observed link provides access to the Internet; useful only in the context of passive network border measurement. Has no units.

- `observer.link` (string): Topological scope for probes which perform passive measurements: determines where the probe is and from which traffic it can observe. Expressed in terms of link name; useful only in the context of passive network backbone measurement. Has no units.

6.5 Elements supporting reference implementation

The reference implementation can perform two basic active measurements: ping and traceroute. The additional elements required to support this are as follows:

- `destination.ip` (address): Target of an active measurement; for passive measurements, indicates the destination address of an IP packet or flow. Has no units.
- `intermediate.ip` (address): Address of an intermediate node along a path.
- `delay.twoway.icmp` (natural): Two-way (round-trip) delay measured between two endpoints measured via ICMP, as per ping. May be expressed in units of `s` (seconds), `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds).
- `delay.twoway.udp` (natural): Two-way (round-trip) delay measured between two endpoints measured via UDP, as per traceroute. may be expressed in units of `s` (seconds), `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds).
- `hops.ip` (natural): The number of IP hops associated with a given record; a location along a path from a source to a destination. Has no units.
- `hops.ip.max` (natural): The maximum number of IP hops to report; used as a parameter to traceroute. Has no units.
- `delay.twoway.udp.count` (natural): The number of two-way delay samples per record; used as a parameter to traceroute. Has no units.

6.6 Link section URL schemes

The link section in mPlane messages directs a component or client to the client or component to send the next message in the sequence to, if applicable; it is used for component and client discovery as well as redirection. The URL schemas here identify which session binding to use when connecting, one for each supported session protocol. There are two `scmplane-https` is used to identify mPlane interfaces running over HTTP over TLS as in section 5.1 and `mplane-ssh` is used to identify mPlane interfaces running over SSH as in section 5.2

Additional link section schemes are supported in capabilities in order to signal that a capability may be accessed via an external control protocol; in this case, no specifications or results are exchanged via the mPlane interfaces. This mechanism is intended primarily to advertise direct access to a repository's database. These schemes are component- and application-specific.

6.7 Export section URL schemes

The export section in mPlane statements directs a component to send (for the `measure` verb) results or to collect (for the `collect` verb) via a specified protocol at a specified URL. There are five schemas supported by the core architecture; the default indirect export protocol for interoperability is `mplane-http`.

- `mplane-http` specifies indirection via HTTP and TLS: the exporter will connect to the collector and POST mPlane result statements, expecting no reply.
- `mplane-ssh` specifies indirection via SSH: the exporter will connect to the collector and send mPlane result statements, expecting no reply.
- `ipfix-tcp` specifies IPFIX export via TCP + TLS.
- `ipfix-sctp` specifies IPFIX export via SCTP + DTLS.
- `ipfix-udp` specifies IPFIX export via UDP + DTLS.

Additional export section URL schemes are component and application specific.

7 Conclusions

This document defines the mPlane architecture and the interfaces between mPlane clients and components, based on an information model and set of serialization and session protocol bindings. It will serve as background for a reference implementation (see requirements in Appendix B) and further development of the protocol. It reflects the state of the protocol as of the time of writing; lessons learned during future development of the platform will be documented in the forthcoming final architecture specification.

A Notes on Interoperability with LMAP

We note that the architecture and protocol as defined are quite similar to the LMAP (Large-scale Measurement of Access-network Performance) effort presently under development within the IETF[1].

The primary differences arise from different requirements: LMAP is more directly focused on measurement of access network performance, as the regulatory verification scenario described in Section 3.8 of Deliverable 1.1, while mPlane has a wider set of target scenarios leading to a requirement to integrate heterogeneous measurement components across multiple scales.

However, we believe that interoperability with LMAP is a goal of mPlane, and that type-primacy and a facility for capability advertisement as provided by mPlane may be useful in an LMAP context as well; we will therefore seek opportunities to work together with the LMAP Working Group, through contribution of parts of this specification to the Working Group, and/or the definition of bindings to treat LMAP Measurement Agents as mPlane Components.

[1] `http://datatracker.ietf.org/wg/lmap/charter/`

B Reference Implementation Requirements

We have identified the following requirements for a reference implementation of the protocol specified in this document.

B.1 Component Reference Implementation Requirements

1. Serialize and deserialize statements between a Python class and a file representation
2. Allow access to capabilities stored as files in the filesystem for access via HTTP.
3. Allow posting capabilities to a supervisor or client via HTTP.
4. Allow retrieval of specifications from a supervisor or client via HTTP.
5. Allow specifications to be posted by a supervisor or client via HTTP.
6. Compare received specifications to stored capabilities to determine if they match. This involves matching the verb and types of parameters and results, as well as verifying that actual parameter values are permitted by the formal parameter values in the capability.
7. Allow unpacking of parameters from a specification; this will be used by glue code to generate configurations for probes or queries for repositories.
8. Trigger a method call on a Python object when a matching specification is received/retrieved. This is how the RI invokes the glue code that actually performs measurements on probes or queries at repositories.
9. Reusable classes for common cases for glue code generation for the previous two items.
10. Allow the packing of results into a measurement; this will be used by glue code to generate instantaneous results from on-demand queries or measurements.
11. Allow return of a receipt from the invocation of a measurement that can be used to control ongoing measurements or later retrieve results.
12. Allow a receipt or result to be returned in the same HTTP transaction in which the specification was posted.
13. Allow a receipt or result to be posted to a client or supervisor via HTTP.
14. Allow a result to be stored for later retrieval given a receipt via HTTP.
15. Accept user identity information from the security layer to pass to a repository implementation; for probe components, it is assumed that anyone permitted by the security layer to access the probe has full access, at least in the initial revision.

B.2 Additional Supervisor Reference Implementation Requirements

1. Allow retrieval of capabilities from subordinate components.
2. Allow subordinate components to post capabilities via HTTP.
3. Allow subordinate components to revoke capabilities.
4. Trigger a method call on a Python object when a capability is received/retrieved from a subordinate component, or revoked by a subordinate component. This method will use knowledge of the specific measurement domain to determine what the capabilities of the supervisor are.
5. Trigger a method call on a Python object when a specification is received/retrieved; this code should map this specification to lower level specifications to be sent to subordinate components.
6. Trigger a method call on a Python object when a result/receipt is received/retrieved; this code should aggregate or otherwise compose results into a higher-level result.

Note that, as decomposition/recomposition of measurement problems are domain-specific, these are out of scope for the RI. The RI will probably contain some stub logic here to simply act as a single probe proxy.

References

[1] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), Sept. 2013.

[2] B. Claise and B. Trammell. Information Model for IP Flow Information Export (IPFIX). RFC 7012 (Proposed Standard), Sept. 2013.

[3] B. Claise, B. Trammell, and P. Aitken. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Flow Information. RFC 7011 (Internet Standard), Sept. 2013.

[4] B. Trammell. Textual Representation of IPFIX Abstract Data Types. IETF Internet-Draft draft-trammell-ipfix-text-adt-02 (work in progress), July 2013.