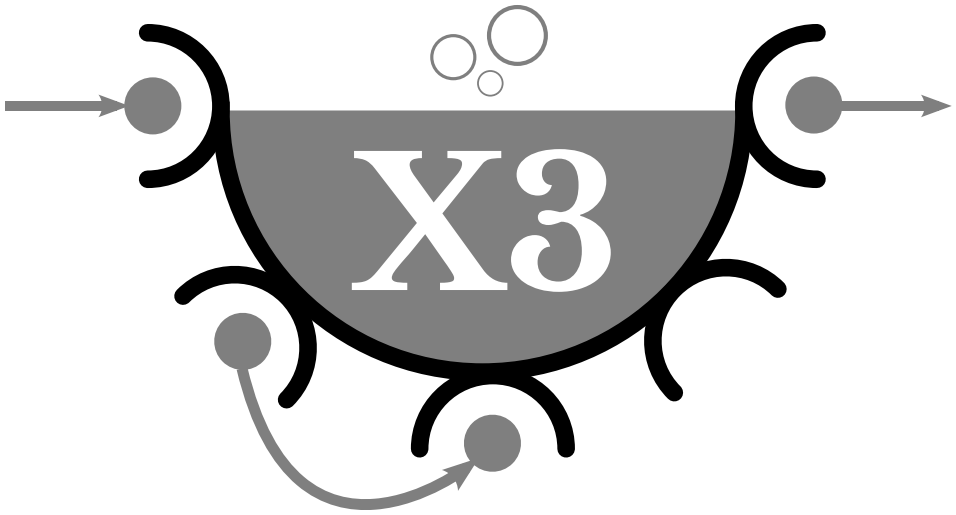


ExtreMon

Manual Generated 17/04/2013

Contents

Chapter 1

Node Structure

1.1 General Structure

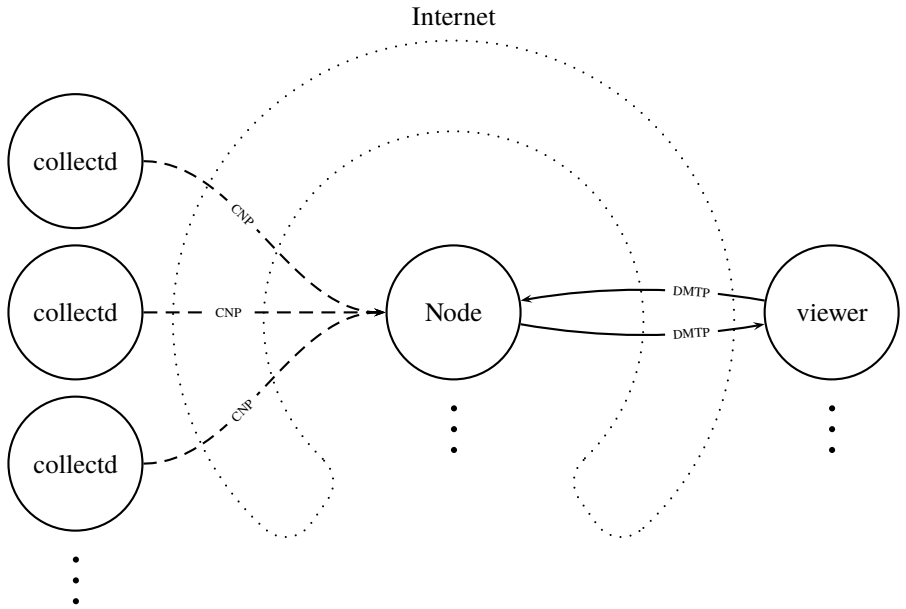


Figure 1.1: Node Components

The high-level structure of ExtreMon in an Internet context is shown in Fig. 1.1: A centralised Node passively receives internal measures from various hosts using CNP (collectd's UDP-based network protocol), actively performs various functional tests on these and possibly other hosts and services, and serves the results and derived results to a number of clients, for example the ExtreMon Display client, over the network, using the DTMP protocol. The dots in Fig. 1.1 indicate that normally, there will be many hosts monitored, at least two ExtreMon Nodes, and various clients.

1.2 Inside a Node

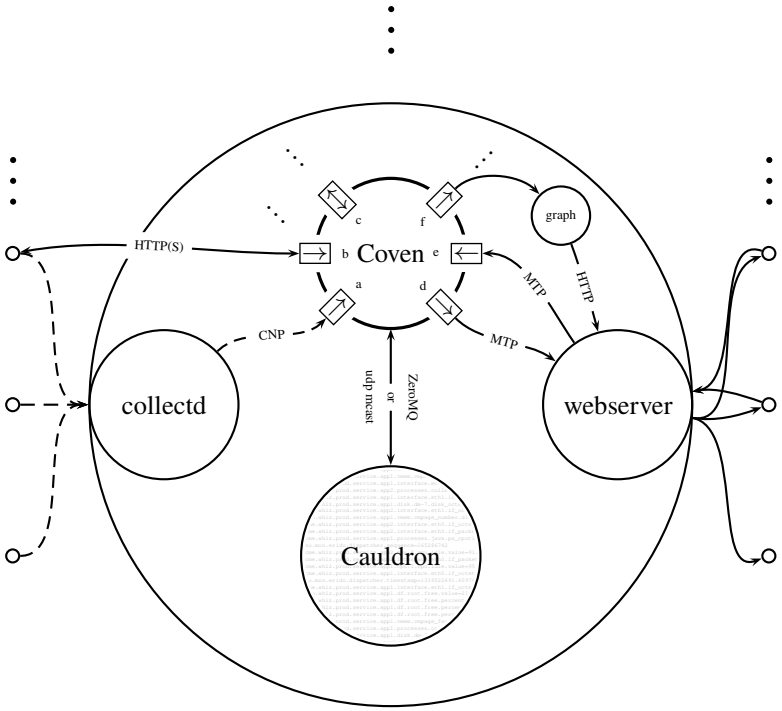


Figure 1.2: Node Components

The structure of a Node is shown in Fig. 1.2: The central component of a Node is the "Cauldron" around which the other components gather. The Cauldron is an N..M publish-subscribe mechanism: Any number of writers may present it with shuttles (see 3.1), it will duplicate them and send a copy of any received shuttles to all subscribers. To uncouple the other components from whichever mechanism is used to achieve this, the Coven component connects to the Cauldron and abstracts away its pub-sub functionality behind a set of simple roles, to be implemented by a set of Plugins. Plugins fit into one of 3 types:

1.3 Write Plugins

Write Plugins acquire measures from some arbitrary source and write these to the Coven as MTP shuttles within a particular subset of the namespace. The Coven feeds these measures into the Cauldron using its publish mechanism. In Fig. 1.2, a,b, and e represent write Plugins: a reads CNP packets from a collectd instance, b probes external web services and measures their response times, e receives operator feedback.

1.4 Read Plugins

Read Plugins receive shuttles containing a particular subset of the namespace, from the Coven, who feeds them by maintaining subscriptions to the Cauldron's subscribe mechanism, for them. In Fig. 1.2, d and f represent read Plugins: d has a local DMTP listener to which the webserver directs DMTP requests from the Internet: All external DMTP clients will end up being served by this important Plugin. f is a specialised read Plugin that feeds measures into third-party graphing engine.

1.5 Read/Write Plugins

Read/Write Plugins are Read and Write Plugins combined: They both read values from the Coven, and contribute other values. While other cases are certainly possible, Read/Write Plugins will most often supply aggregation functionalities, reading measured values and contributing new values derived from these. In Fig. 1.2, c represents a read/write Plugin, clearly an aggregator, for its lack of external connections. For example, c could be taking absolute values, say, free bytes on disk partitions, or free bytes in memory, etc.. Combining them with other measures to produce the same information in percentage form, which is easier to formulate general thresholds against. Another read/write Plugin could then be taking these percentages, applying threshold conditions to them and contributing alert states, or trending them to predict alert states.

Current Nodes rely heavily upon a local instance of the collectd agent to receive and aggregate measures from various collectd agents deployed on the hosts to be monitored, and an instance of a webserver, to supply SSL termination, authentication, authorisation, and integration with other services, like graphing engines. The Coven, and any of the current DMTP Plugins have no form of security whatsoever: while they may read metadata supplied by the webserver, the veracity of this is entirely left to the latter. This was done intentionally.

Chapter 2

Example Plugins

ExtreMon Plugins live in a managed directory, are required to be in the venerable *Nix-like "shebang" format, that is, they need to start with a hash mark and exclamation mark, followed by the path to the intended interpreter. Additionally, the interpreter used must recognize the hash mark as comment. Of the currently written Plugins, most are themselves Python scripts, and hence have "#!/usr/bin/python" in the first line. For our few Java-based Plugins, we'll use the shell, as interpreter, and launch the plugin, compiler into a .jar file, from there. Comments starting with "#x3" are parsed as label-value pairs by the Coven, and placed into environment variables when it executes the plugin. Special arguments, such as x3.in.filter and x3.out.filter are, additionally, interpreted by the Coven: in.filter is a regular expression determining which subset of the namespace will be presented to the plugin, out.filter is a regular expression determining which subset of the namespace the plugin may contribute to.

All of the Plugins in the sources are currently called "Example Plugins" because of the Framework nature of the project. Undoubtedly, quite a few of them will get consolidated into base classes or "default" Plugins as the project progresses. For example, most users are likely to want from_collectd, httpprobes, chalice_out_http, several _percentage and several _state Plugins, so these will probably become more generic and go into a "default" class of their own very soon.

2.1 Input Plugins

Input Plugins acquire measurements from an external source. We distinguish these logically from the External Functional Tests because, while the functional tests undertake their down actions by connecting, downloading, probing.. these are entirely passive and merely accept values from an outside source.

2.1.1 from_collectd

The from_collectd plugin arguably, the most important plugin in the current portfolio. Its function is to join a multicast group, decode CNP packets, possibly making some adjustments and translations, and contribute them to the Cauldron as MTP shuttles.

CNP uses strongly typed, but externalised semantics. It is therefore necessary to parse the collectd types databases to be able to interpret the binary format. The databases contain the number of values, value types, and possibly valid ranges for each data type. Because collectd uses a binary format, it can liberally allow many characters, even in labels. Converting for MTP which is a text-based format therefore requires limiting the range of characters allowed in labels. The from_collectd plugin removes the characters specified in the x3.zapgremlins configuration parameter to remain in labels, as well as replacing the dots (which are reserved, in MTP) in IP addresses by dashes.

Note that the code has the ability to add post-processing functions by pattern match, and this is currently used to convert the collectd "jiffies" counter for cpu percentage to pseudo-percentage, and to clamp such pseudo-percentage values to 100, as collectd's highly efficient but odd strategy in this regard is wont to produce occasional values > 100% otherwise (but only as a rounding error, since > 100% actually represents a measure of 100%, we can safely clamp this here).

Finally, parsing CNP ourselves, we are also faced with the task of deriving values marked as such, ourselves (where collectd does this internally, otherwise). For DS_TYPE_DERIVE, we therefore contribute to the Cauldron a value computed as $\frac{V_n - V_{n-1}}{T_n - T_{n-1}}$ where V_n is the current measure, V_{n-1} the last measure, T_n the time of the current measure, and T_{n-1} the time of the last measure, instead of the raw, received value, which is actually a counter, counting upwards. For example, CPU usage percentages, per CPU core or per process are actually transmitted as the number of *Nix *jiffies* that the target has been consuming cpu resources. This may be problematic in case of low usage.

2.1.2 httpprobes

The httprobes plugin is a very general, multithreaded http(s) service tester. Based upon a separate configuration file, formatted as JSON, which is read by the main ResponseTimeProbes class, a series of ResponseTimeProbe instances is created, one per URL to test. Each ResponseTimeProbe is passed the parameters of what and how to test, and runs this in a separated thread. Any decisions reached are directly contributed to the main class through its put method (which it inherits from X3Out). In this manned, many HTTP(S) tests can be ran, with different rules, speeds, expectations and responses, each only trivially impacting the timing of the others.

The httpprobes can follow redirects if required, seek through the response for text and for a specific HTTP response code, for http and https URLs. Each

URL tested can have its own testing frequency and settings. It contributes an appropriate status and diagnostic message.

httpprobe's configuration file is JSON-formatted:

```
{
"name.space.extension":{"url":"https://the.url.to.probe"[,option..]}
...
}
```

Options are: "interval" the number of seconds to wait after each probe (default:1), "follow" ("yes" or "no") whether to follow HTTP redirects (default:no), expect_code the HTTP response code considered "normal" for the URL, and expect_text, a string that is supposed to be found in the data returned from the URL. For example, to test http://secure.globalsign.net/crl/root.crl every 10 seconds for connection and a 2XX HTTP response:

```
{
"ext.globalsign.crl":{"url":"http://secure.globalsign.net/crl/root.crl",
                      "interval":10}
}
```

The config above will insert $prefix.ext.globalsign.crl.responsetime, $prefix.ext.globalsign.crl.result and $prefix.ext.globalsign.crl.result.comment into the Coven, every 10 seconds, representing the response time in ms, the result (was it OK or an error) and a textual comment explaining the result.

2.1.3 chalice_in_http

The chalice_in_http plugin in is the counterpart of the chalice_out_plugin in that, instead of streaming shuttles to clients using a HTTP GET method, it allows clients to submit shuttles using a HTTP POST request. Although this might be used, in the future, to accept MTP streams from external sources, at time of writing, its only function is to enable operators at the ExtreMon Viewer to indicate their involvement with specific states or components, and to write their comments for others. Obviously, this requires both authentication and authorization. None of the internal parts of ExtreMon have security features, as by design, this is left to the front-end webserver. This is a case where information from that front-end is passed on, though. The chalice_in_plugin requires the presence of two custom HTTP header fields: X-FirstNames and X-LastName which it will use to construct an "Operator Name" for use in contributing *responding* states.

Listing 2.1: Mutual SSL Authentication and Authorization using DN

```
1
  <VirtualHost XXX.XXX.XXX.XXX:443>
3
```

```
[...]

  <Location /console/libation>
    ProxyPass http://localhost:17617 retry=0 ttl=10
    Order allow,deny
    Allow from all
    SSLVerifyClient require
    SSLOptions +FakeBasicAuth
    RequestHeader set X-NotAfter            "%{SSL_CLIENT_V_END}s"
    RequestHeader set X-DistinguishedName   "%{SSL_CLIENT_S_DN}s"
    RequestHeader set X-Firstnames          "%{SSL_CLIENT_S_DN_G}s"
    RequestHeader set X-LastName            "%{SSL_CLIENT_S_DN_S}s"
    Satisfy All
    AuthType Basic
    AuthName "x3_console_libation"
    AuthBasicProvider file
    AuthUserFile /etc/apache2/extremon_eid_users
    Require valid-user
  </Location>

[...]

  SSLEngine on
  SSLCertificateChainFile "/etc/apache2/ssl/gov_ca_chain.pem"
  SSLVerifyDepth 4

[...]

</VirtualHost>
```

Listing 2.1 is a (redacted) snippet from the Apache webserver configuration that handles the security for this plugin: It is a VirtualHost with SSL enabled, and mandatory Mutual Authentication. The CertificateChainFile contains the Belgian Citizen CA's, so that it requires a valid Belgian Electronic Identity card to be able to use this URL (and the card will require the citizen's PIN code before being able to set up Mutual SSL). Moreover, the combination of the +FakeBasicAuth option and Satisfy All requires that the Distinguished name of the citizen's Authentication certificate be listed in /etc/apache2/extremon_eid_users before access is granted. Enrolling operators with feedback capability therefore becomes a matter of appending their DN to that file. Once established, the SSL_CLIENT_ variables contain various fields from the operator's authentication certificate, their firstnames and last name, amongst others. These are then added as HTTP Headers by RequestHeader directives. Finally the request is served by the chalice_in_http plugin, in the backend, which may be confident that the caller is an authorized operator, and that the names in the X-FirstNames and X-LastName headers are accurate.

2.1.4 twitter

The twitter plugin was a somewhat playful addition to the prototype's plugin portfolio. Its rationalized purpose might be to display theme-related tweets during incidents that would impact large parts of the population. The plugin is capable of following a list of users on Twitter using that company's streaming API, and contributing their tweets to the Cauldron, as they occur, allowing any other Plugins or DMTP clients to read these, e.g. display them on an ExtreMon Viewer. Early versions have been extremely useful, however, in testing many other mechanisms inside the prototype, for properly handling encoding and volume issues: Connecting the Stream API's testing features flooded the Cauldron and related components with a high volume of records whose data part used many different writing systems. This amounted to an effective "fuzzing" test of those components. Also, because of the unavailability of the required oauth components for python 3, this has forced us to ensure python 2 compatibility for the whole of the prototype, and important exercise because python 2 is still widely used.

Config is in "INI" like syntax, and must contain at least the oauth credentials for the Twitter Streaming API, and a target group containing either users to follow and/or strings to track.

```
[token]
key=123456789-dsfkgjsdfhlkcgsjdfhcglkdjfcghldjkcgshdnl
secret=dsjghdslfcgjsdfhlncgjsdnhflcgksjldfjcgsnhl

[consumer]
key=sgdgdfgcdfgcsdkgjdgd
secret=xdfjghcsldfukcgshgjlksdfhcgldsfjcghsnljkgs

[target]
follow=patrickdebois,krisbuytaert,lusis,fedtron,amonthemetalhead
track=loadays,extremon,commons-eid,eid-mw
```

2.2 Output Plugins

Output Plugins use selected values boiling at the ExtreMon Cauldron and output them to other systems.

2.2.1 chalice_out_http

The chalice_out_http plugin is an HTTP server, based on the standard python HTTPServer and BaseHTTPRequest classes, and rendered multithreaded using the accompanying standard ThreadingMixIn class.

The main class: HTTPSelectiveChaliceDispatcher is an X3IO subclass (see Section 5.4) that passes incoming shuttles to the single HTTPSelectiveChalice-

Server it instantiates, and contributes statistics about the number, and possibly identity of the connected clients.

The single HTTPSelectiveChaliceServer instance, being a also ChaliceServer has a write() method to accept and enqueue incoming shuttles.

When a new connection arrives, accepts the connection, instantiates a new HTTPSelectiveChaliceRequestHandler, passing it the connected socket, so each connection is handled by its own HTTPSelectiveChaliceRequestHandler instance.

The Request Handler's do_GET() method is called by the framework, for HTTP GET requests. The request handler interprets the URI path received as a simplified regular expression, sends HTTP response headers, creates an output queue, adds itself to the HTTPSelectiveChaliceServer as a consumer, and loops reading from its output queue and writing records from the shuttles enqueued there, filtered by the regex in the path, to the open connection. This is an endless loop, that will keep writing shuttles until either the client disconnects, or the plugin is terminated.

shuttles are enqueued because the HTTPSelectiveChaliceServer will distribute shuttles to all its subscribers using by calling their write() methods. HTTPSelectiveChaliceRequestHandler's write() method simply queues the shuttle, allowing the ChaliceServer to serve all the other subscribers, decoupling the throughput of the server from any and all subscribers.

Any lag in reading shuttles will cause the queue to grow, a shuttle written out to the client will cause the queue to shrink. When the queue reaches one thousand entries, new entries are silently discarded. Clients can easily detect this because this will cause missing shuttle sequence numbers.

Listing 2.2: Integrating chalice_out_http

```
<VirtualHost XXX.XXX.XXX.XXX:443>

    <Location /console/chalice>
    ProxyPass http://localhost:17817 retry=0 ttl=10
    Order allow,deny
    Allow from YYY.YYY.YYY.YYY ZZZ.ZZZ.ZZZ.ZZZ
  </Location>

</VirtualHost>
```

Listing 2.2 gives an example of how to integrate chalice_out_http into your webserver's URL space (Apache HTTP example).

2.2.2 to_graphite

The to_graphite plugin feeds records into graphite's processing backend called *carbon*. The to_graphite plugin reads all values present in the namespaces specified, connects to the carbon_cache daemon and writes the values to it in its internal (pickle) API, which is the fastest way of injecting large volumes of mea-

sures effectively.

2.3 Input/Output "Derivative" Plugins

2.3.1 average

The average plugin plugin calculates a basic cumulative moving average over an essentially eternal period. It features state persistence to be able to survive a restart without losing state. This was the first attempt at contributing averages, and was mainly useful in establishing the visualisation and state determination based on averages. This type of average becomes too viscose to be useful, after a few hours, which is why the Modified Moving Averages Plugins were introduced.

2.3.2 mma_ Plugins

The mma_ Plugins currently compute $\forall W \in [1, 5, 15], A(W)_n = V_n + e^{-\frac{1}{W \times 60}} \times (A(W)_{n-1} - V_n)$ —averages over 1, 5 and 15-minute intervals, but which could be trivially adapted to produce averages over any interval—. Note the unusual configuration directive *reject.outlier.above*, which should stand out as instantly suspicious: However, the combination of the manner in which the Linux kernel reports cpu usage, our 1 Hz sampling interval, and our custom-built derivation method for DS_DERIVED_ types transmitted by the collectd daemons, create issues when measuring low CPU usage, causing cpu percentages of up to 30 times the actual value reported by the OS's own toolsets to appear in our calculated output. Careful comparison of the build-in GNU/Linux tools' appraisal of CPU usage and our results showed that these outlier needed to be rejected to obtain a near-linear correlation between those tools, considered the reference, and our own results.

2.3.3 _percentage Plugins

_percentage Plugins are very similar in terms of structure: All of them convert a series of related absolute measures into corresponding percentages. They are prime examples of the usage of the cache and capture facilities of the X3In Plugin Facilitator framework, as they all need to capture groups from the labels of their input filters (capture), and require all values in a series in order to make their calculation (cache). The _percentage Plugins have much redundant logic and are prime candidates for consolidation into one more general plugin.

2.3.4 _state Plugins

The _state Plugins below are very similar in terms of structure: The take a series of measured contributed by other Plugins, make a determination of the *normality* of those values and contribute corresponding *state* information: OK, WARNING, ALERT or MISSING, and possible comments that go with their appraisal. Most

of the decision making centers around thresholds found in typical monitoring tools. Some go a little further. For example, the responsetime_state compares to thresholds, but also to an average value for the measure, where available (see the averages and mma_repsonsetime Plugins, resp. and to an overall minimum it keeps for itself, reasoning that if a service responds faster than the overall minimum, this can indicate a faulty test.

The node_state and system_state Plugins stand out in terms of functionality. They are both state and aggregators Plugins. Both aggregate states of parts of the namespace and contribute the worst state in that prefix. node_state does this at the level of one application server node, the smallest unit of redundancy for a backend cluster or load-balanced back-end: The contributed state is the worst of the states of any of the states reported within that node's namespace. The system_state plugin does the same, at a system level, encompassing nodes, dependencies, and functional tests within that system's namespace. These Plugins are useful at an overview level, to rapidly ascertain where problems exist, before zooming in to details. This is also the point where external dependency tests should be integrated: If we know that a system cannot function without a certain external dependency, than the state of that system's namespace should be at ALERT whenever that external dependency is proven to be unavailable.

Chapter 3

Protocols

3.1 The MTP Shuttle

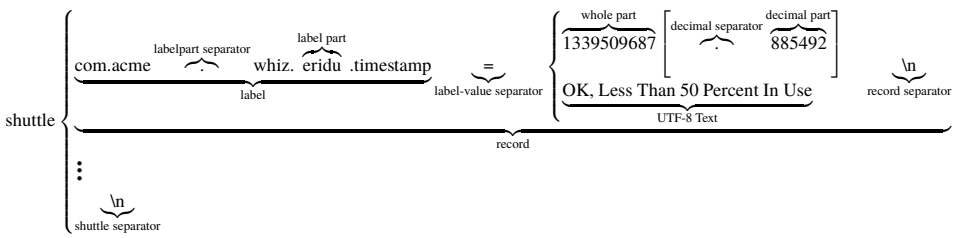


Figure 3.1: The MTP Shuttle

An Measure Transport Protocol shuttle is, intentionally, text-based, human-readable and simple. Fig. 3.1 expresses its constituent parts, explained below:

- records are sent as label-value pairs, with the ASCII "=" (equals) character (hexadecimal value 3d) separating label and value, and ASCII linefeed (hexadecimal value 0a) separating records (but being considered part of the record)
- where records are logically grouped, in terms if having one or more common elements such a group is called a shuttle. Shuttles are separated by an empty line, that is, two record separators (and this is considered part of the shuttle)
- a label consists of label-items separated by ASCII "." (full stop) character (hexadecimal value 2e). Label-items either represent an information hierarchy, general-to-specific from left to right, or a collection of labels whose position is not relevant. Only the first situation will be considered here.
- a value consists of a decimal number, with arbitrary precision, optional decimals indicated by one ASCII "." (full stop) character (hexadecimal

value 2e), or text, with arbitrary values except as stated below, encoded in UTF-8.

- since the equals and linefeed characters thus have special meaning, they are illegal in both labels and values. A full-stop character may occur zero or once in a numerical value, zero to N in a textual value.

- as a minimum, a shuttle contains a timestamp and a sequence number.

  - a shuttle timestamp last label item is "timestamp", with the rest of the hierarchy being determined by the organisation. It's value is always a number, a high-precision variant of posix time: The whole part a decimal representation of the number of seconds elapsed since midnight on the 1 January 1970 (UTC), the decimal part representing sub-second precision, the first digit representing $\frac{1}{10}$s, the second $\frac{1}{100}$s, etc..
  - a shuttle sequence number is a monotonic whole number that increases by one. The main sequence that should always be present is the sequence counter of the Cauldron component, which is system-wide, by Node: This sequence allows all consumers of any MTP-based protocol to discover lost shuttles instantly by comparing incoming shuttle sequence numbers to the previous shuttle's sequence: it should be exactly one higher. If the difference is more than one, this indicates shuttles have been lost. While this is not an issue in message-based raw MTP clients, DMTP clients may choose to mark such a connection as unreliable, or open a new connection to get a full cache dump, to make sure no information was missed.

- a value that is derived from one single other value, will append a label-item to the label of the value it was derived from.

Fig. 3.2 shows 2 shuttles captured from our experimental setup at client. The first shuttle is on lines 1 to 32, the second, on lines 33 to 35. (the empty lines serving as shuttle separators are considered as part of the shuttle). The shuttle on lines 1 to 32 has a set of records, most of them numeric, and most of them related to measures emanating from collectd agents deployed on the target hosts, or contributed from a remote testing probe (line 27 is from such a remote probe). Some, however, are internal measures contributed by the cauldron or MTP output, as in lines 13 and 18, which give the incrementing shuttle sequence number, and local timestamp of the central host. Some are derived values. Line 20 is a direct measure of free disk space, in bytes. On line 21, a read/write Plugin has consolidated this value with the 2 related ones for this specific disk partition (free and reserved), and has calculated that 2107334656 bytes corresponds to 50.6% free space. On line 22 another read/write Plugin has taken that percentage value, compared it to some appropriate threshold and decided that for this disk

```
 1 com.acme.whiz.prod.service.app1.interface.eth0.if_octets.tx=239923
 2 com.acme.whiz.prod.service.app1.disk.xvda.disk_octets.write=38909.1
 3 com.acme.whiz.prod.service.app1.vmem.vmpage_number.mapped.value=6474.000000
 4 com.acme.whiz.prod.service.app1.interface.eth1.if_packets.rx=3.49887
 5 com.acme.whiz.prod.service.app2.processes.collectd.ps_cputime.user=5003.68
 6 com.acme.whiz.prod.service.app1.interface.eth1.if_packets.tx=16.4947
 7 com.acme.whiz.prod.service.app1.disk.dm-7.disk_octets.write=0
 8 com.acme.whiz.prod.service.app2.interface.eth1.if_octets.rx=8838.65
 9 com.acme.whiz.prod.service.app1.vmem.vmpage_number.dirty.value=69.000000
10 com.acme.whiz.prod.service.app2.interface.eth0.if_octets.tx=232.004
11 com.acme.whiz.prod.service.app2.interface.eth0.if_packets.rx=1.00002
12 com.acme.whiz.prod.service.app1.processes.java.ps_cputime.user=5000.05
13 be.apsu.mon.eridu.dispatcher.sequence=165286742
14 com.acme.whiz.prod.service.app2.cpu.0.cpu.idle.value=91.7808
15 com.acme.whiz.prod.service.app2.interface.eth0.if_packets.tx=2.00003
16 com.acme.whiz.prod.service.app1.cpu.1.cpu.idle.value=95.9926
17 com.acme.whiz.prod.service.app1.interface.eth0.if_octets.rx=153.951
18 be.apsu.mon.eridu.dispatcher.timestamp=1339522691.609749
19 com.acme.whiz.prod.service.app1.interface.eth1.if_octets.tx=20260
20 com.acme.whiz.prod.service.app1.df.root.free.value=2107334656
21 com.acme.whiz.prod.service.app1.df.root.free.percentage=50.65194619031866
22 com.acme.whiz.prod.service.app1.df.root.free.percentage.state=1
23 com.acme.whiz.prod.service.app1.df.root.free.percentage.state.comment=Less Than 60% Fr
24 com.acme.whiz.prod.service.app1.vmem.vmpage_faults.minflt=116.963
25 com.acme.whiz.prod.service.app1.processes.collectd.ps_cputime.syst=0
26 com.acme.whiz.prod.service.app1.disk.dm-3.disk_octets.write=38908.7
27 com.acme.whiz.prod.service.revokedauthcertchain.xkms2probe.responsetime=136.000000
28 com.acme.whiz.prod.service.app1.disk.xvda2.disk_ops.write=1.49942
29 com.acme.whiz.prod.service.app1.processes.java.ps_code.value=16564224.000000
30 com.acme.whiz.prod.service.app2.processes.collectd.ps_disk_ops.read=266.196
31 com.acme.whiz.prod.service.app1.processes.java.ps_disk_octets.read=551.006
32
33 be.apsu.mon.eridu.dispatcher.timestamp=1339522691.60963
34 be.apsu.mon.eridu.dispatcher.sequence=165286741
35
```

Figure 3.2: Examples of MTP shuttles

partition, 50.6% free was too little, but not yet cause for alarm. It contributed a state suffix to the percentage measure indicating a state of “1” meaning "warning’, and, on line 23 appended a human-readable explanation to that warning. Note that the namespace chosen at client site is hierarchical, according to reverse-dns fashion as is usual, for example, in Java package names. This allows, for example, for a service provider to gather data from many organisations without name collisions.

The second shuttle on lines 33 to 35 has no real measures. It consists entirely of the 2 internal measures sent by the dispatcher, to confirm that the connection is live, and to allow continuous comparison between the clocks of the dispatcher and the local clocks. We would recommend that a shuttle be dispatched every tenth of a second, with or without actual data.

3.2 Transports

Depending on the component it is used in, the MTP will be carried on different transport protocols: It is designed to carry its own sequence indicators, and maintains this even on message-based transports, to facilitate development of clients, and to allow multiple shuttles per message in message-based transports.

Two cases of MTP over message-based transports are inherent in the backend side of the prototype design, in the local network context of Nodes. These are not used in the front-end strategy, and hence never travel over any actual wires or routers.

Preliminary experiments used the UDP multicast facility present in the operating system as a low-level publish-subscribe mechanism: Plugins wishing to contribute records could merely transmit shuttles in UDP packets to a local multicast address. Plugins wishing to obtain records would join that multicast group, and receive copies of all shuttles thus transmitted.

Although, theoretically, transmission errors and lost packets are possible in this configuration, the design of the multicast code in e.g. the Linux kernel makes this extremely unlikely. The worst-case in practice: lost packets, is not a major issue in our design, as the message-based transports are only used in high-bandwidth, local, short-circuited protocol stacks, such as the local interface (lo) of Unix-like systems, and in that context we're retransmitting all measure results constantly at a high rate. A lost packet's values will be resent within a very short period, all the time.

Any contributor in this design accumulates records into a local buffer. A running calculation is maintained as to how large a shuttle the accumulated data would represent. The records are sent as one shuttle, as soon as either the next records would not fit into the maximum size of UDP packet allowed, or one tenth of a second have passed since the last such packet was sent.

With this technique, the Cauldron component could be considered another case of COTS reuse: It merely consists of the local network of the Node(s), and the multicast facilities of the local IP stack.

Later experiments have used the ZeroMQ framework to provide message-based PUB-SUB and PUSH facilities. The strategy is the same as in the original multicast facility: a contributor accumulates records until a configurable maximum is reached or one tenth of a second have passed, after which the resulting shuttle is encapsulated into a ZeroMQ message, and transmitted from a 0MQ PUSH type socket towards a 0MQ PULL socket listening in a cauldron process, whose only role is to transmit the shuttle over a 0MQ PUB socket, from where it will be received by all 0MQ SUB type sockets that have subscribed to it.

With this technique, the Cauldron component is a small daemon program, to be developed. The 0MQ technique is now preferred after concerns were raised about the availability of effective multicast semantics in cloud-based environments. We'll design our API in such a way as to be agnostic of which technology the Cauldron component is based upon.

In both cases above, the data per message is exactly one shuttle, as described, including the end-of shuttle marker, however, any code should interpret messages received as containing 1–n shuttles: multiple shuttles per message are likely to occur in later versions.

The input, output and input/output Plugins that are managed by the Coven component read lines of text from their stdin file descriptor, write lines of text

to their stdout file descriptor, or both. The Coven component presents them with copies of shuttles containing values limited to their subscription, and/or reads shuttles presented. The Unix Pipe mechanism used in both directions is essentially a half-duplex stream: shuttles are written back-to-back as they appear. The Plugin reads "lines of text" from stdin, accumulating or handling records as they appear, one per line (see Fig. 3.1). An empty line signified a shuttle has just ended, and reading zero bytes signifies that the Coven component wishes for the Plugin to shut down.

"Raw" Measure Transport Protocol is essentially shuttles (see Fig. 3.1) of all measures, within one tenth of a second of their being measures, sent back-to-back over an HTTP connection, in what amounts to an infinitely long "file" whose size is never known beforehand. The approach is a pragmatic way to leverage the ubiquity of the HTTP protocol, its wide availability and accessibility in many environments. In practice, raw MTP will occur rarely, as the bandwidth requirements may be prohibitive for general Internet use. Outside the Node environment, DMTP will be preferred.

DMTP is exactly the same as MTP (see 3.2), with the following exceptions:

- The first shuttle(s) sent after the HTTP response header consist of a dump of all values that the sender knows of (and that match the URL requested) at that moment in time. These special shuttles contain an indicator record of "cachedump" with value "true" within the appropriate namespace. Although it is expected that in most implementations this "key frame" will be sent in one (potentially huge) shuttle, implementations should not count on this and use the "cachedump" indicator instead, if they need to distinguish between these and subsequent "differential" shuttles.

- The first shuttle sent that has no cachedump indicator, and all subsequent shuttles of the current HTTP connection are "differential" shuttles: They do not repeat state already sent, but only indicate changes to previous state. Because many values change rarely, even when measured frequently, this allows for huge bandwidth gains.

A DMTP client that needs to compare previous or related values with incoming updates may wish to keep a local cache that is instantly provisioned after the cachedump shuttle(s) and kept up-to-date by differential ones. In such clients, DMTP effectively is a remote cache update protocol, and it thus becomes easy to create DMTP proxies to limit bandwidth usage, for example, if multiple clients on a LAN require the same, or subsets of the same namespace.

DMTP clients that merely need to respond to a particular subset of the namespace, without requiring state, may safely ignore the differential nature of the protocol: The cache dump will, in their case merely serve to supply an initial state.

In practice, The DMTP protocol, as such is expected to occur mostly on LAN's, or, on public services. Most organisations will probably chose to use one

of the secured variants, known as DMTPS

Even casual visual inspection of a few MTP shuttles immediately highlights one striking characteristic of MTP labels: redundancy. All labels carry the same organisation prefix, and since one host in one system in one realm can have thousands of measures being streamed in, a large part of those longer prefixes will be repeated, many times, as well.

```
1 com.acme.whiz.prod.service.df.boot.free.percentage=30
2 com.acme.whiz.prod.service.df.boot.free.percentage.state=1
3 com.acme.whiz.prod.service.df.boot.free.percentage.state.comment=Less Than 40% Free!
4 com.acme.whiz.prod.service.df.home.free.percentage=10
5 com.acme.whiz.prod.service.df.home.free.percentage.state=2
6 com.acme.whiz.prod.service.df.home.free.percentage.state.comment=Less Than 20% Free!
7
```

Figure 3.3: Small MTP shuttle before compression

Fig. 3.3 shows a small shuttle of 6 records, with much redundancy in the labels: All the labels are identical up to and including the 6th label part *df*, and even then half the records have *boot.free.percentage* following that one common label prefix. The prefixtree encoding attempts to remove some of that redundancy by

- expressing the shuttle as a tree, with nodes created according to label parts, from left to right, containing either more nodes and/or values. Fig. 3.4 shows our small example, thus represented, and with the tree levels below.

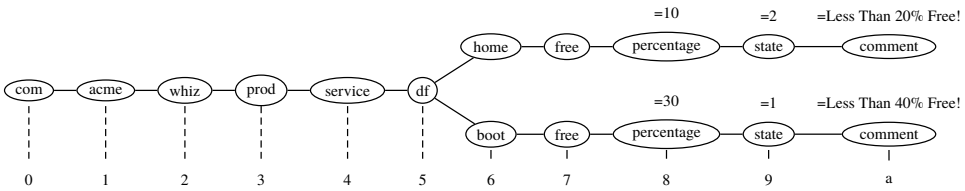


Figure 3.4: Small (D)MTP shuttle represented as a tree

- collapsing tree nodes with only 1 child that is not a value to a single node with the combined names of the collapsed nodes. Fig. 3.5 shows out small example tree, in this collapsed state: Only 4 tree levels remain.

- walking the resulting tree outputting lines containing the tree depth, the node name, and optionally, an equals sign and the node value as more formally described in Fig. 3.6).

Fig. 3.7 shows the result of prefixtree-encoding the small shuttle from Fig. 3.3: In line 1, the collapsed nodes *com,acme,whiz,prod,service, and df* from their corresponding label parts in lines 1–6 of Fig. 3.3 occur only once, at tree level zero, thereby indicating that all lower levels should be prefixed with this. In

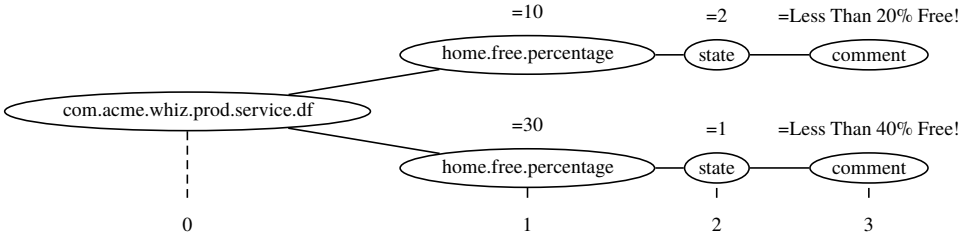


Figure 3.5: Small (D)MTP shuttle represented as a collapsed tree

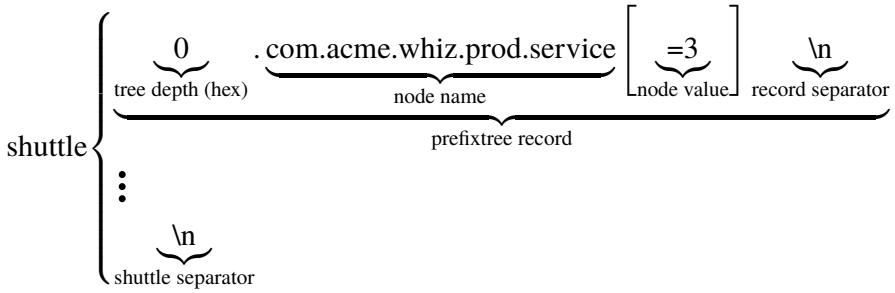


Figure 3.6: The DMTPC Shuttle

lines 2 and 5, the nodes at level one in the tree, express the 2 further label part possibilities, either *home.free.percentage* or *boot.free.percentage*. Note that in these cases, these node have their own values as well as expressing prefixes for further levels. This is a consequence of the MTP convention that derived values suffix the labels of their source, and indicates *percentage.state* was derived from *percentage*. The mechanism continues in the same manner into levels 2 and 3 of the tree, on lines 3,4,6 and 7 of Fig. 3.7

```
1 0.com.acme.whiz.prod.service.df
2 1.home.free.percentage=10
3 2.state=2
4 3.comment=Less Than 20% Free!
5 1.boot.free.percentage=30
6 2.state=1
7 3.comment=Less Than 40% Free!
8
```

Figure 3.7: Small (D)MTP shuttle encoded as a DMTPC Shuttle

With the kind of hierarchy shown, prefixtree-encoding easily reduces the size of shuttles by 50% or more for trivially small shuttles, individually encoded. We expect far better ratios for larger shuttles, and for variants of the protocol that maintain state between shuttles.

DMTPS and DMTPCS are DMTP or DMTPC transported over HTTPS instead of HTTP. The acronyms therefore covers a range of implementations, SSL and TLS variants, authentication schemes, key sizes, etc.. Its exact technical

implementation at any configuration will depend on the requirements, and rules of the organisation involved. All the back-end components implement only the plaintext variants of the protocols, and make no attempt at authentication or authorization whatsoever, and it is left to the front-end webserver and remote clients to agree on the exact security requirements per case. The only requirement for an implementation to qualify as being DMTPS resp. DMTPCS is that after the authentication was successfully performed, the payload transported (encrypted) by the resulting protocol is exactly DMTP or DMTPC. We expect most real-life traffic over the Internet will be DMTPCS, compressed differential MTP encapsulated inside a HTTPS (TLS) stream.

Chapter 4

SVG Viewer Namespace

First of all, a word of warning: ExtreMon Display is subject to intense discussion and no small measure of thinking on my part into where to take it, technology-wise. Drawing the "animatible" SVG schematic with all details is no trivial task, and I'm thinking on how to generate it rather than requiring someone to draw it. Also, the underlying Apache Batik library gives smooth change, zoom, rotate, all I could wish for, but *requires a J2EE runtime environment, and I have serious doubts about the future of Java on the workstation. A JavaScript solution is being pondered and experimented with, but so far the major stumbling block is the rendering performance (or lack thereof) of SVG on browsers: zooming non-trivial SVG takes seconds in the best case. This, and the lack of layoutmanagers in SVG leads me to consider everything between JavaScript Canvas and OpenGL even including (ab)using a 3D game engine. This means the information below may change, dramatically, unless the SVG engines inside the major browsers get much, much better, soon.*

To effectively map MTP records onto SVG elements in an in-memory representation of an SVG Document, several new attributes are introduced to mesh with existing SVG elements:

4.1 SVG Namespace

The x3mon:define attribute

```
<svg-tag x3mon:define="variable name:[attr name[/attr-part-name]]" svg-attr="value" [...] />
```

The x3mon:define attribute creates a named variable referring to the value of an attribute or the text content of the SVG element that contains it. This allows for arbitrary attributes of any SVG elements in the same document to be used in

```
1  <svg xmlns:x3mon="http://extremon.org/ns/extremon">
2   ...
3   <rect x3mon:define="statestyle[0]:style" style="fill:#79ffb3;"/>
4   <rect x3mon:define="statestyle[1]:style" style="fill:#ffff00;"/>
5   <rect x3mon:define="statestyle[2]:style" style="fill:#ff0000;"/>
6   ...
7   <g x3mon:usage="template" id="x3mon-cpu-core">
8    <g x3mon:id="cpu.steal.value">
9     <rect x3mon:map=":nset(width:#);.state:tset(style:$statestyle[#])" width="20" style
10    <text x3mon:map=":nset(:%.0f %% steal)"></text>
11   </g>
12  </g>
13  ...
14  <g x3mon:id="com.acme.whiz.prod"
15   <g x3mon:id="service"
16    <g x3mon:id="app1">
17     <g x3mon:id="cpu.0">
18      <use xlink:href="#x3mon-cpu-core"/>
19     </g>
20     <g x3mon:id="cpu.1">
21      <use xlink:href="#x3mon-cpu-core"/>
22     </g>
23    </g>
24   </g>
25  </g>
26  ...
27 </svg>
```

Figure 4.1: An X3Mon SVG Namespace Example

subsequent mappings, so that these can be managed visually from within e.g. an SVG editor. For example, I keep rectangles outside the document boundaries, that define the colors representing state (ok, alert, warning, ..), so these become easy to change from within the document. Lines 3–5 of Fig. 4.1 show define attributes being used to establish the variables “statestyle[0]” to “statestyle[2]”, containing the style information of the elements they occur in. At runtime, this will result in “statestyle[0]” having the value “fill:#79ffb3”, for example. Note that although the notation suggests some kind of subscript mechanism, this is only implied, the square brackets are simply part of the name here. The optional syntax attribute name/attribute part name allows to set the resulting variable to part of the attribute’s value. Currently, the only attribute on which this is defined are CSS svg:style attributes, where the resulting variable may be defined to one part of the element’s CSS style, its primary use being to use that part in a later x3mon:map sset action (see below)

The x3mon:usage attribute

<g x3mon:usage="template" svg:id="template-id"> [svg-elements...] <g/>

The x3mon:usage attribute marks an SVG element as being suitable to be used in a specific role. Currently, only the value "template" is valid which indicates that the element is an SVG group suitable to be used as an Extremon template. Lines 7–12 of Fig. 4.1 form an x3mon template because of this use of x3mon:usage in its topmost SVG <g> element. The identity of this template is determined by its SVG id attribute. This is the only instance where the SVG id is significant for our purposes (it is ignored everywhere else). This exception was made so as to be able to make use of the SVG templating mechanism, the SVG

<use> tag, as seen in lines 18 and 21 of Fig. 4.1, where our template used, twice. Re-using the existing SVG mechanism allows standard SVG editors to represent our template usage, which greatly facilitates editing. The similarity ends there, though, since x3mon will replace <use> directives by *copies* of the template to allow treating them differently to one another, where the original <use> elements all remain identical to the template. Fig. 4.2 shows the same SVG document after template instantiation.

```
1  <svg xmlns:x3mon="http://extremon.org/ns/extremon">
2   ...
3   <rect x3mon:define="statestyle[0]:style" style="fill:#79ffb3;"/>
4   <rect x3mon:define="statestyle[1]:style" style="fill:#ffff00;"/>
5   <rect x3mon:define="statestyle[2]:style" style="fill:#ff0000;"/>
6   ...
7   <g x3mon:id="com.acme.whiz.prod">···com.acme.whiz.prod
8    <g x3mon:id="service">·····················com.acme.whiz.prod.service
9     <g x3mon:id="app1">·····················com.acme.whiz.prod.service.app1
10     <g x3mon:id="cpu.0">·····················com.acme.whiz.prod.service.app1.cpu.0
11      <g x3mon:id="cpu.steal.value">··com.acme.whiz.prod.service.app1.cpu.0.cpu.steal.v
12       <rect x3mon:map=":nset(width:#);.state:tset(style:$statestyle[#])" width="20" st
13       <text x3mon:map=":nset(:%.0f %% steal)"></text>
14      </g>
15     </g>
16     <g x3mon:id="cpu.1">·····················com.acme.whiz.prod.service.app1.cpu.1
17      <g x3mon:id="cpu.steal.value">··com.acme.whiz.prod.service.app1.cpu.1.cpu.steal.v
18       <rect x3mon:map=":nset(width:#);.state:tset(style:$statestyle[#])" width="20" st
19       <text x3mon:map=":nset(:%.0f %% steal)"></text>
20      </g>
21     </g>
22    </g>
23   </g>
24  </g>
25  ...
26 </svg>
```

Figure 4.2: An X3Mon SVG Namespace with templates replaced

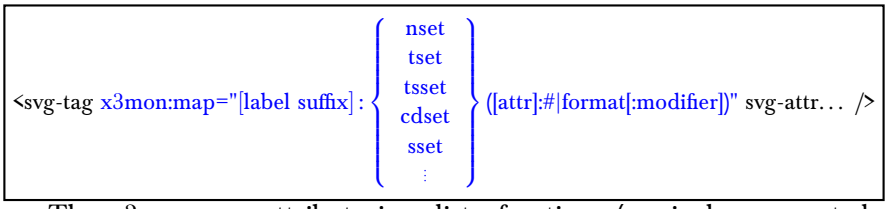
In 4.2, the SVG <use> elements of Fig. 4.1 have both been replaced by the SVG <g> they referred to. The <g> that was marked as the template has been removed. Note that this substitution will only occur with <use> elements that refer to <g> elements marked as x3mon templates: Any other <use> elements will be left undisturbed.

The x3mon:id attribute

<svg-tag x3mon:id="label part[.label part]..." [svg-attr="value"]... />

The x3mon:id attribute suffixes the MTP namespace of the containing SVG elements, making the identity of the element that contains it and those contained by it more specific. The MTP identity of any SVG element in the document is the concatenation, in depth-first traversal order, of any x3mon:id attributes encountered. Practically, this means that SVG group elements may semantically group all the child elements they contain into a single branch of the MTP namespace, requiring its children to have only more detailed label parts. This establishes a flexible hierarchy of identities inside the SVG DOM. Fig. 4.2 shows the element's resulting identities, in gray.

The x3mon:map attribute



The x3mon:map attribute is a list of actions (semicolon-separated, as inspired by the SVG style attribute), each one creating a relationships between the MTP namespace (or sub-namespace) and an attribute or the text content of the SVG element that contains it

Lines 12,13,18, and 19 of Fig. 4.2 show map attributes. Lines 12 and 18 have two, while lines 13 and 19 only specify one each. The nset actions show here don't specify a namespace suffix (the field is empty, we start with the colon used as separator, meaning they apply to the x3mon:id of the element that contains them. The tset actions do specify a suffix meaning they apply to the x3mon:id of their elements with that suffixed. In plain English, lines 12 and 13 of Fig. 4.2 signify:

When a record with id com.acme.whiz.prod.service.app1.cpu.0.cpu.steal.value arrives, set the width of the <rect> element in line 12 to its value, and the text content of the <text> element from line 13 to "(its value truncated without decimals) % steal"

When a record with id com.acme.whiz.prod.service.app1.cpu.0.cpu.steal.value.state arrives, construct a variable substituting "#" for its value in "statestyle[#]", look up its value, and set the style attribute of the <rect> element from line 12 to that value.

Some essential action types are defined below (but this syntax was designed to be extensible); all these set the attribute or text content of the element that contains them:

- *nset* (set numeric) formats a value using printf-like syntax, as a floating-point number
- *tset* (set text) formats a value using printf-like syntax, as a string of text
- *tsset* (set timestamp) formats a value given in seconds since the Unix-era into a timestamp
- *cdset* (set countdown) formats a value given in seconds to the period of time those seconds represent in years, months, days, weeks, days, hours, minutes and seconds. (this is an approximation since leap years (nor seconds) are taken into account).
- *sset* (set style) formats a value using printf-like syntax, and set an individual style element within the target's svg:style attribute to the resulting formatted string. This allows for SVG using CSS style (as opposed to distinct style attributes), to still have individual style elements impacted by

incoming data. The “attribute” argument is used to determine the style element, as the attribute is hard-coded to “style”.

The original SVG document, after defines have been recorded, templates replaced, and identities established, is ready to respond to some MTP records.

Chapter 5

Development

5.1 Typical Component Interaction Within A Node

Fig. 5.1 shows the interaction between Coven components. This diagram was simplified for legibility. In reality, many of the components have their own asynchronous queueing mechanism, effectively connecting 2 threads through a synchronized queue. This is only explicitly shown in the DMTPCOutPlugin/ DMTPCConn interaction, but will also be implemented in e.g. the ZeroMQ implementation of the Cauldron and many read and write Plugins

We show one (unspecified) write Plugin, one specific read Plugin (DMTP server backend), and no read/write Plugins.

From top to bottom:

- The Coven has launched one write and one read Plugin, subscribed to the Cauldron for its read Plugin, and opened a PUSH type channel for each read Plugin. The latter being stateless, it is not shown.
- The input Plugin has values to contribute, hands them to the Coven by writing them to stdout as MTP shuttles, whenever it has them (which may be frequently).
- The coven writes the values gathered (possible consolidated) to the Cauldron's PUSH channel.
- The Cauldron copies the values received by any input Plugins, (possibly consolidated) to all subscribers.
- The coven writes the values received from the Cauldron, to the stdin pipes of all output Plugins (possibly filtered for a particular subset of the namespace)
- The DMTPOutputPlugin updates its cache with the values received. While no clients are connected to its front-end listener, that is all it does.

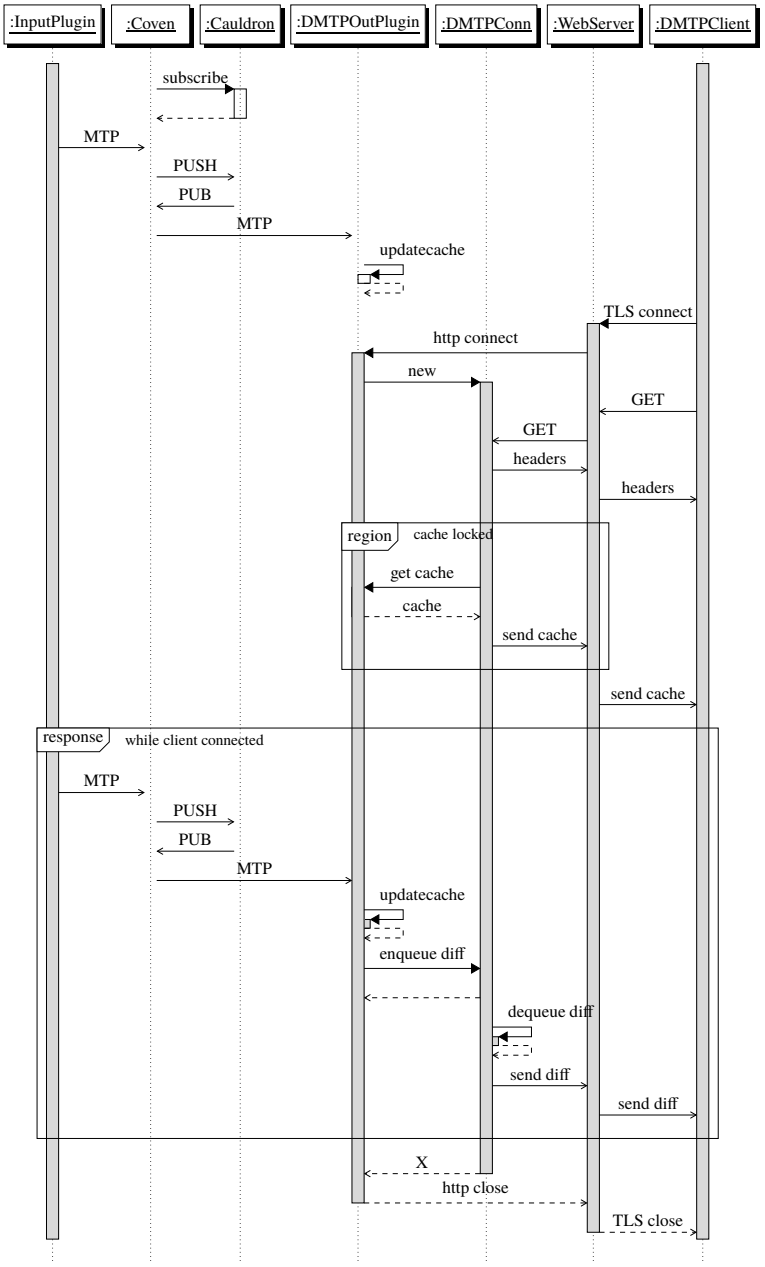


Figure 5.1: Typical Interactions Within a Node

- When an HTTPS connection is accepted from the Internet, the web server makes a back-end connection to the DMTPOutputPlugin's listener.
- for each new connection, the DMTPPlugin instantiates a DMTPConnection and hands over the connected socket, which handles the connection from reading the HTTP request and until the connection closes.
- the DMTPConnection interprets the HTTP GET, and URL information, possibly sets filters to limit the output, to set variants in its format, etc..
- the DMTPConnection requests a cache lock from the DMTPPlugin, and requests the entire cache contents (possibly filtered), which it sends to the webserver, and releases the lock.
- While 1..n clients remain connected to the DMTPPlugin via its DMTP-Connection instances, any cache updates will also be journalled, and the differences copied to all outgoing queues, where they will be sent over the existing connection at the tempo attainable by the specific client. This is the response to incoming data for as long as there are clients connected.
- to terminate, a client closes its TLS connection, which will cause the corresponding DMTPConnection to unregister itself and terminate.

5.2 ExtreMon Viewer Runtime

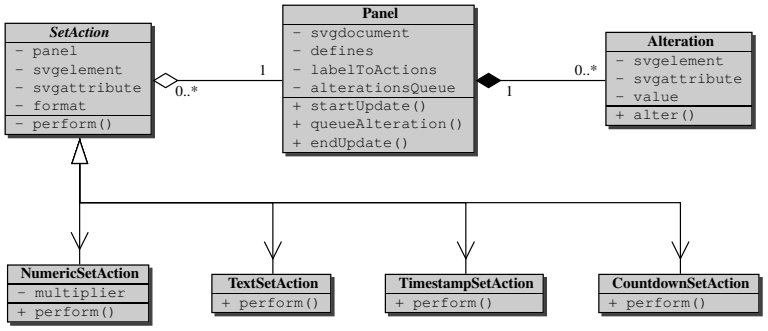


Figure 5.2: Runtime Class Diagram

The in-memory representation of the relationships established between the MTP namespace and the elements in the SVG DOM Tree maintained by the SVGCanvas from the Apache Batik library is represented in Fig. 5.2. The central class is an X3Panel which contains the Batik SVG document, doctored as described above, a dictionary storing the variables established by x3mon:define attributes, above, a queue of Sets of Alteration objects: One Alteration encapsulates exactly one .. alteration to either an attribute or the text content of exactly

one SVG element in the DOM, and a dictionary mapping certain MTP identities to sets of SetAction subclasses that they should cause. Every x3mon:map instance in the original SVG document instantiates one SetAction subclass, according to its type: “tset” instantiates a TextSetAction, “nset” a NumericSetAction, and so forth.

Mapping

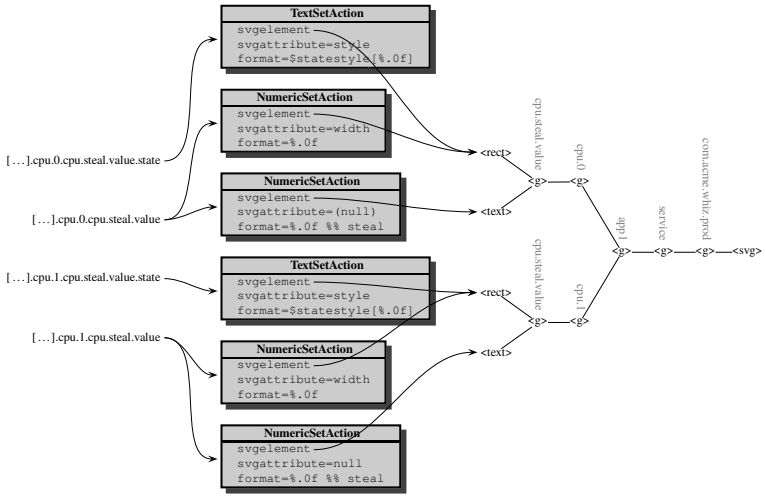


Figure 5.3: From MTP records to SVG DOM elements

Fig. 5.3 illustrates how incoming record are converted into SVG alterations at runtime: on the left is an associative array whose keys are the full MTP labels (truncated in front to save space in the drawing), mapping to values that are lists of SetAction subclass instances. An incoming label is looked up in the associative array, and this yields a list of SetActions to be executed in response. Every SetAction maintains a reference to the SVG element to act upon, the attribute of that SVG element to act upon (or null indicating that the text content of the element is the target instead), the format string to format the data first, and any modifiers, according to its type. However, this does not suffice to alter the SVG DOM: The Batik library does not allow its DOM representation to be altered from just any thread, only from its own update queue, and the updates can only be passed to it as instances of the Java Runnable interface.

Batching Updates

While the SVG DOM is exposed, and may be directly manipulated, this leads to undefined behaviour. Because the runtime cost of each Runnable object is relatively high vis-a-vis the number of alterations per second we expect, we’ll present Batik’s update queue with a type of Runnable that will set as many

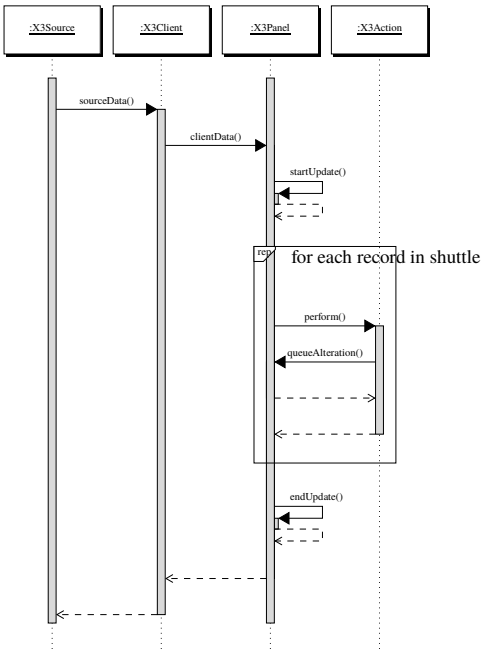


Figure 5.4: X3Mon Viewer Calling Sequence Phase 1

objects as once as is optimal. Our Alternator Class (not shown) is merely a Runnable containing a Set of Alterations, that, in its run() method calls each Alteration's alter() method. Our X3Panel class, then, receiving records, looks up corresponding SetAction subclasses, but these SetActions merely call X3Panel's enqueueAlteration to add the required alterations to its alteration queue. This is shown in Fig. 5.4

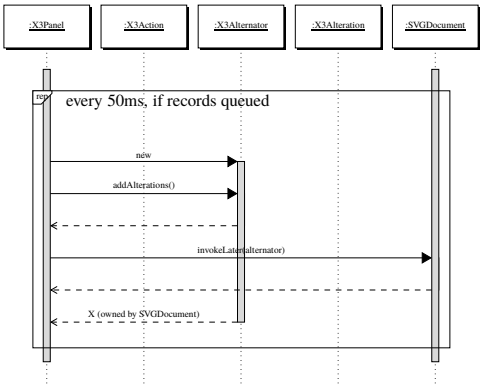


Figure 5.5: X3Mon Viewer Calling Sequence Phase 2

In a separate Thread, shown in Fig. 5.5, X3Panel unqueues these Alterations, gathers them into Alternator instances and presents them to Batik's update queue. This intermediary stage is where it will be possible to tune the

number of updates per Alternator, according to how the Batik library has tuned updates, which it does according to the capabilities of the underlying platform.

Updating The DOM From Batik's updateQueue

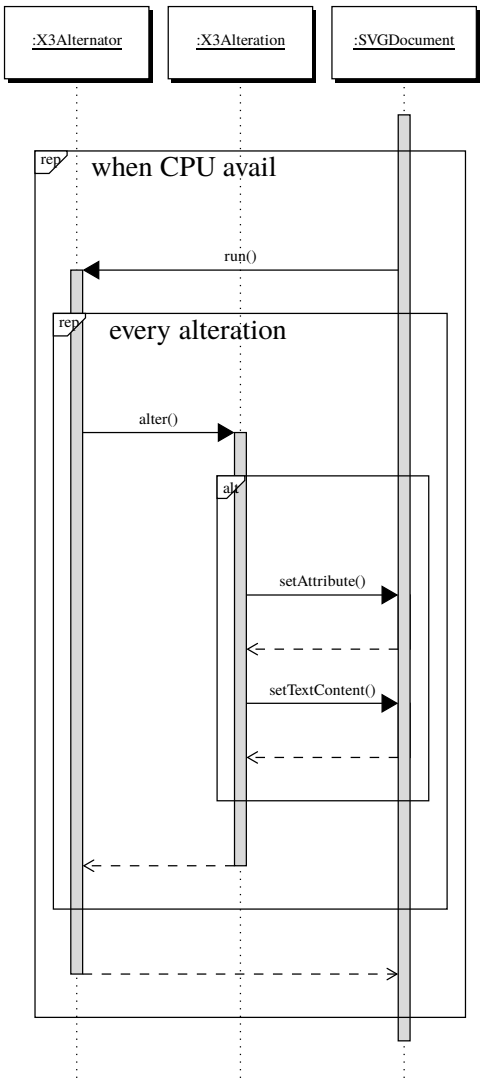


Figure 5.6: X3Mon Viewer Calling Sequence Phase 3

Batik will then, in its own good time, apply them to the SVG DOM, eventually rendering them visible. Fig. 5.6 shows how Batik's updatequeue dequeues X3Mon's Alternator instances and runs them, causing various SVG DOM elements to be modified, inside Batik's update thread, and at a time appropriate to fluent visual updates (as managed by Batik).

The Result: Updated SVG DOM

```
com.acme.whiz.prod.service.app1.cpu.0.cpu.steal.value=77
com.acme.whiz.prod.service.app1.cpu.0.cpu.steal.value.state=2
com.acme.whiz.prod.service.app1.cpu.1.cpu.steal.value=0
com.acme.whiz.prod.service.app1.cpu.1.cpu.steal.value.state=0
```

Figure 5.7: Example Incoming MTP Records

For the example incoming MTP records in Fig. 5.7, eventually, our example SVG document from 4.2 should end up being transformed and reach the state as shown in Fig. 5.8: The appropriate elements altered according to format specified.

```
<svg xmlns:x3mon="http://extremon.org/ns/extremon">
 ...
 <rect x3mon:define="statestyle[0]:style" style="fill:#79ffb3;"/>
 <rect x3mon:define="statestyle[1]:style" style="fill:#ffff00;"/>
 <rect x3mon:define="statestyle[2]:style" style="fill:#ff0000;"/>
 ...
 <g x3mon:id="com.acme.whiz.prod">
  <g x3mon:id="service">
   <g x3mon:id="app1">
    <g x3mon:id="cpu.0">
     <g x3mon:id="cpu.steal.value">
      <rect width="77" style="fill:#ff0000;" ...>
      <text>77 % steal</text>
     </g>
    </g>
    <g x3mon:id="cpu.1">
     <g x3mon:id="cpu.steal.value">
      <rect width=".01" style="fill:#79ffb3;" ...>
      <text>0 % steal</text>
     </g>
    </g>
   </g>
  </g>
 </g>
 ...
</svg>
```

Figure 5.8: Example SVG DOM as altered by incoming records

Finally, Fig. 5.9 shows what the altered SVG may look like, represented on the visual display (for cpu.0)

5.3 Core Classes

The core operation at the heart of many other classes is the assembly and timely transmission of MTP shuttles (see 3.1). The Loom classes encapsulate the desired behaviours of assembling and transmitting shuttles of a maximum size, and containing records of a maximum age. Thus, the calling class may blindly use the Loom's put methods to add individual records, and the Loom instance will accumulate these until either there are almost too many of them (max shuttle size) or too much time has elapsed (max shuttle age), at which point it will

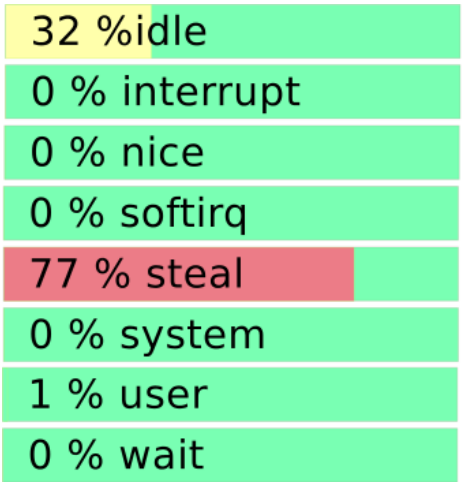


Figure 5.9: A Visual Example According To The Altered SVG DOM

call its assigned launcher instance, delegating the actual transmission of the shuttle data, since shuttles may be transmitted in various ways. For example, the chalice_out_http plugin will write shuttles to an open TCP connection inside an existing HTTP session, the Coven will write shuttles to unnamed pipes for its plugins, and multicast them over UDP packets, the dump tool will write shuttles to stdout. For example: The CauldronSender class is a Loom subclass, and merely provides for UDP multicast transport, inheriting everything else from Loom.

5.3.1 CauldronServer

CauldronServer is the base class for creating servers that handle distributing shuttles to multiple clients. It handles distribution and allows each client their own private queue, so that slow clients do not slow down operations for fast clients.

5.3.2 ChaliceServer

ChaliceServer is a specialization of CauldronServer that implements the differential protocol (see 3.1), by sending each new connecting client a full set of known values, and only changes afterwards.

The Server base classes have only been implemented in Python.

5.3.3 DirManager

DirManager is a utility class that encapsulated the inotify functionality if the Linux Kernel, as exposed by the pyinotify module, expands it with MD5 checksums, and exposes an interface that calls methods from its “watcher” delegate

with files have been added to, removed from, or changed inside the directory it is monitoring. The Coven class uses a DirManager to watch its plugins directory, allowing plugins to activate automatically when added to the plugins directory, shut down when removed, shut down and restarted when modified.

5.3.4 Coven

The Coven class is at the heart of ExtreMon. A Coven instance monitors a plugin directory (using a DirManager) and activates the files there as ExtreMon plugins, with a hot-plug strategy. Files added (or modified) are scanned for ExtreMon plugin headers. If such are found, the file is associated with a Plugin instance, which will attempt to execute the file, with the configuration headers in its execution environment, and with its stdin/stdout file descriptors redirected to Plugin2Cauldron/Cauldron2Plugin instances, according to its interpretation of the presence of input and/or output filters: Plugins with input filters (#x3.in=) get a Cauldron2Plugin on their stdin descriptor, Plugins with output filters (#x3.out) get a Plugin2Cauldron instance on their stdout. The Plugin instance blocks reading from the plugin's stderr, passing any lines read to the Coven instance for logging purposes: Anything a plugin writes to stderr ends up in the Coven's log, preceded by the plugin's name. Finally, the Coven class maintains some statistics about its running plugins, and gathers statistics from Cauldron2Plugin and Plugin2Cauldron instances running, to contribute using its own CauldronSender instance. The counting functionality is encapsulated in the Countable class.

Cauldron2Plugin

Cauldron2Plugin instances set up a CauldronReceiver read every shuttle that boils, and assemble these into new shuttles containing only records matching the plugin's input filter *but not its output filter, if it has one.* These are then written to the plugin's stdin. The limitation on the output filter is to avoid endless loops: If the plugin's logic were to respond to a record matching its input filter with another record that would, it would endlessly repeat that record. Hence, input plugins that are also output plugins are not allowed to see their own records.

Plugin2Cauldron

Plugin2Cauldron instances set up a CauldronSender read every shuttle from the plugin's stdout, passes any that match the plugin's output filter to the CauldronSender for contribution to the boiling records.

5.4 Plugin Facilitator Frameworks

While the protocol used in ExtreMon plugins was intentionally kept extremely straightforward: (for input: read lines from stdin, split them around an equals sign, part zero is the label and part one the value, ignore empty lines (end of

shuttle), to output, write label=value pairs to stdout, an empty line if you want your records to be transmitted ASAP), this does not mean that it is as trivial do write a CPU-efficient plugin when faced with high volumes of records. Also, there are some tasks that many plugins will have in common. Many plugins will want to match parts of their input filter, and many, for example aggregator plugins operating on multiple records at once, will want to maintain a cache of certain values.

5.4.1 X3In

By subclassing X3In, a plugin written in the Python language will have its overridden "receive" method called for incoming shuttles matching its input filter. Each shuttle is available as a dict() argument. In the constructor, the subclass may choose to enable the caching and/or capture facility, where caching means that the X3In superclass will keep a dict() of the last values received, and capture means the X3In superclass will call the receive method with an extra argument: a dictionary of all the named regular expression group matches present in the input filter.

5.4.2 X3Out

The XOut superclass gives a plugin written in the Python language the freedom to call its put() method whenever it has values to contribute, and to have these sent as shuttles on its stdout, in a timely and efficient manner, even if there are a great many of them, or in the other extreme, when there are very few.

X3Out also has a Java implementation.

5.4.3 X3IO

X3IO is a convenience class that makes its subclass both an X3In and X3Out.

5.4.4 X3Conf and X3Log

All X3In, X3Out, and X3IO instances are also X3Conf and X3Log instances, giving their subclasses access to the plugin configuration data (extracted from the environment variables set by the Coven), and exposing the log() method by which a plugin may write log entries into the Coven's log.

5.5 Network Client Facilitators

5.5.1 X3Client

Although writing a MTP or DMTP client was intentionally made as simple as possible, and the simplest cast comes down to making an http connection, and parsing individual lines after the HTTP headers for label-value pairs, even this

simple task may get more complex once https, mutual authentication, or multiple DMTP sources are to be read at once, and when reading large volumes or records. The DMTP client classes have been created to greatly simplify such tasks.

To receive shuttles belonging to the part of the namespace it is interested in, an application would instantiate an X3Client add as many X3Sources as it has DMTP servers to connect to for that namespace (for example, the chalice_out_http plugin) and add itself as an X3Client listener. The X3Client will then call the application's clientData method, passing it all the records that have changed from the previous call. If more than one X3Source have been added, the X3Client will merge the results and only pass the most recently received values. In this way, the application will continue to receive records, without any interruption, even if one redundant X3Source's connection is lost, for whatever reason.

To be able to receive X3Measure records , the application should implement the X3ClientListener interface.

An application interested in the state of X3Source connections might also implement the X3SourceListener interface , and add itself to the individual X3Sources, to obtain notification of connections and disconnections. In the same way, an application might choose to use only a single X3Source, and receive that source's records directly, without the mediation of an X3Client. An X3Source has a socketFactory setter, allowing alternate implementation to be injected, for example, to support Mutual SSL with tokens, as we do in the prototype.

5.5.2 X3Drain

The counterpart of an X3Source is an X3Drain , which allows an application to contribute records to a DMTP server, such as the chalice_in_http plugin described above. An X3Drain is a Loom, and will therefore disconnect the application's lifecycle from the sending of shuttles: The application is free to contribute individual records, the X3Drain will transmit them at most max_shuttle_age later, in its own thread, by HTTP POST. Because of the semantics of the HttpsURLConnection class used, HTTP connections that are regularly used will actually make use of one TCP connection, using the HTTP keepalive mechanism. This means that as long as there are records available every few seconds, shuttles will arrive at the server with very little delay, as the connection is already open, and any SSL handshake has been performed. This is important especially when using Mutual SSL, with smart card tokens, where the SSL handshake may take several seconds. An X3Drain has a socketFactory setter, allowing alternate implementation to be injected, for example, to support Mutual SSL with tokens, as we do in the prototype.

5.6 ExtreMon Viewer

The Visualisation Console ("Console") is a Java application, deployed by Java WebStart, that will open a live schematic view for any Java- enabled client.

5.6.1 X3Panel

The main classes of the "X3 Console" are the X3Panel which contains an Apache Batik JSVGCanvas representing an SVGDocument into which an SVG schematic with a supplementary namespace is loaded. The X3Panel parses the SVG tags inside the schematic and builds an internal structures that allows it to rapidly translate incoming DMTP records into alterations to the SVG DOM maintained by the JSVGCanvas.

Since our chalice_out_http plugin plugin will interpret the HTTP GET path it is called with as a regular expression, and use that to select which subset of the namespace to transmit, here is an opportunity to fine-tune bandwidth usage: The X3Panel, after parsing the SVG tags linking the schematic to the ExtreMon namespace, has a list of any and all DMTP labels it is capable of responding to. Because it makes no sense to subscribe to a subset it could not respond to, the Subscription class exist to build a specialized regular expression based on the labels an X3Panel can respond to, to allow an X3 Console to receive only relevant records, and gain bandwidth and CPU cycles.

5.6.2 Respondable

Some SVG elements in the schematic will represent states, and may therefore become eligible to connect operator feedback to those states. Elements thus selected are stored by the X3Panel as instances of Respondable

5.6.3 Actions

The Action classes are the runtime representation of the action tags present in the supplementary SVG namespace.

5.6.4 Alteration and Alternator

Since the SVGCanvas instance exposes the underlying DOM, but allows no alterations outside a specialised internal Thread, the Action classes can never directly alter the DOM (technically, they could, but this leads to undetermined behaviour and ultimately, a crash), instead, they convert their actions into Alterations, which contain the low-level information on what to alter on which element. The Canvas' UpdateManager may be passed a Runnable, that is executed inside the Canvas's update thread and this runnable is the only way to reliably alter the SVG. To avoid having to pass a great number of Alterations, all Runnables, which would be very inefficient, a number of Alterations are gathered inside an Alternator and this is passed to the update manager instead.

5.6.5 Supporting Classes

A few supporting utility classes and interfaces follow. SVGUtils encapsulates convenience methods to making working with the SVG DOM more readable and less verbose. ResponderListener is an interface which allows an application to listen for operator actions on Respondables. This is how an X3Console detects an operator responding to and commenting on a visual state. X3PanelListener is an interface which allows its implementor to know when an X3Panel is ready loading and parsing the schematic. This is how an X3Console knows that the schematic is ready, and it waits for this event before connecting to any X3Sources.

5.6.6 User Interaction

While arguably, user interaction could have been done entirely in SVG, development time dictated that we use a more classical approach for those few user interactions to be handled at this stage.

A CredentialsDialog is presented when an X3Source encounters a HTTP authentication request. It requests the username and password. This could happen, for example, if a front-end webserver were configured for HTTP Simple Authentication.

An X3UserAgent is an implementation of an SVGUserAgentAdapter, which is the way the Batik classes communicate problems. Our implementation simply logs anything it receives. It was essential to implement this and replace the default handler since the latter creates an error dialog for each problem, potentially overwhelming the user with dialogs for even a small misconfiguration.

A ResponderCommentDialog uses a JOptionPane to present operators with a simple dialog into which to type their comments when responding to a state.

5.6.7 X3Console

Finally, the X3Console class itself, which brings all this together. An X3Console instantiates an X3Panel inside an undecorated JFrame of exactly Full HD Size, sets a default authenticator that will request credentials from the user with a CredentialsDialog and asks the X3Panel to load the SVG schematic.

When the SVG is loaded (and interpreted for actions), the X3Panel calls panelReady() and the X3Console creates an X3Client and subscribes to it, creates an X3Source, retrieving the subscription regular expression that the Subscription class inside the X3Panel has prepared while interpreting the SVG.