Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Elio Gubser

Building a Path Transparency Observatory

Master Thesis MA-2016-FS
February 2016 to August 2016

Tutors: Brian Trammell & Mirja Kühlewind
Supervisor: Prof. Dr. Laurent Vanbever

Abstract

In the early days of the Internet, one could send a packet from a source to a destination and it would arrive unaltered. The deployment of so-called *middleboxes* in recent years ensured scalability and manageability of the growing Internet with the drawback that they violate the classic end-to-end principle by altering or filtering traffic for other purposes than packet forwarding. This makes it very hard to innovate the Internet stack and deploy new features and protocols.

The discussions about engineering a solution are mostly backed by small datasets. This master thesis developed the basis of a technical solution to this problem: a Path Transparency Observatory, where *path transparency* is defined as the probability that a stream of packets is received at its destination as it was sent by its source. The observatory is built upon a common data model of *observations*: an assertion, based on raw data, that a certain condition was observed on a certain Internet path at a particular time. The collection of large numbers of observations will allow new types of questions to be asked about path transparency: what is the Internet-wide prevalence of certain impairments? What are the correlations among common impairments?

Observations are generated from various different raw data formats by observatory plug-ins called *analyzer modules*. Observations can also depend on other observations. Because the expected amount of input data is very large, the observatory utilizes cluster computing technology. The process of detecting changes and updating the observations upon the addition of new data is done efficiently by processing only a subset of data that has been determined based solely on measurement time spans. By policy, raw data is never deleted from the observatory but can be marked invalid if it is flawed. This information is automatically propagated to all descendant observations.

The observatory allows users to query for a specific version of observations (e.g. to use it as a reference in a paper) and records how each observation was derived and which raw data was involved in its generation.

Contents

Chapter 1

Introduction

In the early days of the Internet, one could send a packet from a source to a destination and it would arrive unaltered. To ensure scalability and manageability, so called *middleboxes* have been introduced in recent years. Middleboxes are networking devices that violate the classic end-to-end design principle by altering or filtering traffic for other purposes than packet forwarding. They have implicit assumptions on the protocol stack and often operate transparently within the network. Examples are: firewalls, intrusion detection and prevention systems (IDS/IPS), wide area network optimizers, network address translators (NAT), load balancers, etc.

1.1 Motivation

Although their usefulness is undisputed, the existence of middleboxes makes it very hard to innovate the stack and deploy new features and protocols. The Internet becomes ossified. And its architects are very well aware of this problem. Unfortunately, the discussions about engineering a solution are mostly backed by small datasets. That is because measurement data is difficult to obtain: it is a matter of trust, terms and NDA.

This master thesis develops the basis of a technical solution to this problem: a Path Transparency Observatory, where *path transparency* is defined as the probability that a stream of packets is received at its destination as it was sent by its source. The observatory is built upon a common data model of *observations*: an assertion, based on raw data, that a certain condition was observed on a certain Internet path at a certain point in time. The transformation of raw data into observations makes it possible to respect data policy and enable research on measurements which would otherwise be inaccessible to third parties. The collection of large numbers of observations will also allow new types of questions to be asked about path transparency: what is the Internet-wide prevalence of certain impairments? What are the correlations among common impairments (e.g. that could be attributed to the same device or operational practice)?

1.2 Goals

- The observatory should enable analysis on large datasets by providing a framework that manages automatic transformation from various application-specific measurement data formats into a common data format.
- It should be efficient and avoid repeating the same data processing operations if possible. For scalability, the framework should be based on cluster computing technology.
- The observatory should be a data vault in the sense that raw measurement data should never be deleted once added. Because raw data can be flawed it should be possible to mark them as invalid and not consider them for the analysis anymore.
- Every result of the observatory should be accompanied by detailed information on how the result was derived and which raw data has been involved in its generation.

- For the sake of repeatability, the observatory should also be able to provide results based on the data available at a given point in the past. If a part of it is considered invalid today, it should add a corresponding note to the result.

1.3 Code Archive

There is a zip file accompanying this thesis containing the complete code.

Chapter 2

Background

2.1 MongoDB

MongoDB[5] is a schema-less NoSQL database system. Records are stored in documents which are order-sensitive key-value stores. Documents can be nested and also contain arrays. Examples of documents can be seen in section 2.3 and 3.1.

A database consists of many collections and a collection consists of many documents. MongoDB supports clustered operation: data can be partitioned (sharding) and replicated over many machines (Figure 2.1). MongoDB is scalable and includes a powerful aggregation framework which will execute complex queries on all shards in parallel. For an aggregation pipeline example see listing A.5.

2.2 Apache Hadoop and Spark

Apache Hadoop[8] is a framework providing a fault-tolerant distributed filesystem and data processing services. In the observatory only the Hadoop Distributed File System (HDFS) is used.

Apache Spark[7] is a library for distributed data processing and is inspired by functional programming. In contrast to Hadoop it also supports interactive programming sessions (e.g. JupyterHub, Section 2.4). Spark is designed to run alongside Hadoop with the goal of exploiting data locality. A simplified view of the architecture is shown in figure 2.2.

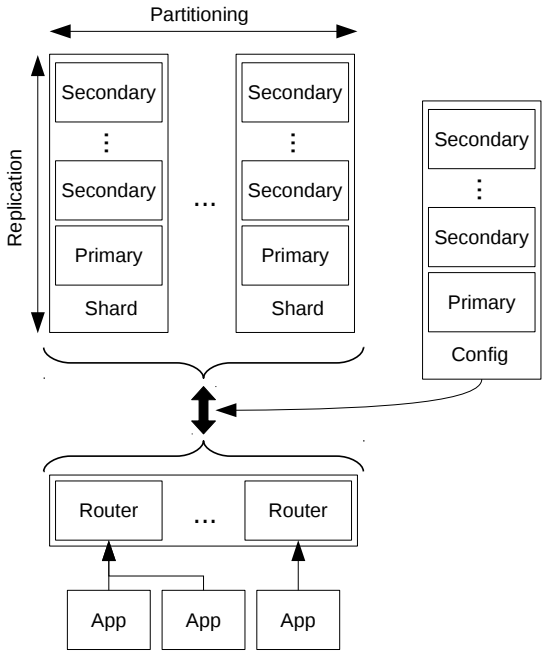

Figure 2.1: MongoDB Architecture: The application connects to a router which will issue queries to and collect responses from all shards with the help of the config shard. Each shard is a replica set consisting of a primary database server and several secondary database servers that mirror data from the primary database server. Query, aggregation and map-reduce-style operations are executed on the cluster and provide a high-performance data processing environment.

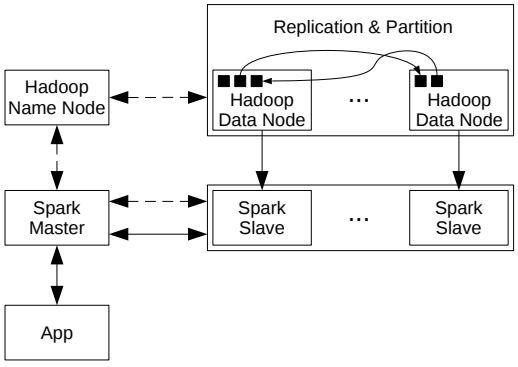

Figure 2.2: Apache Hadoop and Spark Architecture: The application connects to the Spark master and provides it with tasks. The hadoop cluster consists of a central name node and many data nodes. Data chunks are replicated according to a chosen policy (e.g. two copies on hard drives and one copy on a solid-state drive). The master builds an execution graph of the tasks and orders the Spark slaves to load data from the Hadoop data nodes. Spark tries to assign the subtasks to slaves that are close to the needed data (data locality). The result of a data processing task is then collected by the master and sent back to the application.

2.3 Observatory Upload

The Path Transparency Observatory is part of the MAMI research project[1] and will hold large amounts of measurement data relevant to path impairments. For uploading raw data into the observatory a RESTful HTTP upload service has been implemented by developers from ZHAW. The raw data files are kept in a HDFS storage, while the metadata is saved in a MongoDB collection. This is an example metadata document:

MongoDB Document ID (auto)
HDFS path to sequence file
Key in sequence file
Upload completed
Information supplied by uploader
Hash of raw data
Time of upload
User that uploaded the file
Action id of upload and current valid status for each environment. (Not part of Observatory Upload)

```
{
  '_id': ObjectId('5774a52d31e34a206bde8abc'),
  'path': 'hdfs://localhost:9000/uploads/
ecn-june/ecnspider1-zip-csv-ipfix/0000.seq',
  'seqKey': '20160629-a.zip',
  'complete': True,
  'meta': {
    'format': 'ecnspider1-zip-csv-ipfix',
    'msmntCampaign': 'ecn-june',
    'seq': '0000',
    'start_time': datetime(2016, 6, 29, 0, 0),
    'stop_time': datetime(2016, 6, 30, 0, 0)},
  'sha1': 'c950536c56327b4b883abd6c7ebd214ca13770b9',
  'timestamp': 1467262253,
  'uploader': 'gubser',
  'action_id': {'prod': 27, 'dev': 130},
  'valid': {'prod': True, 'dev': True}
}
```

2.4 JupyterHub

JupyterHub[3] is a multi-user interactive Read-Eval-Print-Loop (REPL) web interface running on the observatory server. Originally made for Python, it supports a multitude of programming languages today. The code is organized in notebooks and notebooks can be augmented with formatted text, plots and images. It is widely used for explorative data analysis and rapid prototyping.

2.5 Observation

An observation represents a statement that a condition has been observed on a given path at a certain measurement time. Depending on the condition, the observation may also have additional information attached.

A path consists of path elements which can be anything from IP addresses, subnets, AS numbers and more. A `'*'` denotes that there is/are one or more unknown path elements.

- `['1.1.1.1', '*', '2.2.2.2']`
  Path from an IP version 4 endpoint over an unknown number of intermediate devices to an IP version 4 endpoint.
- `['aaaa:aaaa::12', '*', '000b::12']`
  Path from an IP version 6 endpoint over an unknown number of intermediate devices to an IP version 6 endpoint.
- `['*', '2.2.2.2']`
  Path from one or more unknown endpoints and intermediate devices to an IP version 4 endpoint.

- `['1.2.3.4', 'AS36', '2034:db8:1::33', '2034:db8:2::/48', '9.10.22.33']`
  A complete route from an IP version 4 endpoint over an AS network over a IP version 6 gateway through a IP version 6 subnet to an IP version 4 endpoint.
- `['tag:123123.sources.observatory.mami-project.eu, 2016:c2756bf487aa826b0b3b13e833f352f26625ee5b']`
  A pseudonymous path using RFC4151 tags[9].

Chapter 3

Observatory Design and Implementation

The observatory architecture is depicted in figure 3.1 and consists of a raw data storage and an upload metadata collection as well as a collection of observations. The dashed lines indicate references to input data in order to be able to show the origin of observation and raw data. Raw data from various data sources is uploaded through the upload interface (Section 2.3) and is stored on a distributed filesystem (Section 2.2). The metadata of a raw data upload, such as data format and the time of measurement, are stored in the MongoDB uploads collection (Section 2.1). The observatory application itself is a small, easily installable Python 3.5 package and includes unit tests and docstrings (Section 1.3).

The analyzer engine (Section 3.3) governs access to the storage and to computing services such as Apache Spark (Section 2.2). Because of the diverse nature of input formats ranging from low-level packet capture to high-level A/B-testing (e.g. PATHspider, Section 4.1.2), the actual data transformation logic is not integrated into the observatory core itself. Instead it happens in separate, versioned executables/scripts called *analyzer modules*. The analyzer engine supervises the execution of analyzer modules which create observations based on the available raw data and existing observations. All observations are stored in the observations collection.

The query engine (Section 3.4) performs higher-level analysis on top of these observations. There is no direct contact with raw data anymore and it supports executing a query for a specific version of observations.

The Hacking Console is a JupyterHub service (Section 2.4) for explorative data analysis (Section 3.3.4) and the development of analyzer modules in a REPL environment. REST-APIs are used to control the operation of the system: Currently there are the upload interface (Section 2.3) and the admin interface for managing analyzer modules and upload validation (Section 3.3.3).

Every time a piece of raw data is uploaded, the observatory needs to make sure that the observations reflect the new data situation. Regenerating all observations each time something has changed is not a good solution because it will not be able to process terabytes of data in a reasonable time without a huge computing infrastructure. A major goal of the observatory is therefore to be able to execute an analyzer module over only a subset of data such that it yields the same observations as if it would had been executed over the whole dataset (Section 3.2). A subset that satisfies this is called a *suitable subset* in this thesis.

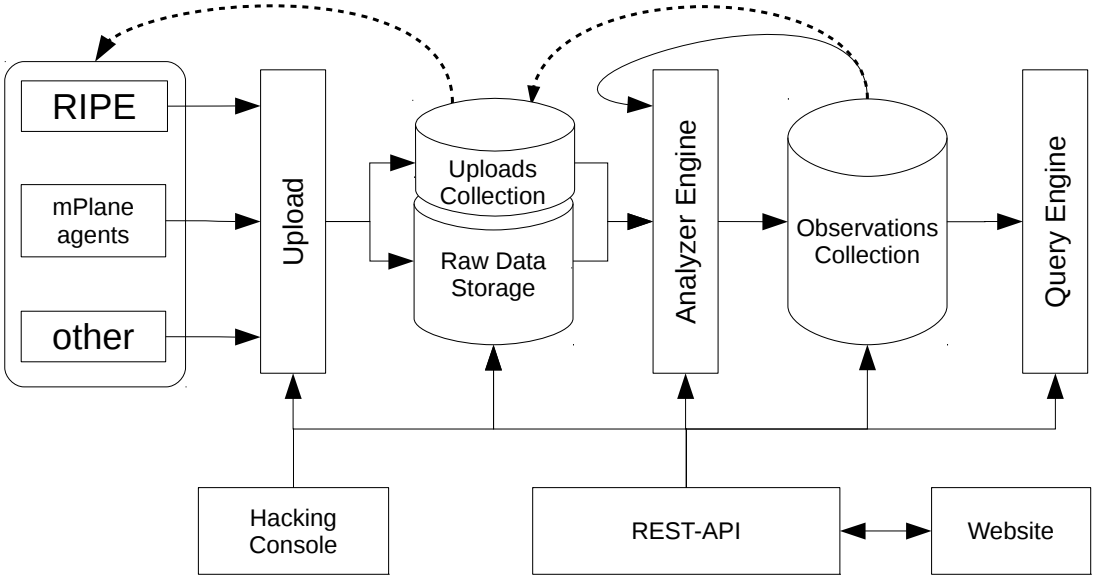

Figure 3.1: Overview of the observatory architecture (Chapter 3). Raw data is uploaded via the upload interface and transformed into observations in the analyzer engine. The query engine allows querying for a specific version of observations. The outside world is communicating via a Hacking Console and REST-APIs with the observatory components.

3.1 Observation

Based on the concept of an observation (Section 2.5), this is how the data is stored in the observations collection:

MongoDB Document ID (auto)
List of conditions (string)
List of path elements (string)
Time interval
Associated Value (document)
Analyzer Module ID (string)
List of documents, each with a valid status and an action_id when the status change happened
Hash for comparisons
Reference to source actions

```
{
  '_id': ObjectId('5790f03efed8f54132d620ac'),
  'conditions': [
    'ecn.connectivity.works',
    'ecn.negotiated'
  ],
  'path': ['188.166.146.182', '*', '199.91.102.136'],
  'time': {
    'from': datetime(2016, 6, 28, 9, 29, 37, 185000),
    'to': datetime(2016, 6, 28, 9, 29, 42, 234000)
  },
  'value': {},

  'analyzer_id': 'analyzer-ecnspider1',
  'action_ids': [
    {'id': 85, 'valid': True},
    {'id': 82, 'valid': False},
    {'id': 66, 'valid': True}
  ],
  'hash': 'We6Qgw3+FhfrTNBuJkPYC0Hsn4E=',
  'sources': {'upl': [14], 'obs': []}
}
```

The reason why the field `conditions` in an observation is a list instead of a single value as described in section 2.5 is because 1.) this keeps a link between directly related conditions from the same analyzer module, 2.) this is as convenient to query as if only a single condition were allowed because the query language of MongoDB does not differentiate between a single value and an array and 3.) this is more storage efficient because normally, multiple conditions arise that have the same path, time, value, etc. (Note that argument 3 is weakened by the fact that MongoDB applies collection-level compression).

To simplify description, the condition will denote the *observation type* in the rest of this thesis.

The field `action_ids` is an array of subdocuments with fields `id` and `valid`. Each time the validity of an observation changes, a new subdocument is pushed to the front of the array (Section 3.2.3). So the first element always signifies the current valid status (Figure 3.2).

The field `hash` contains an SHA1 digest for comparing observations. This intermediate step is necessary because comparing documents in MongoDB is order-sensitive. See section 3.2.3 for details.

The field `sources` contains a list of `action_id`s of raw data uploads in (`upl`) and a list of ids of observations in (`obs`). There are three kinds of observations: *direct observations* are inferred from a single raw data upload, *basic observations* are inferred from one or more raw data uploads and *derived observations* are inferred from zero or more raw data uploads and one or more observations. With the information in `sources` and the action log (Section 3.2.1) it can be derived which raw data uploads have influenced the creation of the observation.

The field `value` is a document whose shape is depending on the conditions. Each condition requires a set of mandatory fields. An observation with multiple conditions needs to have the union of all fields required by each condition.

3.2 Subset Analysis

In this section, a concept is presented to perform analysis on a subset of input data such that it yields the same observations as if it would had been executed over the whole data.

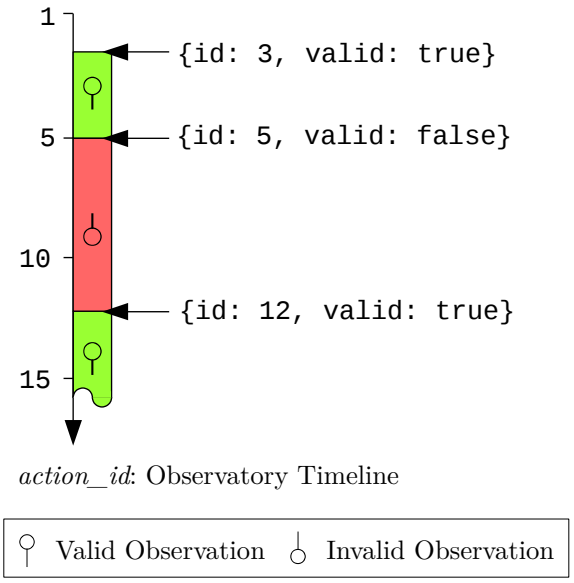

Figure 3.2: Changing of the valid status in the course of the observatory lifetime. The observation was added at action `3`, marked invalid at `5` and marked valid again at `12`.

In the Internet measurement scope, most analysis procedures try to assess the state of the Internet for a given time instance or over a given period of time. For example, analyzer modules can compare flows within some proximity of time, aggregate events over specific time periods or assess performance values month after month. Indeed, the measurement time span seems to be a reasonable candidate to define a subset because all measurement datasets have a start time and a stop time by nature.

In essence, the concept works as follows: the relevant input and output actions for each analyzer module are mapped onto the measurement time axis. The time spans of input actions that are not covered by the time spans of output actions define the time spans that contain unprocessed data. Without knowing something about the methodology of the analyzer module it is not possible to determine a reasonable suitable subset. That is why the analyzer module is asked to select and modify the time spans such that they form a suitable subset. The observatory provides three methods (*basic*, *extend* and *margin*) at the analyzer module's disposal (Point 4 in Section 3.2.2), but it can also implement its own methods.

For determining if the observatory is up-to-date, all actions that have an effect on the raw data and observations are recorded in an `action_log` (Section 3.2.1). To build and interpret it, the raw data and analyzer modules need to provide the following information:

The metadata of each upload of raw data has a format identifier and a measurement time span.

The analyzer modules declare what raw data formats and what types of observations they use as input as well as what types of observations they generate as output. These three sets are called `input_formats`, `input_types` and `output_types` respectively and they are for determining which actions are relevant to the analyzer module.

After selecting a suitable subset in terms of time spans, the analyzer module processes the input data and stores the generated observations into a temporary collection. The content of the temporary collection is then compared with the existing observations within the time spans and committed into the observations collection (Section 3.2.3).

3.2.1 Action Log

The `action_log` is a collection that stores information about all actions that had an effect on the data within the observatory. Each action is identified with an incrementing sequence number

called `action_id` and a list of time spans that define the range of data that are affected by this action.

There are four different actions:

- `upload` indicates that raw data has been added. Mandatory fields are: the format identifier of the raw data and a reference to the upload entry from the upload engine.
- `analyze` indicates that an analyzer module has been executed. Mandatory fields are: the version of the analyzer module and the `output_types` it has generated.
- The special action `marked_invalid` is used to declare that an upload should not be trusted anymore. As stipulated in section 1.2, deleting an upload is forbidden. Uploads that are invalid will be ignored by the analyzer modules and as a consequence the observations based on the invalid upload will also become invalid (Section 3.2.3).
- The inverse action of `marked_invalid` is `marked_valid`. Initially, an upload is valid and the action `marked_valid` is only necessary when the upload has been `marked_invalid` before.

3.2.2 Determining Subsets

Normal and derived observations can depend on multiple uploads and/or observations. While the method presented in this section also works for direct observations, it can be impractical. An optimization for this case is presented in section 3.2.4.

For each analyzer module generating normal or derived observations, the following procedure is carried out:

1. A list is compiled of every upload or (in)validation of raw data that matches the analyzer module's `input_formats` and every execution of analyzer modules that generate observations that matches the analyzer module's `input_types`. This list is called *input actions* and each action is associated with a list of time spans.
2. A list is compiled of all recent executions carried out by the current version of the analyzer module. This list is called *output actions* and each action is associated with a list of time spans and the `max_action_id` that was considered as input at the time of execution (explained in step 4). Note that if the version of the analyzer module has changed, the list of output actions is empty because upgrading an analyzer module will need all data to be processed again.
3. The two lists are combined and sorted by the time the action was carried out. Beginning with the earliest action, its timestamps are entered or removed from a timeline as illustrated in figure 3.3. The resulting time spans in the timeline describe the subsets of data that have not been analyzed yet.
4. Now, the analyzer module itself is asked to choose and modify the time spans such that they define a suitable subset. The analyzer module is allowed to choose to process a fraction of all unprocessed data as long as this fraction is a suitable subset. This can be advantageous for example to keep the execution time at a reasonable level. (The observatory will just execute the analyzer module again if it notices that there is still unprocessed data.)

   For convenience, the observatory provides three different methods at the analyzer module's disposal. Namely, *basic*, *extend* and *margin* which are illustrated in figure 3.4. Method *basic* does not change the time spans at all, method *extend* stretches the time spans to the next absolute boundary defined by a time period (full hour, full day, full month, etc), and the third method *margin* stretches the time spans such that there are no measurements outside of the boundaries within a given offset. This last method is explained in more detail in the next section.

   To avoid problems when new data is added while the analyzer module is running, the analyzer module also has to declare the last action it considers for input. This number is called `max_action_id`.

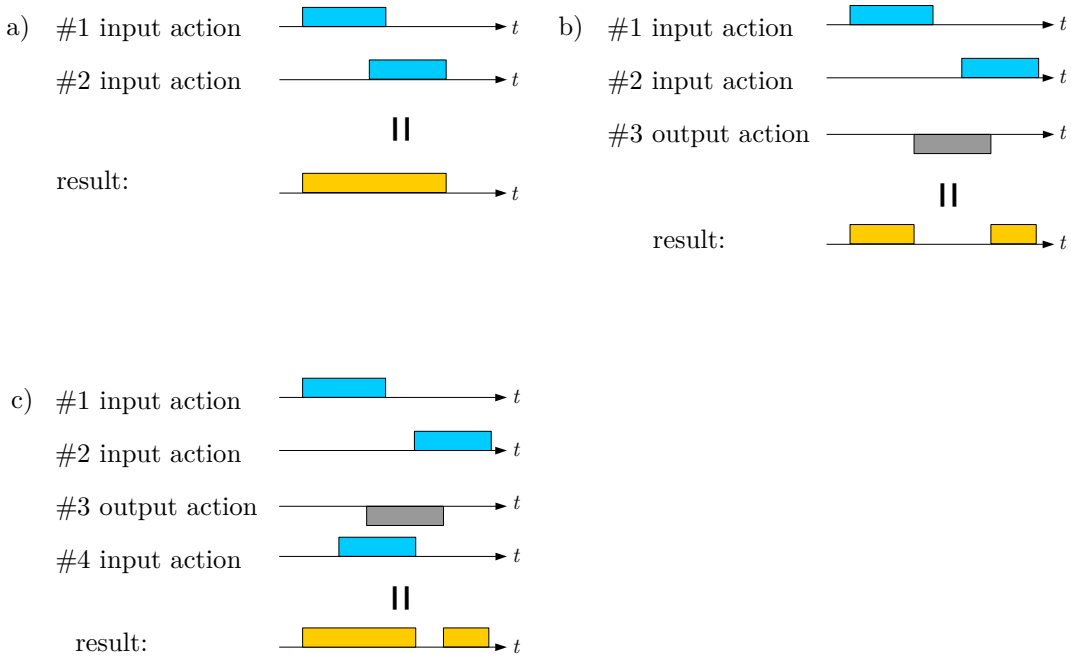

Figure 3.3: An additional input action increases the total amount of time while an additional output action decreases it. In situation a) the analyzer module has not been executed yet and the resulting timeline consists of the union of the input time spans. In b) the analyzer has been run over a subset already (#3) and thus the remaining timeline consists of two smaller time spans. The order of actions is important because in c) the input #4 is added after the execution of the analyzer module and thus this input's time span has to be evaluated again by the analyzer module.

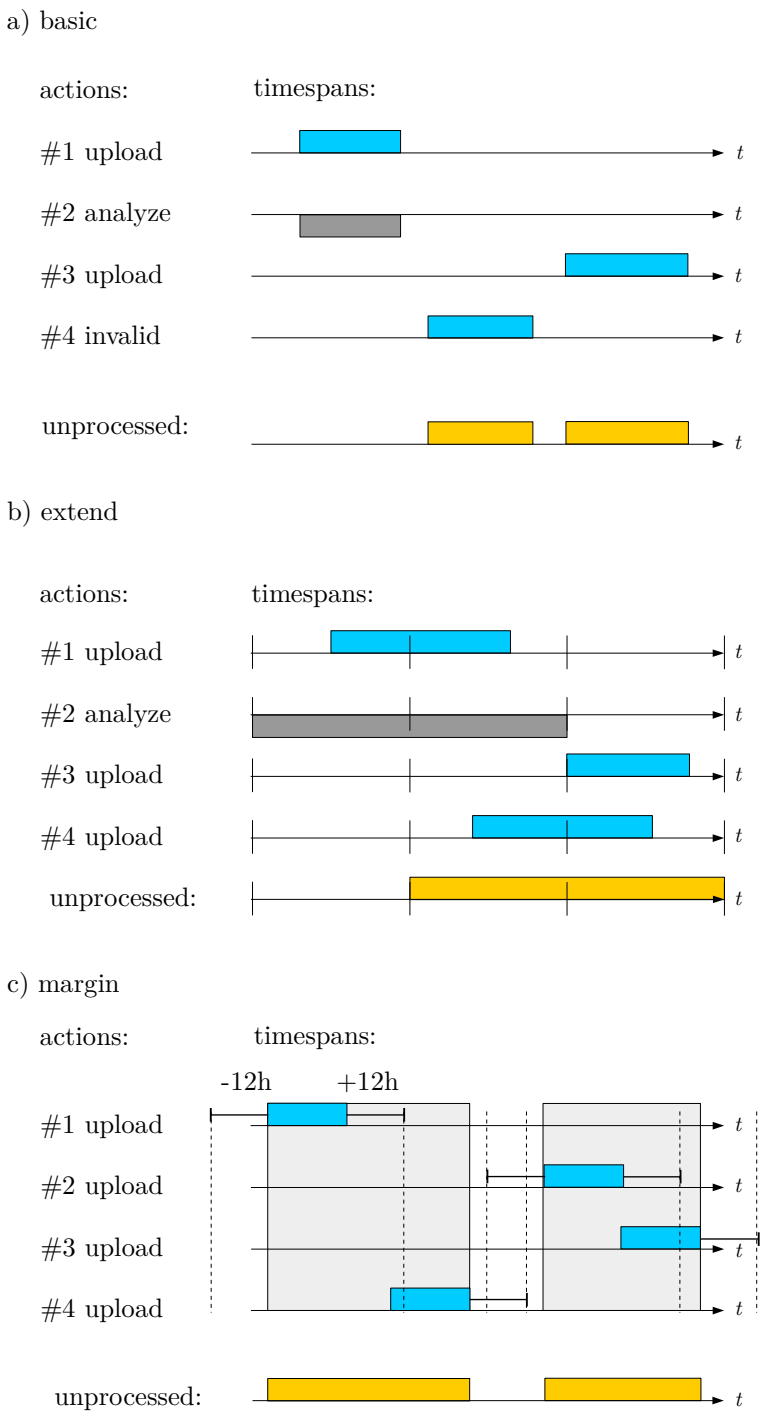

Figure 3.4: Three methods for achieving a suitable subset are implemented: In a) all time spans that weren't analyzed are considered to be unprocessed (no modification). In b) time is slotted according to a custom period. The time spans are expanded to match the slot edges. In c) time spans are expanded in a way that there are no measurements outside of the selected time spans within a custom margin.

Margin Method

With the margin method, time spans are expanded in a way that there are no other measurements within a given offset outside of selected time spans (Figure 3.4 c)).

First, all time spans are added to a set `pool`. Another set, `islands`, is initialized empty. Each island will define a suitable subset. Figure 3.5 illustrates the following steps. The small blue rectangles are the time spans in `pool` and the grey rectangles are the time spans in `islands`.

1. A time span is popped from `pool` and its `begin` and `end` time are extended by `offset` (in this case $\pm 12h$).

2. Each remaining timestamp of `pool` that is intersecting the extended time span will be popped. The minimum `begin` and the maximum `end` time of all intersecting time spans is determined and extended by `offset`. This process is repeated until the minimum `begin` and the maximum `end` time do not change anymore or `pool` is empty.

3. The resulting time span is called an island and is added to the set `islands`. The whole process is repeated from step 1 until `pool` is empty.

3.2.3 Execution and Commit

Once the analyzer module has chosen a suitable subset of input data in terms of time spans, it processes all valid data within these time spans and stores the generated observations in a temporary collection. The corresponding set of all observations in the observations collection generated by the same analyzer module within the same time spans is called the *candidate set*.

For each observation in the temporary collection, the `hash` field is computed by the SHA1 algorithm from the flattened and alphabetically sorted fields `conditions`, `time`, `path`, `value`, `sources` and `analyzer_id`.

For each observation in the candidate set the hash is computed in the same way. Then for each candidate, its hash is matched against the hashes of all observations in the temporary collection. If there is a match, the observation in the temporary collection is called the *counterpart* of the candidate. For speeding up the counterpart search, the field `hash` is indexed.

Why compute hashes? There are two reasons why the hash is computed by the observatory instead of using MongoDB's built-in features: 1.) MongoDB document comparison is order-sensitive but Python's `dict` is order-insensitive. This means that executing the same analyzer modules twice without using order-sensitive mapping types everywhere will create different documents because the order of the elements is non-deterministic. 2.) Hashed index over multiple fields with some fields being an array is not supported[6]. This means querying for counterparts would be unbearably slow.

After determining all counterparts, the observations are compared. These are all the possible comparison outcomes (Figure 3.6):

1. *add*: When the analyzer module generates an observation that isn't the counterpart of any candidate, the observation is added to the observations collection. The array field `action_ids` is initialized with `[{'id': <current action id>, 'valid': true}]`.

2. *mark invalid*: When the analyzer module does not generate an observation and there is a valid candidate observation, the existing observation is marked invalid: A new element `{'id': <current action id>, 'valid': false}` is prepended to the array `action_ids` of the candidate.

3. *mark valid*: When the analyzer module generates an observation and there is an invalid candidate observation, the candidate observation is marked valid. A new element `{'id': <current action id>, 'valid': true}` is prepended to the array `action_ids` of the candidate.

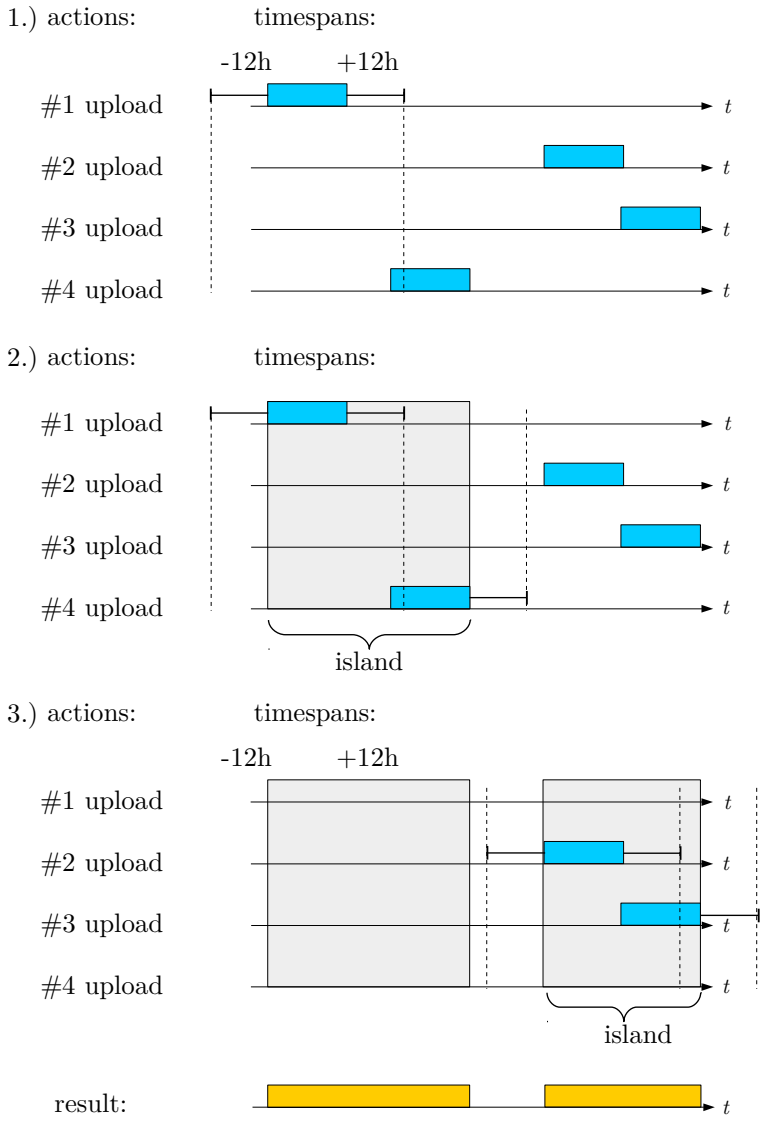

Figure 3.5: Determining suitable subsets with the margin method (Section 3.2.2).

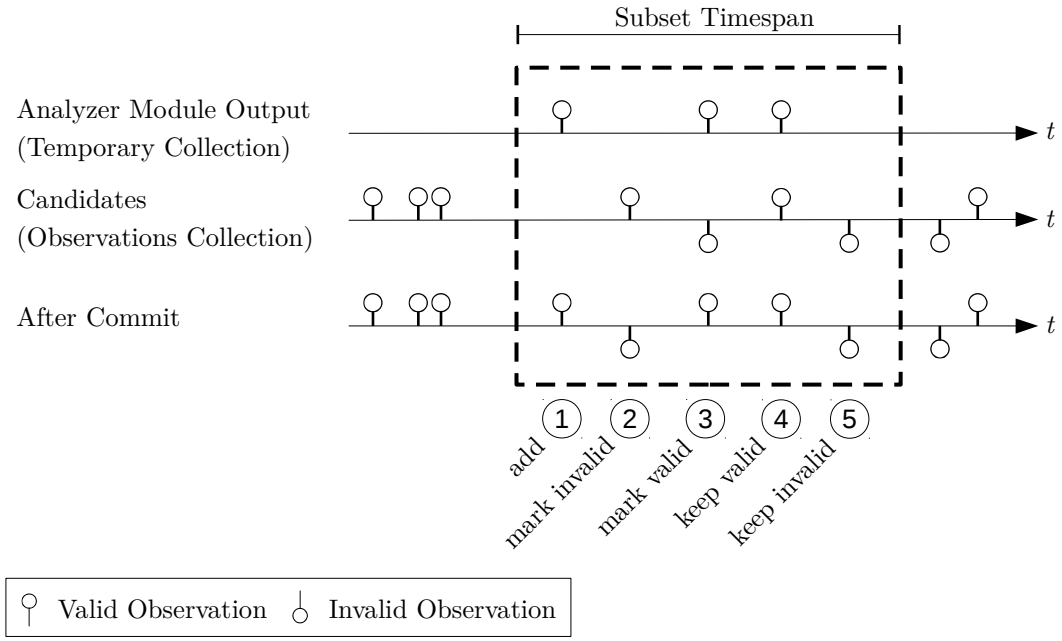

Figure 3.6: When the input data changes, the analyzer module will eventually produce different observations than before. The five possible outcomes are illustrated here and described in section 3.2.3.

4. *keep valid*: When the analyzer module generates exactly the same observation, the candidate observation is not changed·

5. *keep invalid*: When the analyzer module does not generate an observation and there is an invalid candidate observation, the candidate observation is not changed.

The information stored in the field `action_ids` enables querying the observatory for results at a given time in the past (Query engine, section 3.4).

3.2.4 Optimization for Direct Observations

It's worthwhile to look at this special case because it is expected that many analyzer modules will be generating direct observations only. Direct observations depend, by definition, only on a single upload (Section 3.1), and therefore each upload already forms a suitable subset. The analyzer module has to explicitly state that it only creates direct observations by setting the field `direct` to true in the analyzer module specification file (Section 3.3.3, Listing A.2).

When many uploads are in close proximity of measurement time to each other and another upload is added, the time spans approach presented in the previous section will force the analyzer module to analyze all uploads again although only the newly added upload would need to be analyzed.

The first two steps are identical to the procedure in section 3.2.2:

3. A list of (upload, `max_action_id`) is derived from the input actions list where `max_action_id` is the `action_id` of the last action related to the upload. Using the last action takes into account that there can be multiple `marked_valid` or `marked_invalid` actions after an `upload` action.

4. Remove those uploads from the list that were analyzed by an analyzer module with an `action_id` larger than the last action related to the upload.

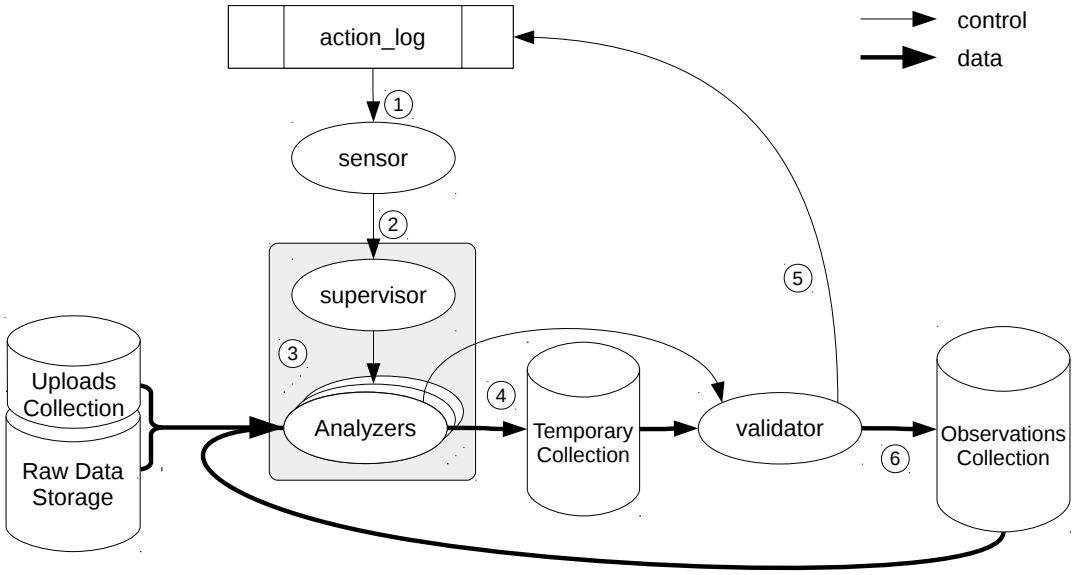

Figure 3.7: The sensor detects changes and orders the supervisor to execute analyzer modules which store their generated observations in their respective temporary collection. Afterwards, the validator takes the observations and commits them to the observatory. The details 1-6 are described in section 3.3.

3.3 Analyzer Engine

The analyzer engine consists of three services: sensor, supervisor and validator. In addition, there is an admin interface designed as a REST-service for registering analyzer modules and marking uploads (in)valid (Section 3.3.3). The process of detecting changes, executing analyzer modules and committing results into the observations collection is illustrated in figure 3.7. Analyzer modules are assigned a state (Figure 3.8) and depending on the state, they are being handled exclusively by one of the services. The state is stored in a MongoDB collection, where the sensor, supervisor and validator have access. There is no direct communication channel between them.

The numbers ① - ⑥ refer to both figures 3.7 and 3.8.

Sensor In the *sensing* state, the sensor (Section 3.3.1) periodically scans the action log ① and determines if the analyzer module has unprocessed data. When the sensor decides that there is unprocessed data or the analyzer module itself has changed, the sensor changes the state from *sensing* to *planned* ②.

Supervisor Now the analyzer module is in the domain of the supervisor. The supervisor will set up a temporary collection, grant access to the observatory services, put the analyzer module from *planned* into *executing* state and invoke the analyzer module (Section 3.3.2). The analyzer module chooses a suitable subset as described in point 4 of the list in section 3.2.2, generates observations from its input data and stores them into the temporary collection. When the analyzer module ended without error, the supervisor sets the state from *executing* to *executed* ④. If there was a problem (e.g. non-zero exit code), the analyzer module is put into the *error* state instead and has to be enabled by the user via the admin interface (Section 3.3.3).

Validator The validator periodically checks for an analyzer module that is in the *executed* state. It takes the observations from the analyzer module's temporary collection, validates them, adds an `analyze` action into the action log ⑤ (Section 3.2.1) and commits the observations into the observations collections as described in section 3.2.3 ⑥. If everything went well, the analyzer module is put into state *sensing* again.

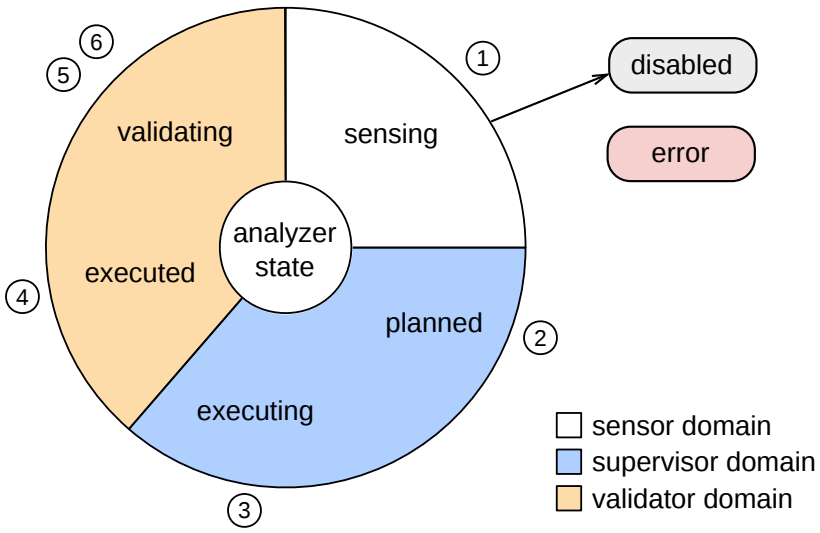

Figure 3.8: Executing an analyzer module and integrating the results into the observatory involves several steps which are descirbed in section 3.3.

To avoid having to implement locking mechanisms, the validator is the only service that writes to the action log.

Consequently, it is also responsible for inserting `upload`, `marked_invalid` and `marked_valid` actions. It periodically scans the uploads collection for completed uploads and 1.) assigns each upload document an `action_id` and a `valid` field (See section 2.3) and 2.) inserts an action into the action log (Section 3.2.1). In the observatory configuration, a custom query can be specified to, for example, only consider raw data uploaded by certain users or certain raw data formats. Marking an upload (in)valid is a manual process initiated by the user via the admin interface (Section 3.3.3).

3.3.1 Sensor

Most of the time, analyzer modules are in the *sensing* state. Upon request, the sensor can also set analyzer modules to *disabled* and it will ignore them (Section 3.3.3). In the *sensing* state, the sensor periodically scans the action log and determines if there is unprocessed data by comparing input and output actions: The time spans that contain unprocessed data are the time spans of input actions that aren't covered by the time spans of output actions (Section 3.2, Figure 3.3). Notabene, the sensor does not need to execute the analyzer module to determine if there is unprocessed data and therefore avoids the overhead of execution.

But not only the input data can change, also the analyzer module itself is allowed to change. The filesystem representation of an analyzer module is in fact a git repository[11]. The version information in the `analyze` action in the action log (Section 3.2.1) is comprised of an url to a repository and the commit hash. If the version of the last analyzer module run is different from the current analyzer module version, the sensor decides that the whole input data has to be analyzed again.

The encapsulation into analyzer modules also allows the observatory to execute multiple of them simultaneously. There are two important reasons as to why parallel execution is preferable to sequential execution: 1.) It reduces latency because short-running analyzer modules are not delayed by long-running ones. 2.) When an analysis algorithm cannot be split up in enough distributed tasks such that it cannot fully occupy the cluster computing infrastructure, the utilization can be increased by running multiple, independent analysis algorithms at the same time.

But when running multiple analyzer modules, the sensor needs to make sure that no input data is modified during execution of an analyzer module. More precisely, it is not allowed to start an

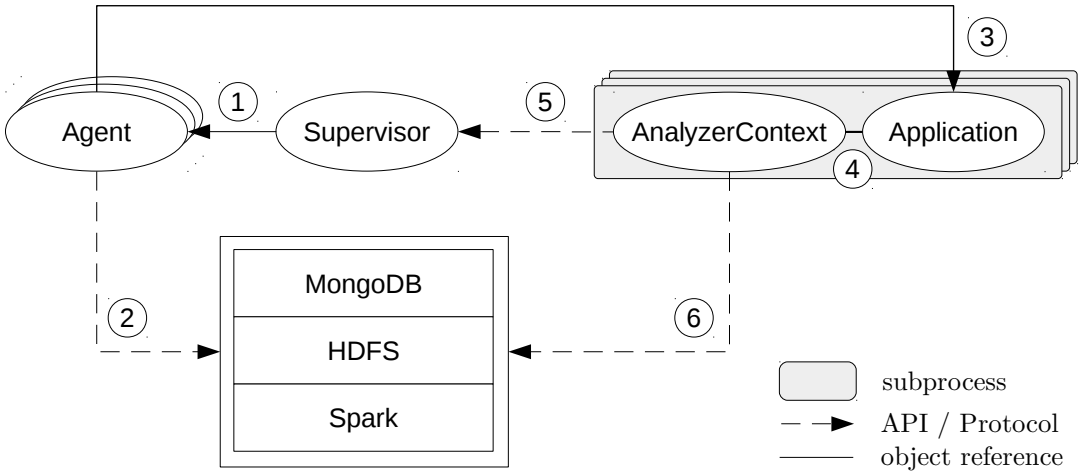

Figure 3.9: Steps involved when executing an analyzer module or an online analyzer. Section 3.3.2

analyzer module that has a *blocked* output type or an *unstable* input type: An observation type is *blocked* if it's an input type of a running analyzer. And correspondingly, an observation type is *unstable* if it's an output type of a running analyzer.

When the sensor decides that an execution is necessary and allowed, the sensor changes the state from *sensing* to *planned*.

3.3.2 Supervisor

The steps involved in the execution of an analyzer module are illustrated in figure 3.9. When the supervisor detects an analyzer module in the `planned` state, it creates an agent ①, which will create a database user and a temporary collection for the analyzer module ②. This step can be extended to create accounts for Apache Spark, HDFS and other services. Currently, Spark and Hadoop's authentication are disabled due to its high initial setup effort.

Then, the agent executes the command line given by the analyzer module ③. The application logic creates an `AnalyzerContext` (the observatory's API for analyzer modules) ④, which in turn connects to the supervisor via a token-based authenticated TCP connection and exchanges line-separated json-encoded messages ⑤. The token is given by the agent via an environment variable.

There are other ways to exchange information between analyzer module and supervisor. For example standard input & output, text files, MongoDB collection etc. But the advantage of the chosen method is that the code difference between analyzer module code and an interactive console is very small, which makes developing analyzer modules easier (Section 3.3.4). Also, the API is easy to extend with more sophisticated functionality (Section 6).

For convenience, the `AnalyzerContext` hides most of the complexity and provides functions to choose a suitable subset and gain access to storage and computing services ⑥ (Listing A.1). The fact that the supervisor executes the analyzer module via an ordinary command line and the exchange of messages is following a simple protocol should make it easy to port the `AnalyzerContext` to different programming languages.

3.3.3 Admin Interface

The admin interface is a simple REST API powered by the flask microframework. Registering an analyzer module requires only two pieces of information: an URL pointing to a git repository and a commit hash. The remaining information such as `input_formats`, `input_types` and

`output_types` is retrieved from a special specification file in the repository called `ptocore.json` (Example Listings A.2 and A.3). This makes deploying analyzer modules very convenient. The admin interface allows to enable/disable/cancel and change the version of an analyzer module as well as marking uploads (in)valid.

There is one important drawback in the current implementation: Because the `valid` field in the upload collection only holds the current valid status (Section 2.3) and is updated instantaneously by the validator, the analyzer module will not notice when an upload has to be marked (in)valid. A workaround is that the user manually needs to make sure that there are no analyzer modules running. A solution to this is proposed in chapter 5.

Listing A.4 contains a sample code for interfacing with the API using the `requests` library[12].

3.3.4 Interactive Development

Often, the first steps for exploring a new measurement dataset happen in interactive environments such as JupyterHub (Section 2.4). The observatory is designed to make the code transition from these environments to analyzer modules as easy as possible.

Apart from analyzer modules, the supervisor supports so-called *online analyzers*. The main difference is that online analyzers are not executed by the supervisor and cannot contribute to the observations collection. Instead, the supervisor creates credentials upon request, which can be passed manually to an `AnalyzerContext`. After that, everything is the same except the observations stored in the temporary collection are discarded as soon as the online analyzer account is deleted.

In the current implementation, the supervisor creates an online analyzer on startup and prints the credentials to standard output.

3.4 Query Engine

Querying for a specific version of the observations collection is possible in an efficient manner by inspecting the field `action_ids` using the MongoDB aggregation framework.

First, the desired version (= `action_id`) of the observation is chosen. Then, for each observation, the field `action_ids` is filtered, keeping only observations that existed prior to the chosen `action_id` and that were valid at that time.

The aggregation pipeline implementing querying for a specific version of the observatory is shown in Listing A.5.

Chapter 4

Proof Of Concept

To show that the observatory works as expected, a small measurement study about ECN connectivity and negotiation has been conducted.

The idea is that measurements are performed with two different versions of the ECN analysis tool developed at ETH: ECNspider (Section 4.1.1), the software used in the 2014 ECN study[13], and the new PATHspider (Section 4.1.2), a rewrite of ECNspider developed within the MAMI project.

These tools measure the same thing but output completely different data formats. The use case of the observatory is to transform the results of both tools into observations of a common shape and then perform further analysis on top of observations independent of the raw data format (Section 4.2).

4.1 Background

Explicit Congestion Notification[10] (ECN) is a TCP/IP protocol extension aiming to reduce packet loss, latency and jitter. When there is congestion on the path between two endpoints, the router experiencing the congestion can set a flag in the IP header which will be echoed back from the receiving endpoint to the sending endpoint in a TCP flag. The sending endpoint will then reduce its packet transmission rate and thus reduce the load on the congested router. To be effective, both endpoints have to negotiate the use of ECN. This extension was disabled by default in operating systems because old and faulty hardware have broken connectivity[13].

Two numbers are of primary interest: The probability that connectivity breaks when ECN is enabled and the proportion of hosts that actually negotiate ECN.

For each host, two connection attempts were made: the first without ECN and the second with ECN (Figure 4.1). Depending on whether the connection was successful, the measurement is classified into *ECN safe*, *ECN broken*, *nobody home* and *transient/other* according to table 4.1. Further, the classification whether ECN is *negotiated* or *not negotiated* is done by looking directly at the SYN packets.

When performing the same measurement from different vantage points, it can be determined if the connectvitiy is path or site dependent (Figure 4.2).

Connection	w/o ECN	w/ ECN
ECN safe	success	success
ECN broken	success	failure
nobody home	failure	failure
transient/other	failure	success

Table 4.1: Characterization of ECN safety based on the results of two connection attempts.

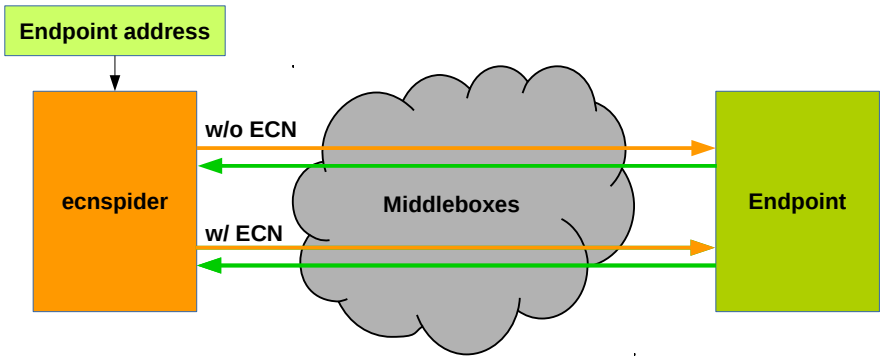

Figure 4.1: ECN connectivity test, first connect without the ECN feature and second connect with the ECN feature enabled. Image from [14]

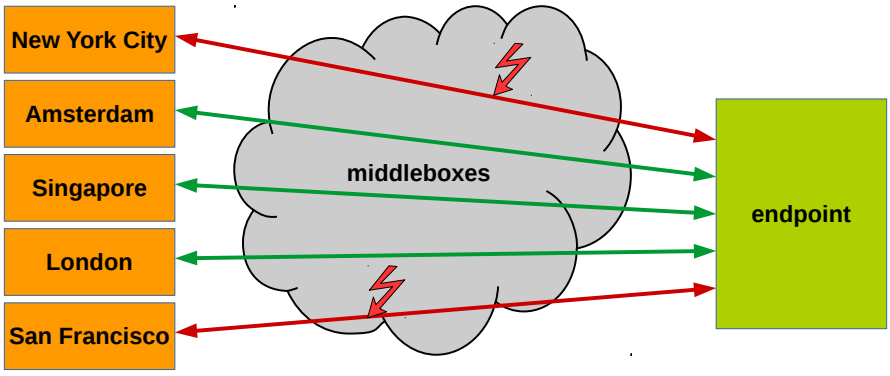

Figure 4.2: Measurement performed from multiple vantage points. When some but not all vantage points fail to establish an ECN-enabled connection, the connectivity is path-dependent. Image from [14]

4.1.1 ECNspider

Performing a measurement with ECNspider[13] requires recording flows using Quality of Flow[15] (QoF) at the same time. ECNspider outputs a csv file with each line corresponding to a connection trial with the fields: destination ip, source port, ECN enabled, connection result, etc. QoF outputs an ipfix file with each record corresponding to a flow: an exchange of packets between two endpoints characterized by source ip, source port, destination ip, destination port. For this analysis, it records TCP/IP SYN-flags used later to determine if ECN was successfully negotiated.

Note that for assessing ECN support, the two files have to be combined and it has to be searched for the two connection attempts for each endpoint. This is not done by the ECNspider application itself but happens in a separate analysis code instead.

4.1.2 PATHspider

The new PATHspider[4] is a plugin-based generalization of ECNspider that integrates the functionality that was originally provided by QoF. The ECNspider specific logic has been refactored into a PATHspider plugin and is called *ecnspider3*[1]. The application outputs line-separated JSON-encoded values with each line corresponding to a connection trial.

4.2 Design Of The Study

Figure 4.3 shows the setup of measurement tools, analyzer modules and observable conditions. ECNspider and PATHspider raw data is uploaded by hand and is automatically analyzed by analyzer-ecnspider1 or analyzer-ecnspider3 (see below) respectively. Based on observations generated by them, another analyzer module analyzer-ecnspider-vp determines path- and site-transparency.

analyzer-ecnspider1 and analyzer-ecnspider3 produce direct observations (subset selected as described in section 3.2.4), whereas analyzer-ecnspider-vp produces derived observations (subset selected as described in section 3.2.2 with margin method). Note that analyzer-ecnspider-vp is raw data independent.

Due to the small production environment, just one single-core VM with 8GB RAM, it was found unreasonable to assess the performance of the cluster computing components, namely Spark and MongoDB. The proof of concept study therefore only assesses the functionality of the observatory.

Analyzer Module for ECNspider For assessing ECN support, the two files (Section 4.1.1) have to be analyzed jointly outside of the ECNspider application. The analysis code used in the original ECNspider has been refactored and integrated into the observatory as an analyzer module with the name *analyzer-ecnspider1*. It expects the raw data upload to be a zip file containing the two files: an `.ipfix`-file (from QoF) and a `.csv`-file (from ECNspider).

Amendments to PATHspider and Analyzer Module for PATHspider The ecnspider3 plugin of PATHspider has been enhanced as part of this thesis to automatically merge the two connection attempts and classify the test into *ECN safe*, *ECN broken*, *nobody home* and *transient/other*. In addition, the output format has been changed to resemble the shape of an observation (Section 2.5). The analyzer module reading PATHspider format is called *analyzer-ecnspider3* and is really simple because the analysis is already done in PATHspider.

Analyzer Module for Path- and Site-Dependency This analyzer module uses the margin method (Section 3.2.2) to select suitable subsets. Because the measurements were taken at roughly the same time one week apart, the suitable subsets produced by the margin method match the measurement runs.

The following procedure is implemented as a MongoDB aggregation pipeline:

[1] The reason it is version 3 is because the tool developed in [14] is ecnspider 2.

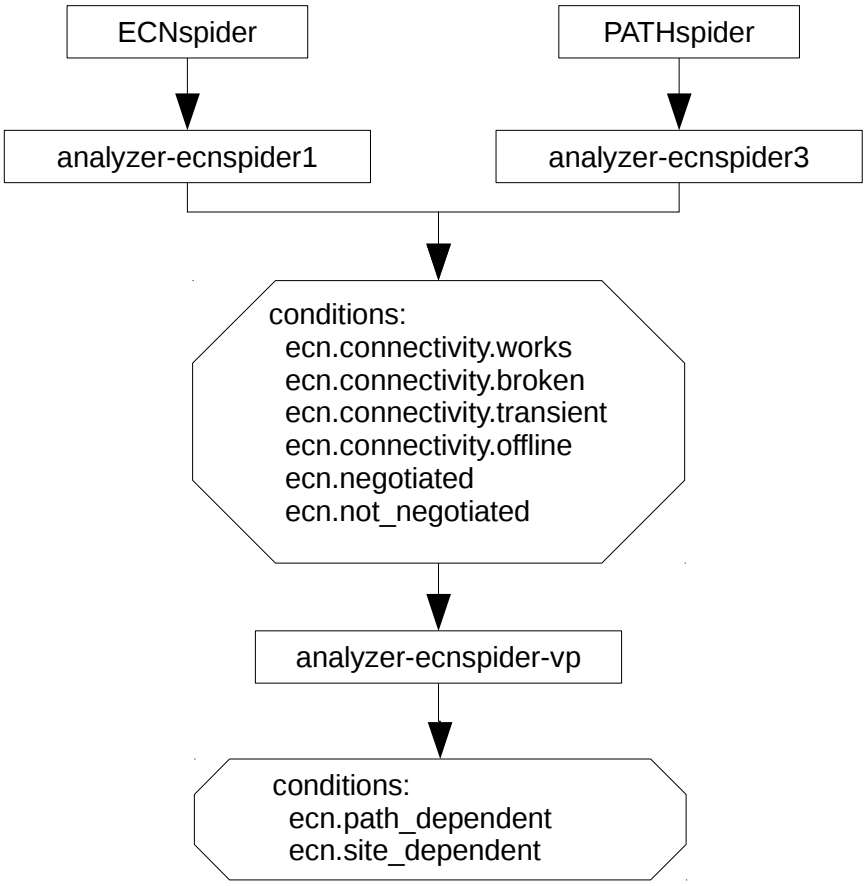

Figure 4.3: ECNspider and PATHspider generate raw data which is analyzed by three analyzer modules.

1. Observations are grouped by destination IP address. For each destination IP: source IPs and conditions are added to sets `sips` and `conditions` respectively.
2. At least three source IPs (vantage points) are required in `sips`; otherwise the record is ignored.
3. Then, the `conditions` set is analyzed for each destination IP address: If there is at least one `ecn.connectivity.broken`, the connectivity can be path dependent or site dependent.
4. If there is no `ecn.connectivity.works` then the connectivity is classified as `ecn.site_dependent`. Otherwise, if there is at least one `ecn.connectivity.works`, the connectivity is classified as `ecn.path_dependent`.

London	IP4			IP6			Both		
Date	total	broken	pct	total	broken	pct	total	broken	pct
2016-07-18	617602	249	0.04%	24910	6	0.024%	642512	255	0.04%
2016-07-27	613912	344	0.056%	25980	4	0.015%	639892	348	0.054%
2016-08-04	611782	243	0.04%	26871	5	0.019%	638653	248	0.039%
2016-08-10	610506	698	0.11%	27574	1	0.0036%	638080	699	0.11%

Table 4.2: ECN dependency measured at four dates from the London vantage point. Plotted in figure 4.4.

	IP4			IP6			Both		
Date	total	path	pct	total	path	pct	total	path	pct
2016-07-18	617602	444	0.072%	24910	6	0.024%	642512	450	0.070%
2016-07-27	613912	509	0.083%	25980	2	0.008%	639892	511	0.080%
2016-08-04	611782	465	0.076%	26871	5	0.019%	638653	470	0.074%
2016-08-10	610506	1121	0.184%	27574	14	0.051%	638080	1135	0.178%

Table 4.3: ECN dependency measured at four dates from the London vantage point. Plotted in figure 4.4.

4.3 Results

Four measurement runs were performed roughly one week apart from each other from three vantage points located in London, New York City and Singapore. The raw data is stored in the observatory[1] and the analysis has been done there as well.

Comparing the percentage of ECN dependent connectivity (Table 4.2, Figure 4.4) with the numbers in the study[13] the values are smaller by a factor of 4 to 10. My assumption is that there is a bug in the analyzer-ecnspider1 code such that broken connectivity is not detected properly. Obviously this also affects the classification into path- & site-dependency (Table 4.3, Figure 4.5).

The percentage of successful negotiation (Table 4.4, Figure 4.6) aligns with the results in the study.

Each measurement run at a vantage point tested roughly 600'000 endpoints which resulted in a total of 7.2M connectivity tests. The compressed measurement raw data occupies 1.5 GB of disk space. In total, there exist now 7'042'284 connectivity observations, 1'975'740 negotiation observations and 1514 path- & site-dependency observations which results in a total of 9'019'538 observations.

Unfortunately, it was not possible to perform measurements with PATHspider due to a spurious bug. Nonetheless the analyzer-ecnspider1 and analyzer-ecnspider-vp successfully analyzed the data produced by ECNspider. Also marking raw data invalid is functioning and the invalid status propagates correctly to all affected observations.

Date	total	nego	pct
2016-07-18	642512	459122	71.457%
2016-07-27	639892	460562	71.975%
2016-08-04	638653	461247	72.222%
2016-08-10	638080	461362	72.305%

Table 4.4: Number of successful ECN negotiation measured at four dates from the London vantage point. Plotted in figure 4.6.

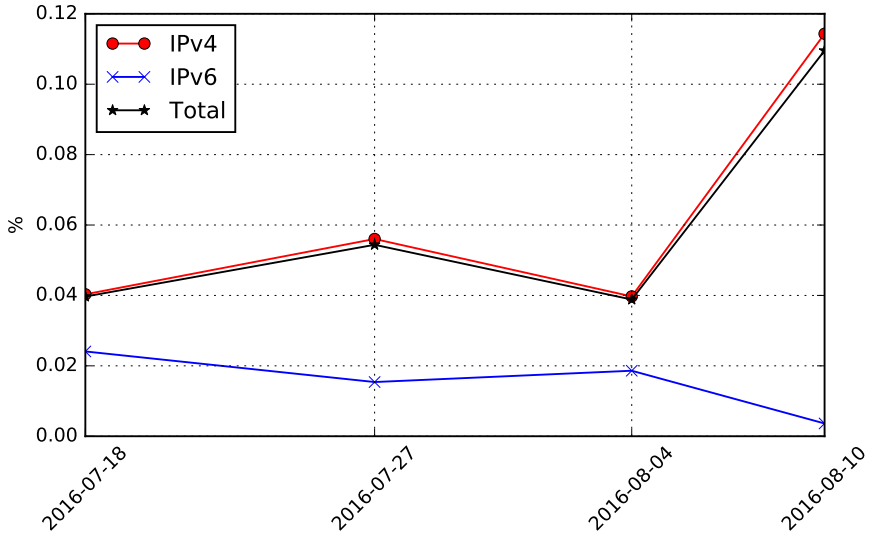

Figure 4.4: ECN dependency measured at four dates from the London vantage point.

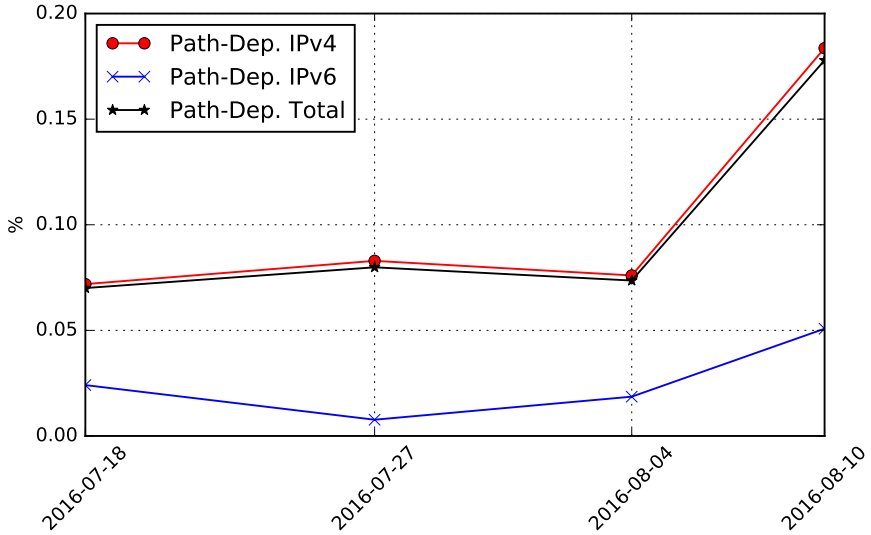

Figure 4.5: ECN path dependency determined by comparing connectivity tests from all vantage points.

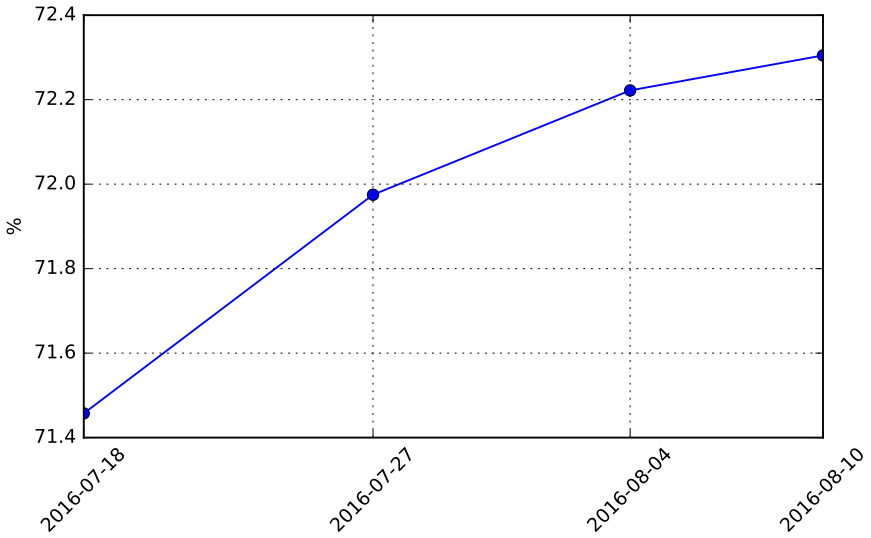

Figure 4.6: Proportion of ECN negotiation measured from the London vantage point.

Chapter 5

Conclusion

In this thesis the basis of a Path Transparency Observatory has been designed and implemented. At the time of writing, the software is up and running on MAMI infrastructure[1]. The observatory is able to analyze large datasets efficiently by providing analyzer modules access to cluster computing technology, MongoDB (Section 2.1) and Apache Hadoop & Spark (Section 2.2). The performance gain by using cluster computing in the observatory could not be assessed because the server cluster was not yet ready.

The process of detecting changes and updating the observations upon the addition of new data is done efficiently by processing a subset of data that has been determined by the analyzer module based solely on measurement time spans (Section 3.2). By policy, raw data is never deleted from the observatory but can be marked invalid if it is flawed (Section 3.2.1). The information is automatically propagated to all observations that are directly or indirectly depending on the invalidated raw data. Every observation is accompanied by information how it was generated (analyzer module version) and which raw data has been involved in this process (Action log in section 3.2.1 and field `sources` of an observation in section 3.1). In addition, observations are versioned, which allows to query the observatory for a specific version of observations using the query engine (Section 3.4).

A few thoughts about the chosen technologies MongoDB, Apache Hadoop and Apache Spark (Chapter 2): MongoDB was a good choice because the document model aligns with the nature of the observatory data model (nested documents, arrays) and the aggregation pipeline is very powerful. In my opinion, the query language is a bit odd (e.g. matching a single value has the same query shape as an exact match on an array, and matching an array element[1]) and it is missing some important features (e.g. match documents where array size $>$ a constant integer, getting the last element of an array, or hashed indexes over arrays (Workaround in section 3.2.3)).

Apache Hadoop and Apache Spark work well together but introduce a massive overhead when used from Python, they are very inefficient when storing small files and only support Kerberos as authentification method. For the small files problem, there exists a workaround with using SequenceFiles, but it makes the code unnecessary complex. A lightweight and pythonic alternative for Spark would be the Dask+Distributed framework[2].

The proof of concept was successful and has shown that the concepts for automated subset analysis, observation versioning and derived observations fit together in practice. The observatory is now ready to be loaded with analyzer modules and to be filled with raw data.

[1] https://docs.mongodb.com/manual/tutorial/query-documents/#query-on-arrays
[2] http://distributed.readthedocs.io/en/latest/

Chapter 6

Outlook

- A public website on top of this work is being implemented by the MAMI team.
- The array `action_ids` in the observation (Section 3.1) is not necessary, two action id fields would suffice: an `insert_action_id` and a `deprecation_action_id`. When an observation is added to the observations collection, the insert action id is set. When the observation is marked invalid, the deprecation action id is set. Now in the case that the same observation appears again, it is inserted as a new observation into the observations collection. This will result in a duplicate observation with different action ids. It is expected that the impact on storage requirements will be marginal, because MongoDB employs collection-level compression, and querying for specific version will be less complex and faster, because the costly `$unwind` and `$group` aggregation stages in the query engine can be eliminated (Appendix A.4).
- Currently when raw data is marked invalid and an analyzer module has generated direct observations from it, the analyzer module is executed again. In this case the analyzer module needs to exit without generating any observations. It would be better that the observations are marked invalid directly without needing to execute the analyzer module.
- Users should be able to order online analyzer accounts (3.3.4) through the admin interface. The admin interface needs to forward this request to the supervisor.
- The `sources` field of an observation (Section 3.1) could be extended to hold more detailed format-specific information such as the line number in the raw data file.
- The `AnalyzerContext` can be extended to allow analyzer modules to report their progress and also allow them to complain about raw data. These complaints can then be monitored for example by the validator and the validator can automatically mark raw data invalid.
- The problem that the user needs to make sure that no analyzer module is running when raw data is being marked (in)valid (Section 3.3.3) can be solved the following way: The admin interface sets a flag that prevents the supervisor to start any more analyzers. The validator waits until there is no analyzer module running anymore, performs the marking and resets the flag so that the supervisor is no longer blocked.

List of Figures

List of Tables

Bibliography

[1] S. Neuhaus, R. Müntener, K. Edeline, B. Donnet, E. Gubser: Towards an Observatory for Network Transparency Research
Proceedings of the 2016 ACM, IRTF & ISOC Applied Networking Research Workshop, Berlin (July 2016)

[2] R. Müntener: Observatory Upload
https://github.com/mami-project/observatory-upload
(Retrieved 03.08.2016, commit 7bfe215759)

[3] Project Jupyter: JupyterHub 0.6.1 Documentation
http://jupyterhub.readthedocs.io/en/stable/
(Retrieved 02.08.2016)

[4] PATHspider: I. Learmonth, B. Trammell, M. Kühlewind, G. Fairhurst: A tool for active measurement of path transparency
Proceedings of the 2016 ACM, IRTF & ISOC Applied Networking Research Workshop, Berlin (July 2016)

[5] MongoDB, Inc: The MongoDB™3.2 Manual
https://docs.mongodb.com/v3.2/
(Retrieved 02.08.2016)

[6] MongoDB, Inc: Hashed Indexes - The MongoDB™3.2 Manual
https://docs.mongodb.com/v3.2/core/index-hashed/#hashing-function
(Retrieved 02.08.2016)

[7] Apache Spark™: Spark Programming Guide 2.0.0
http://spark.apache.org/docs/2.0.0/programming-guide.html
(Retrieved: 02.08.2016)

[8] Apache Hadoop™: Hadoop User Guide 2.7.2
http://hadoop.apache.org/docs/r2.7.2/
(Retrieved: 02.08.2016)

[9] T. Kindberg, S. Hawke: The 'tag' URI Scheme, RFC 4151, IETF (October 2005)

[10] K. Ramakrishnan, S. Floyd, D. Black: The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, IETF (September 2001)

[11] S. Chacon, B. Straub: Pro Git (Second Edition), Apress (2014)

[12] K. Reitz and others: Requests HTTP for Humans
http://docs.python-requests.org/en/master/
(Retrieved: 09.08.2016)

[13] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, R. Scheffenegger: Enabling Internet-Wide Deployment of Explicit Congestion Notification
Proceedings of the 2015 Passive and Active Measurement Conference, New York (March 2015)

[14] E. Gubser: Measuring Explicit Congestion Negotiation (ECN) support based on P2P networks (May 2015)
ftp://ftp.tik.ee.ethz.ch/pub/students/2015-FS/SA-2015-05.pdf

[15] B. Trammell: Quality of Flow: IPFIX flow meter based on YAF, focused on passive TCP performance measurement, September 2014
https://github.com/britram/qof/tree/develop
(Retrieved: 02.08.2016, commit 32699377c5)

Appendix A

Example Code

A.1 AnalyzerContext Example

```python
from ptocore.analyzercontext import AnalyzerContext
from ptocore.sensitivity import extend, extend_hourly

# establish connection to supervisor
ac = AnalyzerContext()

# use extend method to expand to hourly time spans
max_action_id, timespans = extend(extend_hourly, ac.action_set)

# declare suitable subset
ac.set_result_info(max_action_id, timespans)

# get uploads within suitable subset
uploads = ac.spark_uploads()

# write a function to extract interesting values from raw data
def prepare(upload):
  ...

# write a function that takes values and generates observations
def analyze(values):
  ...

# execute on cluster with spark
analyze(uploads.flatMap(prepare)).saveToMongoDB(ac.temporary_uri)
```

Listing A.1: Mockup of analyzer module code. Usage example of `AnalyzerContext`.

A.2 ptocore.json

All analyzer module need to have a file called `ptocore.json` which includes all necessary information about the analyzer module.

```
{
  "input_formats": ["ecnspider1-zip-csv-ipfix"],
  "input_types": [],
  "output_types": ["ecn.connectivity.works",
   "ecn.connectivity.broken",
   "ecn.connectivity.transient",
   "ecn.connectivity.offline",
   "ecn.negotiated",
   "ecn.not_negotiated"],
  "command_line": ["python3", "master.py"],
  "direct": true
}
```

Listing A.2: ptocore.json for analyzer-ecnspider1 (Section 4.2).

```
{
  "input_formats": [],
  "input_types": [
```

```
    "ecn.connectivity.works",
    "ecn.connectivity.broken",
    "ecn.connectivity.transient",
    "ecn.connectivity.offline",
    "ecn.negotiated",
    "ecn.not_negotiated"],
  "output_types": [
    "ecn.path_dependent",
    "ecn.site_dependent"
  ],
  "command_line": ["python3", "master.py"],
  "direct": false
}
```

Listing A.3: ptocore.json for analyzer-ecnspider-vp (Section 4.2).

A.3 Admin Interface Client

```
import requests

# depends on your flask configuration variables
base_uri = 'http://localhost:33425'

def create_analyzer(analyzer_id, conf):
    ans = requests.post(base_uri + '/analyzer/'+analyzer_id+'/create', json=conf)
    print(str(ans))
    return ans

def update_analyzer(analyzer_id, conf):
    ans = requests.post(base_uri + '/analyzer/'+analyzer_id+'/setrepo', json=conf)
    print(str(ans))
    return ans

def an_list():
    return requests.get(base_uri + '/analyzer').json()

def an_enable(analyzer_id):
    return requests.put(base_uri + '/analyzer/'+analyzer_id+'/enable')

def an_disable(analyzer_id):
    return requests.put(base_uri + '/analyzer/'+analyzer_id+'/disable')

def an_cancel(analyzer_id):
    return requests.put(base_uri + '/analyzer/'+analyzer_id+'/cancel')

def ul_valid(upload_id:str):
    return requests.put(base_uri + '/upload/'+upload_id+'/valid')

def ul_invalid(upload_id:str):
    return requests.put(base_uri + '/upload/'+upload_id+'/invalid')

# example: register analyzer-ecnspider1 from github
ans = create_analyzer('analyzer-ecnspider1', {
  "repo_url": "https://github.com/gubser/analyzer-ecnspider1.git",
  "repo_commit": "7d15a2169a4672724c1569608f74a09baf774bf7",
})
print(ans.text)

# example: update analyzer-ecnspider-vp to another repository and/or version
ans = update_analyzer('analyzer-ecnspider-vp', {
  "repo_url": "https://github.com/gubser/analyzer-ecnspider-vp.git",
  "repo_commit": "6678ba09aa86e9e5df869d92cfcdc8b64aacb7a1",
})
print(ans.text)

# example: list all analyzers
print(an_list())

# example: enable analyzer-ecnspider1
ans = an_enable('analyzer-ecnspider1')
print(ans.text)

# example: disable upload
ans = ul_invalid('5767a53731e34a6c3925a72b')
print(ans.text)
```

Listing A.4: Python functions for interfacing with the observatory using the `requests` library.

A.4 Query Engine Aggregation Pipeline

```
# maximum action id to consider (= version of the observatory to query for)
max_action_id = 112

# pipeline stages
pipeline = [
    # unwind action_ids array
    {'$unwind': '$action_ids'},

    # keep all that exist before max_action_id
    {'$match': {'action_ids.id': {'$lte': max_action_id}}},

    # take the first document = most recent action at that time
    {'$group': {
        '_id': '$_id',
        'action_ids_x': {'$push': '$action_ids'},
        'obs': {'$first': '$$CURRENT'}
    }},

    # restore the shape of an observation
    {'$project': {
        'action_ids': '$action_ids_x',
        'path': '$obs.path',
        'time.from': '$obs.time.from',
        'time.to': '$obs.time.to',
        'conditions': '$obs.conditions',
        'value': '$obs.value',
        'analyzer_id': '$obs.analyzer_id',
        'sources': '$obs.sources'
    }},

    # only consider valid observations
    {'$match': {'action_ids.0.valid': True}}

    # insert your query {'$match': <query>} or additional pipeline stages here:
    ...
]

# execute pipeline
cursor = observations_coll.aggregate(pipeline)
```

Listing A.5: Pipeline stages for implementing query for a specific version.

Appendix B

Declaration of Originality