mPlane Protocol RI Documentation

Release 0.9.0

Brian Trammell

February 18, 2016

Contents

This module provides software development kit for building applications on top of the mPlane protocol. It is organized into serval modules:

mplane.model implements the mPlane protocol information model: message types, the element registry, and various support classes. On top of the information model, the *mplane.scheduler* module defines a framework for binding *mplane.model.Capability* classes to runnable code, and for invoking that code on the receipt of mPlane Statements; this is used to build clients and components.

The *mplane.client* module defines interfaces for building clients; the `mpcli` script provides a simple command-line interface to this client.

The *mplane.component* module defines interfaces for building components; the component runtime can be started by running the `mpcom` script.

CHAPTER 1

module mplane.model

Information model and element registry for the mPlane protocol.

This module implements Statements and Notifications, the core messages used by the mPlane protocol to describe measurement and query schemas, and various other classes to support them.

There are three kinds of Statements:

- Capability represents something a component can do
- Specification tells a component to do something it advertised a Capability for
- Result returns the results for a Specification in-band

Notifications are used to transfer other information between components and clients. There are four kinds of Notifications:

- Receipt notifies that a Result is not yet ready or that the results of an operation will be indirectly exported.
- Redemption is a subsequent attempt to redeem a Receipt.
- Withdrawal notifies that a Capability is no longer available.
- Interrupt notifies that a running Specification should be stopped.

To see how all this fits together, let's simulate the message exchange in a simple ping measurement. Initially, we have to load the default element registry and programatically create a new empty Capability, as it would be advertised by the component.

```
>>> import mplane
>>> import json
>>> mplane.model.initialize_registry()
>>> cap = mplane.model.Capability()
```

First, we set a temporal scope for the capability. Probe components generally advertise a temporal scope from the present stretching into the indeterminate future. In this case, we advertise that the measurement performed is periodic, by setting the minimum period supported by the capability: one ping per second.

```
>>> cap.set_when("now ... future / 1s")
```

We can only ping from one IPv4 address, to any IPv4 address. Adding a parameter without a constraint makes it unconstrained:

```
>>> cap.add_parameter("source.ip4", "10.0.27.2")
>>> cap.add_parameter("destination.ip4")
```

Then we define the result columns this measurement can produce. Here, we want quick reporting of min, max, and mean delays, as well as a total count of singleton measurements taken and packets lost:

```
>>> cap.add_result_column("delay.twoway.icmp.us.min")
>>> cap.add_result_column("delay.twoway.icmp.us.max")
>>> cap.add_result_column("delay.twoway.icmp.us.mean")
>>> cap.add_result_column("delay.twoway.icmp.count")
>>> cap.add_result_column("packets.lost")
```

Now we have a capability we could transform into JSON and make it available to clients via the mPlane protocol, or via static download or configuration:

```
>>> capjson = mplane.model.unparse_json(cap)
>>> capjson
'{"capability": "measure",
  "version": 1,
  "registry": "http://ict-mplane.eu/registry/core",
  "when": "now ... future / 1s",
  "parameters": {"source.ip4": "10.0.27.2",
                 "destination.ip4": "*"},
  "results": ["delay.twoway.icmp.us.min",
              "delay.twoway.icmp.us.max",
              "delay.twoway.icmp.us.mean",
              "delay.twoway.icmp.count",
              "packets.lost"]}'
```

On the client side, we'd receive this capability as a JSON object and turn it into a capability, from which we generate a specification:

```
>>> clicap = mplane.model.parse_json(capjson)
>>> spec = mplane.model.Specification(capability=clicap)
>>> spec
<specification: measure when now ... future / 1s token e00d7fe8 schema 5ce99352 p(v)/m/r 2(1)/0/5>
```

Here we have a specification with a given token, schema, and 2 parameters, no metadata, and five result columns.

Note: The schema of the statement is identified by a schema hash, the first eight hex digits of which are shown for diagnostic purposes. Statements with identical sets of parameters and columns (schemas) will have identical schema hashes. Likewise, the token is defined by the schema as well as the parameter values.

First, let's fill in a specific temporal scope for the measurement:

```
>>> spec.set_when("2017-12-24 22:18:42 + 1m / 1s")
```

And then let's fill in some parameters. All od the parameters whose single values are already given by their constraints (in this case, source.ip4) have already been filled in. So let's start with the destination. Note that strings are accepted and automatically parsed using each parameter's primitive type:

```
>>> spec.set_parameter_value("destination.ip4", "10.0.37.2")
```

And now we can transform this specification and send it back to the component from which we got the capability:

```
>>> specjson = mplane.model.unparse_json(spec)
>>> specjson
'{"specification": "measure",
  "version": 1,
  "registry": "http://ict-mplane.eu/registry/core",
  "token": "ea839b56bc3f6004e95d780d7a64d899",
  "when": "2017-12-24 22:18:42.000000 + 1m / 1s",
  "parameters": {"source.ip4": "10.0.27.2",
                 "destination.ip4": "10.0.37.2"},
```

```
  "results": ["delay.twoway.icmp.us.min",
              "delay.twoway.icmp.us.max",
              "delay.twoway.icmp.us.mean",
              "delay.twoway.icmp.count",
              "packets.lost"]}'
```

On the component side, likewise, we'd receive this specification as a JSON object and turn it back into a specification:

```
>>> comspec = mplane.model.parse_json(specjson)
```

The component would determine the measurement, query, or other operation to run given by the specification, then extract the necessary parameter values, e.g.:

```
>>> comspec.get_parameter_value("destination.ip4")
IPv4Address('10.0.37.2')
>>> comspec.when().period()
datetime.timedelta(0, 1)
```

After running the measurement, the component would return the results by assigning values to parameters which changed and result columns measured:

```
>>> res = mplane.model.Result(specification=comspec)
>>> res.set_when("2017-12-24 22:18:42.993000 ... 2017-12-24 22:19:42.991000")
>>> res.set_result_value("delay.twoway.icmp.us.min", 33155)
>>> res.set_result_value("delay.twoway.icmp.us.mean", 55166)
>>> res.set_result_value("delay.twoway.icmp.us.max", 192307)
>>> res.set_result_value("delay.twoway.icmp.count", 58220)
>>> res.set_result_value("packets.lost", 2)
```

The result can then be serialized and sent back to the client:

```
>>> resjson = mplane.model.unparse_json(res)
>>> resjson
'{"result": "measure",
  "version": 1,
  "registry": "http://ict-mplane.eu/registry/core",
  "token": "ea839b56bc3f6004e95d780d7a64d899",
  "when": "2017-12-24 22:18:42.993000 ... 2017-12-24 22:19:42.991000",
  "parameters": {"source.ip4": "10.0.27.2",
                 "destination.ip4": "10.0.37.2"},
  "results": ["delay.twoway.icmp.us.min",
              "delay.twoway.icmp.us.max",
              "delay.twoway.icmp.us.mean",
              "delay.twoway.icmp.count",
              "packets.lost"],
  "resultvalues": [["33155", "192307", "55166", "58220", "2"]]}'
```

which can transform them back to a result and extract the values:

```
>>> clires = mplane.model.parse_json(resjson)
>>> clires
<result: measure when 2017-12-24 22:18:42.993000 ... 2017-12-24 22:19:42.991000 token e00d7fe8 schema
```

If the component cannot return results immediately (for example, because the measurement will take some time), it can return a receipt instead:

```
>>> rcpt = mplane.model.Receipt(specification=comspec)
```

This receipt contains all the information in the specification, as well as a token which can be used to quickly identify it in the future.

```
>>> rcpt.get_token()
'e00d7fe813cf17eeeea37b313dcfa4e7'
```

Note: The mPlane protocol specification allows components to assign tokens however they like. In the reference implementation, the default token is based on a hash like the schema hash: statements with the same verb, schema, parameter values, and metadata will have identical default tokens. A component could, however, assign serial-number based tokens, or tokens mapping to structures in its own filesystem, etc.

```
>>> jsonrcpt = mplane.model.unparse_json(rcpt)
>>> jsonrcpt
'{"receipt": "measure",
  "version": 1,
  "registry": "http://ict-mplane.eu/registry/core",
  "token": "e00d7fe813cf17eeeea37b313dcfa4e7",
  "when": "2017-12-24 22:18:42.000000 + 1m / 1s",
  "parameters": {"destination.ip4": "10.0.37.2",
                 "source.ip4": "10.0.27.2"},
  "results": ["delay.twoway.icmp.us.min",
              "delay.twoway.icmp.us.max",
              "delay.twoway.icmp.us.mean",
              "delay.twoway.icmp.count",
              "packets.lost"],}'
```

The component keeps the receipt, keyed by token, and returns it to the client in a message. The client then which generates a future redemption referring to this receipt to retrieve the results:

```
>>> clircpt = mplane.model.parse_json(jsonrcpt)
>>> clircpt
<receipt: e00d7fe813cf17eeeea37b313dcfa4e7>
>>> rdpt = mplane.model.Redemption(receipt=clircpt)
>>> rdpt
<redemption: e00d7fe813cf17eeeea37b313dcfa4e7>
```

Note here that the redemption has the same token as the receipt; just the token may be sent back to the component to retrieve the results:

```
>>> mplane.model.unparse_json(rdpt, token_only=True)
'{"redemption": "measure",
  "version": 1,
  "registry": "http://ict-mplane.eu/registry/core",
  "token": "e00d7fe813cf17eeeea37b313dcfa4e7"
}'
```

As long as the measurement is running, the client can stop the measurement by sending an interrupt:

```
>>> irpt = mplane.model.Interrupt(specification=spec)
>>> jsonrcpt = mplane.model.unparse_json(irpt)
```

The component receives the interrupt, stops the measurement and returns the results of performed measurement.

Otherwise, in case the component cannot perform the speficied operation, it sends a withdrawal to cancel the previously advertised capability:

```
>>> wtdr = mplane.model.Withdrawal(capability=cap)
>>> mplane.model.unparse_json(wtdr)
'{"withdrawal": "measure",
"version": 1,
```

```
"registry": "http://ict-mplane.eu/registry/core",
"token": "d7e9df75145e209e144bf9c06e7a9d2f",
"when": "now ... future / 1s",
"parameters": {"destination.ip4": "*",
"source.ip4": "10.0.27.2"},
"results": ["delay.twoway.icmp.us.min",
"delay.twoway.icmp.us.max",
"delay.twoway.icmp.us.mean",
"delay.twoway.icmp.count",
"packets.lost"]
}'
```

Further several messages can be send at once by using an Envelope, e.g. a component could announce serval capabilites at once.

In case of our simple ping component we create a second capability that only provides the mean of the measurement values:

```
>>> cap2 = mplane.model.Capability()
>>> cap2.set_when("now ... future / 1s")
>>> cap2.add_parameter("source.ip4", "10.0.27.2")
>>> cap2.add_parameter("destination.ip4")
>>> cap2.add_result_column("delay.twoway.icmp.us.mean")
```

Now we create an Envelope and append the two capabilities.

```
>>> env = mplane.model.Envelope()
>>> env.append_message(cap)
>>> env.append_message(cap2)
>>> env
<Envelope message (2):
 <capability: measure when now ... future / 1s token d7e9df75 schema 5ce99352 p/m/r 2/0/5>
 <capability: measure when now ... future / 1s token a9ec7fce schema ea37cea5 p/m/r 2/0/1>
>
```

Similar as with every other message this Envelope is serialized and send to the client:

```
>>> envjson = mplane.model.unparse_json(env)
```

The client receives the Envelope and decomposes the encapsulated messages:

```
>>> clienv = mplane.model.parse_json(envjson)
>>> messages = [message for message in clienv.messages()]
```

class mplane.model.**BareNotification**(*dictval=None*, *token=None*)

Notifications are used to send additional information between mPlane clients and components other than measurement statements. Notifications can either be part of a normal measurement workflow (as Receipts and Redemptions) or signal exceptional conditions (as Withdrawals and Interrupts).

This class contains implementation common to all Notifications which do not contain any information from a related Capability or Specification.

class mplane.model.**Capability**(*dictval=None*, *verb='measure'*, *label=None*, *token=None*, *when=None*, *registry_uri=None*)

A Capability represents something an mPlane component can do. Capabilities contain verbs (strings identifying the thing the component can do), parameters (which must be given by a client in a Specification in order for the component to do that thing), metadata (additional information about the process used to do that thing), and result columns (the data that thing will return).

Capabilities can either be created programatically, using the add_parameter(), add_metadata(), and add_result_column() methods, or by reading from a JSON object using parse_json().

set_when(*when*, *force=True*)
By default, changes to capability temporal scopes are always forced.

validate()
Checks that this is a valid Capability; i.e., capabilites can not parameter nor results values.

class mplane.model.**Element**(*name*, *prim*, *desc=None*, *namespace='http://ict-mplane.eu/registry/core'*)
An Element represents a name for a particular type of data with a specific semantic meaning; it is analogous to an IPFIX Information Element, or a named column in a relational database.

An Element has a Name by which it can be compared to other Elements, and a primitive type, which it uses to convert values to and from strings.

The mPlane reference implementation includes a default registry of elements; use initialize_registry() to use these.

compatible_with(*rval*)
Determines based on naming rules if this element is compatible with element rval; that is, if transformation_to will return a function for turning a value of this element to the other. Compatibility based on name structure is a future feature; this method currently checks for name equality only.

desc()
Returns the description of this Element

name()
Returns the name of this Element

parse(*sval*)
Converts a string to a value for this Element; delegates to primitive.

primitive_name()
Returns the name of this Element's primitive

qualified_name()
Returns the name of this Element along with its namespace

transformation_to(*rval*)
Returns a function which will transform values of this element into values of element rval; used to support unit conversions. This is a future feature, and is currently a no-op. Only valid if compatible_with returns True.

unparse(*val*)
Converts a value to a string for this Element; delegates to primitive.

class mplane.model.**Envelope**(*dictval=None*, *content_type='message'*, *token=None*, *label=None*, *when=None*)
Envelopes are used to contain other Messages.

append_message(*msg*)
Appends a message to an Envelope

get_label()
Returns the label or None if no label has been set

get_token(*lim=None*)
Returns the token or None if no token has been set

messages()
Returns an iterator to iterate over all messages in an Envelope

trim(*n*)
Removes everything except the last n elements

when()
Returns the envelope's temporal scope. (If it's a bunch of multijob results)

class mplane.model.**Exception**(*token=None*, *dictval=None*, *errmsg=None*, *status=None*)
A Component sends an Exception to a Client, or a Client to a Component, to present a human-readable message about a failure or non-nominal condition.

The status field is used to store an HTTP status code corresponding to the exception to the client and component frameworks.

get_token()
Returns a token that originates from a message that has caused the Expection or None if the token was explictly not set

class mplane.model.**Interrupt**(*dictval=None*, *specification=None*, *token=None*)
An Interrupt cancels a Specification

validate()
Checks that this is a valid Interrupt; performes the same checks as for a Specification.

class mplane.model.**Metavalue**(*parent_element*, *val*)
A Metavalue is an element which can take an unconstrained value. Metavalues are used in statement metadata sections.

get_value()
Returns the value

set_value(*val*)
Sets the value. If the value is a string it parses it.

class mplane.model.**Parameter**(*parent_element*, *constraint=mplane.model.constraint_all*, *val=None*)
A Parameter is an element which can take a constraint and a value. In Capabilities, Parameters have constraints and no value; in Specifications and Results, Parameters have both constraints and values.

can_set_value(*val*)
Returns True if the parameter can take the specified value, False otherwise. Either takes a list of values or a single value of the correct type for the associated Primitive, or a list of strings or a single string, which will be parsed to the correct type.

get_single_value()
If this parameter's Constraint only allows a single value, returns it

get_value()
Returns this Parameter's value

has_value()
Returns True if this component has a value.

is_single_value()
Returns True if this parameter's Constraint only allows a single value

set_single_value()
If this Parameter's Constraint allows only a single value, and this Paramater does not yet have a value, set the value to the only one allowed by the Constraint.

set_value(*val*)
Sets the value of the Parameter. Either takes a list of values or a single value of the correct type for the associated Primitive, or a list of strings or a single string, which will be parsed to the correct type.

Raises ValueError if the value is not allowable for the Constraint.

class mplane.model.**Receipt**(*dictval=None*, *specification=None*, *token=None*)
A component presents a receipt to a Client in lieu of a result, when the result will not be available in a reasonable amount of time; or to confirm a Specification

validate()
Checks that this is a valid Receipt; performes the same checks as for a Specification.

class mplane.model.**Redemption**(*dictval=None*, *receipt=None*, *token=None*)
A client presents a Redemption to a component from which it has received a Receipt in order to get the associated Result.

validate()
Checks that this is a valid Redemption; performes the same checks as for a Specification.

class mplane.model.**Registry**(*uri=None*, *filename=None*, *noparse=False*)
A Registry is a collection of named Elements associated with a namespace URI, from which it is retrieved.

uri()
Returns the URI by which this registry is known.

class mplane.model.**Result**(*dictval=None*, *specification=None*, *verb='measure'*, *label=None*, *token=None*, *when=None*)
A result is a statement that a component measured a given set of values at a given point in time, according to a specification.

Results are generally created by passing the specification the new result responds to as the specification= argument to the constructor. A result inherits its token from the specification it responds to.

schema_dict_iterator()
Iterates over each row in this result, yielding a dictionary mapping all parameter and result column names to their values.

set_result_value(*elem_name*, *val*, *row_index=0*)
Sets a single result value.

validate()
Checks that this is a valid Result; i.e., that all parameters have values.

class mplane.model.**ResultColumn**(*parent_element*)
A ResultColumn is an element which can take an array of values. In Capabilities and Specifications, this array is empty, while in Results it has one or more values, such that all the ResultColumns in the Result have the same number of values.

clear()
Clears values.

class mplane.model.**Specification**(*dictval=None*, *capability=None*, *verb='measure'*, *label=None*, *token=None*, *when=None*, *schedule=None*)
A Specification represents a request for an mPlane component to do something it has advertised in a Capability. Capabilities contain verbs (strings identifying the thing the component can do), parameters (which must be given by a client in a Specification in order for the component to do that thing), metadata (additional information about the process used to do that thing), and result columns (the data that thing will return).

Specifications are created either by passing a Capability the Specification is intended to use as the capability= argument of the constructor, or by reading from a JSON object (see model.parse_json()).

fulfills(*capability*)
Returns True if this Specification fulfills the Capability A specification fulfills a capability if the schemas match, if the temporal scope of the specification is covered by that of the capability, and if the specification's parameter values meet the capability's parameter constraints.

is_schedulable()
Determine if a specification can be scheduled – i.e., that its temporal scope refers to some time in the future at which a measurement or other operation should take place, or some range of time for which existing data should be searched and/or retrieved.

Currently, this just checks to see whether the verb is 'query'.

retoken(*force=False*)
Generates a new token, if necessary, taking into account the current time if a specification has a relative temporal scope.

subspec_iterator()
Iterates over subordinate specifications if this specification is repeated (i.e., has a repeated Temporal Scope); otherwise yields self once. Each subordinate specification has an absolute temporal scope derived from this specification's relative temporal scope and schedule.

validate()
Checks that this is a valid Specification; i.e., that all parameters have values.

class mplane.model.**Statement**(*dictval=None*, *verb='measure'*, *label=None*, *token=None*, *when=None*, *reguri=None*)
A Statement is an assertion about the properties of a measurement or other action performed by an mPlane component. This class contains common implementation for the three kinds of mPlane statement. Clients and components should use the *mplane.model.Capability*, *mplane.model.Specification*, and *mplane.model.Result* classes instead.

add_metadata(*elem_name*, *val*)
Programatically adds a metadata element to this Statement.

add_parameter(*elem_name*, *constraint=mplane.model.constraint_all*, *val=None*)
Programatically adds a parameter to this Statement.

add_result_column(*elem_name*)
Programatically adds a result column to this Statement.

can_set_parameter_value(*elem_name*, *value*)
Determines whether a given Parameter can take a value.

count_metadata()
Returns the number of metavalues in this Statement.

count_parameter_values()
Returns the number of parameters with values in this Statement.

count_parameters()
Returns the number of parameters in this Statement.

count_result_columns()
Returns the number of result columns in this Statement.

count_result_rows()
Returns the number of result rows in this Statement.

get_export()
Returns the Statement's export URL, which specifies where results will be indirectly exported.

get_label()
Returns the Statement's label.

get_link()
Returns the statement's link URL, which specifies where the next message in the workflow should be sent to or retrieved from.

get_parameter_value(*elem_name*)
Returns the value for a named parameter on this Statement.

get_single_parameter_value(*elem_name*)
If a given parameter is single-valued returns that value, otherwise returns None

get_token(*lim=None*)
Returns the token of a Statement. If a token has not been explicitely set, it returns the default token for the Statement type.

has_metadata(*elem_name*)
Returns True if the statement has a metadata element with the given name.

has_parameter(*elem_name*)
Returns True if the statement has a parameter with the given name.

has_result_column(*elem_name*)
Returns True if the statement has results column with the given name.

metadata_names()
Iterates over the names of metadata elements in this Statement.

parameter_names()
Iterates over the names of parameters in this Statement.

parameter_values()
Returns a dict mapping parameter names to values for each parameter with a value.

result_column_names()
Iterates over the names of result columns in this Statement.

set_export(*export*)
Sets the Statement's export URL.

set_label(*label*)
Sets the statement's label

set_link(*link*)
Sets the Statement's link URL

set_parameter_value(*elem_name*, *value*)
Programatically sets a value for a parameter on this Statement.

set_when(*when*, *force=False*)
Sets the statement's temporal scope. Ensures that the temporal scope is within the previous temporal scope unless force is True. Takes either an instance of mplane.model.When, or a string describing the scope.

to_dict(*token_only=False*)
Converts a Statement to a dictionary (for further conversion to JSON or YAML), which can be passed as the dictval argument of the appropriate statement constructor.

verb()
Returns this statement's verb

when()
Returns the statement's temporal scope.

class mplane.model.**When**(*valstr=None*, *a=None*, *b=None*, *duration=None*, *period=None*, *repeated=False*, *inner_duration=None*, *inner_period=None*, *crontab=None*)
Defines the temporal scopes for capabilities, results, or single measurement specifications.

datetimes(*tzero=None*)
Return start and end times as absolute timestamps for this temporal scope, relative to a given tzero.

duration(*tzero=None*)
Return the duration of this temporal scope as a timedelta.

If the temporal scope is indefinite in the future, returns None.

follows(*s*, *tzero=None*)
Returns True if this scope follows (is contained by) another.

in_scope(*t*, *tzero=None*)
Returns True if time t falls within this scope.

is_definite()
Returns True if this scope defines a definite time or a definite time interval.

is_forever()
Returns True if this scope ends in the indeterminate future.

is_future()
Returns True if this is an indefinite future scope.

is_immediate()
Returns True if this is an immediate scope (i.e., starts now).

is_infinite()
Returns True if this scope is completely infinite (from the infinite past to the infinite future).

is_past()
Returns True if this is an indefinite past scope.

is_repeated()
Return True if this temporal scope referes to a repeated when.

is_singleton()
Returns True if this temporal scope refers to a singleton measurement. Used in scheduling an enclosing Specification; has no meaning for Capabilities or Results.

iterator(*tzero=None*)
Returns an iterator over When statements generated by a repeated when.

period()
Returns the period of this temporal scope.

sort_scope(*t*, *tzero=None*)
Returns < 0 if time t falls before this scope, 0 if time t falls within the scope, or > 0 if time t falls after this scope.

timer_delays(*tzero=None*)
Returns a tuple with delays for timers to signal the start and end of a temporal scope, given a specified time zero, which defaults to the current system time.

The start delay is defined to be zero if the scheduled start time has already passed or the temporal scope is immediate (i.e., starts now). The start delay is None if the temporal scope has expired (that is, the current time is after the calculated end time).

The end delay is defined to be None if the temporal scope has already expired, or if the temporal scope has no scheduled end (is infinite or a singleton). End delays are calculated to give priority to duration when a temporal scope is expressed in terms of duration, and to prioritize end time otherwise.

Used in scheduling an enclosing Specification for execution. Has no meaning for Capabilities or Results.

class mplane.model.**Withdrawal**(*dictval=None*, *capability=None*, *token=None*)
A Withdrawal cancels a Capability

validate()
Checks that this is a valid Withdrawal; performes the same checks as for a Capability.

mplane.model.**element**(*name*, *reguri=None*)
Returns the Element with the given name. If reguri is given, searches the speficied Registry, otherwise searches the base Registry.

mplane.model.**initialize_registry**(*uri=None*)
Initializes the mPlane registry from a URI; if no URI is given, initializes the registry from the internal core registry.

Call this after preloading registries, but before doing anything else.

mplane.model.**message_from_dict**(*d*)
Given a dictionary returned from to_dict(), return a decoded mPlane message (statement or notification).

mplane.model.**parse_constraint**(*prim*, *sval*)
Given a primitive and a string value, parses a constraint string (returned via str(constraint)) into an instance of an appropriate constraint class.

mplane.model.**parse_json**(*jstr*)
Parse a JSON object in a string and return the associated mPlane message.

mplane.model.**registry_for_uri**(*uri*)
Get a registry for a given URI, maintaining a local cache. Called when parsing statements; generally not useful in client code.

mplane.model.**unparse_json**(*msg*, *token_only=False*)
Transform an mPlane message into a JSON object representing it. If token_only is True, uses tokens only for message types for which that is appropriate (i.e. Reciepts, Redemptions, Withdrawals, and Interrupts).

CHAPTER 2

module mplane.scheduler

Implements the dynamics of capabilities, specifications, and results within the mPlane reference component.

class mplane.scheduler.**Job**(*service*, *specification*, *session=None*, *callback=None*)

A Job binds some running code to an mPlane.model.Specification within a component. A Job can be thought of as a specific instance of a Service presently running, or ready to run at some point in the future.

Each Job will result in a single Result.

failed()

A job only fails if it is finished and has no results

finished()

Return False if the job is not complete and if there are results pending. Otherwise return True

get_reply()

If a result is available for this Job (i.e., if the job is done running), return it. Otherwise, create a receipt from the Specification and return that.

interrupt()

Interrupt this job.

schedule()

Schedule this job to run.

class mplane.scheduler.**MultiJob**(*service*, *specification*, *session=None*, *max_results=0*, *callback=None*)

A MultiJob spawns multiple jobs determined by its schedule.

Each MultiJob will result in multiple result rows, one for each sub-job.

failed()

A multijob will only fail if it is finished and has no results

finished()

Return True if all jobs are complete.

get_reply()

If results are available for this MultiJob, return them. Otherwise, create a receipt from the Specification and return that.

interrupt()

Interrupt all jobs.

schedule()

Scheduling start function

class mplane.scheduler.**Scheduler**(*config=None*)

Scheduler implements the common runtime of a Component within the reference implementation. Components register Services bound to Capabilities with add_service(), and submit jobs for scheduling using submit_job().

add_service(*service*)

Add a service to this Scheduler

capability_for_key(*key*)

Return a capability for a given key.

capability_keys()

Return keys (tokens) for the set of cached capabilities provided by this scheduler's services.

job_for_message(*msg*)

Given a message (generally a Redemption), return the Job matching its token.

process_message(*user*, *msg*, *session=None*, *callback=None*)

Process a message. If msg is a mplane.model.Specification and the callback parameter is set to a function this function is called with a mplane.model.Receipt each time a result is available.

Returns a message to send in reply.

prune_jobs()

Currently does nothing. Will remove Jobs which are finished and whose Results have been retrieved from the scheduler in a future version.

remove_service(*service*)

Remove a service from this Scheduler

submit_job(*user*, *specification*, *session=None*, *callback=None*)

Search the available Services for one which can service the given Specification, then create and schedule a new Job to execute the statement.

class mplane.scheduler.**Service**(*capability*)

A Service binds some runnable code to an mplane.model.Capability provided by a component.

To use services with an mPlane scheduler, inherit from mplane.scheduler.Service or one of its subclasses and implement run().

capability()

Returns the capability belonging to this service

run(*specification*, *check_interrupt*)

Run this service given a specification which matches the capability. This is called by the scheduler, and should be implemented by a concrete subclass of Service.

The implementation should extract its parameters from a given mplane.model.Specification, and return its result values in a mplane.model.Result derived therefrom.

After each row or logically grouped set of rows, the implementation should call the check_interrupt function to determine whether it should stop; if this function returns True, the implementation should terminate its processing in an orderly fashion and return its results.

Each method will be called within its own thread and/or process.

set_capability_link(*link*)

Sets the capability's link section, if it is not already set

CHAPTER 3

module mplane.client

class mplane.client.**BaseClient**(*tls_state*, *supervisor=False*, *exporter=None*)

Core implementation of a generic programmatic client. Used for common client state management between HttpInitiatorClient and HttpListenerClient; use one of these instead.

capabilities_matching_schema(*schema_capability*)

Given a capability, return *all* known capabilities matching the given schema capability. A capability matches a schema capability if and only if: (1) the capability schemas match and (2) all constraints in the capability are contained by all constraints in the schema capability.

Used to programmatically select capabilities matching an aggregation or other collection operation (e.g. at a supervisor).

capability_for(*token_or_label*)

Retrieve a capability given a token or label.

capability_labels()

list all labels for stored capabilities

capability_tokens()

list all tokens for stored capabilities

forget(*token_or_label*)

forget all receipts and results for the given token or label

handle_message(*msg*, *identity=None*)

Handle a message. Used internally to process mPlane messages received from a component. Can also be used to inject messages into a client's state.

identity_for(*token_or_label*, *receipt=False*)

Retrieve an identity given a capability token or label, or a receipt token.

receipt_labels()

list all labels for outstanding receipts

receipt_tokens()

list all tokens for outstanding receipts

result_for(*token_or_label*)

Return a result for the token if available; return the receipt for the token otherwise.

result_labels()

list all labels for stored results

result_tokens()

list all tokens for stored results

class mplane.client.**CrawlParser**(***kwargs*)
HTML parser class to extract all URLS in a href attributes in an HTML page. Used to extract links to Capabilities exposed as link collections.

class mplane.client.**HttpInitiatorClient**(*tls_state*, *default_url=None*, *supervisor=False*, *exporter=None*)
Core implementation of an mPlane JSON-over-HTTP(S) client. Supports client-initiated workflows. Intended for building client UIs and bots.

invoke_capability(*cap_tol*, *when*, *params*, *relabel=None*)
Given a capability token or label, a temporal scope, a dictionary of parameters, and an optional new label, derive a specification and send it to the appropriate destination.

result_for(*token_or_label*)
return a result for the token if available; attempt to redeem the receipt for the token otherwise; if not yet redeemable, return the receipt instead.

retrieve_capabilities(*url*, *urlchain=[]*, *pool=None*, *identity=None*)
Connect to the given URL, retrieve and process the capabilities/withdrawals found there

send_message(*msg*, *dst_url=None*)
send a message, store any result in client state.

class mplane.client.**HttpListenerClient**(*config*, *tls_state=None*, *supervisor=False*, *exporter=None*, *io_loop=None*)
Core implementation of an mPlane JSON-over-HTTP(S) client. Supports component-initiated workflows. Intended for building supervisors.

invoke_capability(*cap_tol*, *when*, *params*, *relabel=None*, *callback_when=None*)
Given a capability token or label, a temporal scope, a dictionary of parameters, and an optional new label, derive a specification and queue it for retrieval by the appropriate identity (i.e., the one associated with the capability).

If the identity has indicated it supports callback control, the optional callback_when parameter queues a callback spec to schedule the next callback.

listen_in_background(*io_loop=None*)
The server listens for requests in background

class mplane.client.**InteractionsHandler**(*application*, *request*, ***kwargs*)
Handles the probes that want to register to this supervisor Each capability is registered independently

Exposes the specifications, that will be periodically pulled by the components

Receives results of specifications

generate_response(*env*)
Generate the response for the request containing the capabilities

get()
Receives GET specification requests

post()
Receives POST requests that may contain Capabilities, Results, Receipts and Exceptions

class mplane.client.**MPlaneHandler**(*application*, *request*, ***kwargs*)
Abstract tornado RequestHandler that allows a handler to respond with an mPlane Message.

CHAPTER 4

module mplane.component

class mplane.component.**DiscoveryHandler**(*application*, *request*, ***kwargs*)
Exposes the capabilities registered with a given scheduler. URIs ending with "capability" will result in an HTML page listing links to each capability.

class mplane.component.**MPlaneHandler**(*application*, *request*, ***kwargs*)
Abstract tornado RequestHandler that allows a handler to respond with an mPlane Message or an Exception.

class mplane.component.**MessagePostHandler**(*application*, *request*, ***kwargs*)
Receives mPlane messages POSTed from a client, and passes them to a scheduler for processing. After waiting for a specified delay to see if a Result is immediately available, returns a receipt for future redemption.

CHAPTER 5

mpcli command-line client

The mPlane Client Shell is a simple client intended for debugging of mPlane infrastructures. To start it, simply run `mpcli`. It supports the following commands:

- `seturl`: Set the default URL for sending specifications and redemptions (when not given in a Capability's or Receipt's link section)
- `getcap`: Retrieve capabilities and withdrawals from a given URL, and process them.
- `listcap`: List available capabilities
- `showcap`: Show the details of a capability given its label or token
- `when`: Set the temporal scope for a subsequent *runcap* command
- `set`: Set a default parameter value for a subsequent *runcap* command
- `unset`: Unset a previously set default parameter value
- `show`: Show a previously set default parameter value
- `runcap`: Run a capability given its label or token
- `listmeas`: List known measurements (receipts and results)
- `showmeas`: Show the details of a measurement given its label or token.
- `tbenable`: Enable tracebacks for subsequent exceptions. Used for client debugging.

CHAPTER 6

mpcom component runtime

The component runtime provides a framework for building components for both component-initiated and client-initiated workflows. To implement a component for use with this framework:

- Implement each measurement, query, or other action performed by the component as a subclass of *mplane.scheduler.Service*. Each service is bound to a single capability. Your service must implement at least *mplane.scheduler.Service.run()*.
- Implement a `services` function in your module that takes a set of keyword arguments derived from the configuration file section, and returns a list of Services provided by your component. For example:

```
def service(**kwargs):
    return [MyFirstService(kwargs['local-ip-address']),
            MySecondService(kwargs['local-ip-address'])]
```

- Create a module section in the component configuration file; for example if your module is called `mplane.components.mycomponent`:

```
[service_mycomponent]
module: mplane.components.mycomponent
local-ip-address: 10.2.3.4
```

- Run `mpcom` to start your component. The `--config` argument points to the configuration file to use.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

m

A

B

C

D

E

F

G