mPlane
an Intelligent Measurement Plane for Future Network and Application Management
ICT FP7-318627

Final Implementation and Evaluation of the Data Processing and Storage Layer.

Author(s):
- POLITO: A. Finamore, A. S. Khatouni, S. Traverso, M. Mellia
- FUB: V. Guchev, F. Matera, E. Tego
- SSB: S. Pentassuglia
- EURECOM: M. Milanesio (ed.)
- ENST: Y. Gong, D. Rossi
- NEC: M. Dusi, M. Ahmed, K. Kutzkov, S. Nikitaki
- TID: I. Leontiadis, M. Varvello, L. Baltrunas
- NETvisor: Á. Bakay, G. Rózsa, B. Szabó
- FTW: P. Casas, A. Bár, P. Fiadino, A. D’Alconzo
- FW: E. Kowallik

Document Number: D3.4
Revision: 1.0
Revision Date: 15 July 2015
Deliverable Type: RTD
Due Date of Delivery: 30 June 2015
Actual Date of Delivery: 15 July 2015
Nature of the Deliverable: (S)oftware
Dissemination Level: Public
Abstract:

This deliverable collects the software released by the mPlane Consortium at month 32 within Work Package 3. This is a software deliverable, so this document briefly describes the software by collecting the information that is present on the website page at the time of writing. Software and instruction on how to access it must be accessed from http://www.ict-mplane.eu/public/software, and in particular under the Repository Tools section.

Keywords: mPlane software, repository tools, job scheduler, algorithm design
Disclaimer

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

Disclaimer ................................................................................................................ 3
Document change record ............................................................................................ 6
Executive Summary ................................................................................................... 7

1 General description ............................................................................................... 8

2 Query Engines ...................................................................................................... 9
  2.1 DBStream ....................................................................................................... 10
    2.1.1 Developer .......................................................................................... 10
    2.1.2 Description ........................................................................................ 10
    2.1.3 Deployment requirements ................................................................. 12
    2.1.4 New features supported by the mPlane project ................................. 18
    2.1.5 References ......................................................................................... 18
  2.2 EZRepo ........................................................................................................... 19
    2.2.1 Developer .......................................................................................... 19
    2.2.2 Description ........................................................................................ 19
    2.2.3 Deployment requirements ................................................................. 25
    2.2.4 New features supported by the mPlane project ................................. 25
    2.2.5 References ......................................................................................... 26

3 Benchmarking ...................................................................................................... 27
  3.1 DBStream Benchmarking ............................................................................ 28
    3.1.1 Spark Introduction .............................................................................. 28
    3.1.2 Hive Introduction ................................................................................ 28
    3.1.3 System Setup and Datasets ................................................................. 28
    3.1.4 Job Definition .................................................................................... 29
    3.1.5 DBStream Benchmark Implementation ............................................ 30
    3.1.6 Improving DBStream Performance with Intelligent Scheduling ....... 30
    3.1.7 Performance Evaluation and System Comparison .......................... 32

4 Schedulers ........................................................................................................... 35
  4.1 Cache-Oblivious Scheduling of Shared Workloads -- Results .................. 36
    4.1.1 Experimental Setup ............................................................................ 37
    4.1.2 Comparison of Scheduling Algorithms ............................................. 38
4.1.3 Scalability Comparison ................................................................. 39
4.1.4 PostgreSQL Experiments .............................................................. 40

5 Web Page Reports.................................................................................. 45
  Blockmon Controller ................................................................................ 45
  DBStream .................................................................................................. 49
  EZRepo ...................................................................................................... 56
  Math ......................................................................................................... 59
  mPlane Interfaces for Tstat ................................................................. 61
  Mongo DB proxy ..................................................................................... 63
  repoSim ................................................................................................... 70
  HFSP ......................................................................................................... 73
  Schedule .................................................................................................. 75
  Spark jobs for data analytics ............................................................. 77
## Document change record

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Editor(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>5 Jul 2015</td>
<td>M. Milanesio (Eurecom)</td>
<td>initial draft</td>
</tr>
<tr>
<td>0.2</td>
<td>14 Jul 2015</td>
<td>M. Milanesio (Eurecom)</td>
<td>merged version</td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable summarizes the software that the Consortium made available at month 32 since the beginning of the project. The software that is part of this deliverable is listed below:

1. Query engines
   (a) Blockmon repository (NEC)
   (b) DBStream (FTW)
   (c) EZRepo (NETVISOR)
   (d) MATH: Mplane Authorized Transfer via HTTP (FTW)
   (e) mPlane Tstat exporter / importer (Polito)
   (f) MongoDB proxy (TID)
   (g) RepoSIM 2.0 (ENST)

2. Schedulers
   (a) HFSP (Eurecom)
   (b) Novel Job Scheduling algorithm (FTW)

3. Data analytics
   (a) Spark jobs for processing raw data (Eurecom)

For each tool, a web page has been prepared following (where appropriate) the same structure, and an archive is provided so to allow people to download and run the released software. Detailed information of what has been developed within mPlane is given.

Each tool page can be accessed from from http://www.ict-mplane.eu/public/software. The page lists all software that has been developed by mPlane partners, some of which are not part of this deliverable. The list above includes software that either was entirely developed, or that received significant contributions and updates within the project.
1 General description

This deliverable summarizes the software made available by the Consortium at month 32.

The deliverable is organized as follows. First, a detailed overview on two query engines is given: DBStream, developed by FTW, and EZRepo (Netvisor), including some benchmarking results. Next, experimental results and discussion on the cache-oblivious way to schedule shared data-intensive workloads, presented in D3.3, are given.

Then, we report the printed version of each tool description page, that can be found under http://www.ict-mplane.eu/public/software under the Repository tools section. It has to be intended as a snapshot since the page itself is supposed to change in the future. We will follow this order in the report.

1. Query engines
   (a) Blockmon repository (NEC)
   (b) DBStream (FTW)
   (c) EZRepo (NETVISOR)
   (d) MATH: Mplane Authorized Transfer via HTTP (FTW)
   (e) mPlane Tstat exporter / importer (Polito)
   (f) MongoDB proxy (TID)
   (g) RepoSIM 2.0 (ENST)

2. Schedulers
   (a) HFSP (Eurecom)
   (b) Novel Job Scheduling algorithm (FTW)

3. Data analytics
   (a) Spark jobs for processing raw data (Eurecom)

Each tool page can be accesses from from http://www.ict-mplane.eu/public/software.
2 Query Engines
2.1 DBStream

2.1.1 Developer

FTW

2.1.2 Description

DBStream is a flexible, scalable and easy to use Data Stream Warehouse (DSW) designed and implemented at FTW. The main purpose of DBStream is to store and analyze large amounts of network monitoring data. Indeed, DBStream is tailored to tackle the requirements of Network Traffic Monitoring and Analysis (NTMA) applications, both in terms of storage and near real time data processing and analysis. DBStream is a repository system capable of ingesting data streams coming from a wide variety of sources (e.g., passive network traffic data, active measurements, router logs and alerts, etc.) and performing complex continuous analysis, aggregation and filtering jobs on them. DBStream can store tens of terabytes of heterogeneous data, and allows both real-time queries on recent data as well as deep analysis of historical data.

DBStream is implemented as a middle-ware layer on top of PostgreSQL. Whereas all data processing is done in PostgreSQL, DBStream offers the ability to receive, store and process multiple data streams in parallel. As we have shown in a recently published benchmark study [2], DBStream is at least on par with recent large-scale data processing frameworks such as Hadoop and Spark.

One of the main assets of DBStream is the flexibility it provides to rapidly implement new NTMA applications, through the usage of a novel stream processing language tailored to continuous network analytics. Called CEL (Continuous Execution Language), this declarative, SQL-based language is highly precise yet very easy to use. Using CEL, advanced analytics can be programmed to run in parallel and continuously over time, using just a few lines of code.

The near real time data analysis is performed through the online processing of time-length configurable batches of data (e.g., batches of one minute of passive traffic measurements), which are then combined with historical collections to keep a persistent collection of the output. Moreover, the processed data can then be easily integrated into visualization tools (e.g., web portals).

In DBStream, base tables store the raw data imported into the system, and materialized views (or views for short) store the results of queries such as aggregates and other analytics --- which may then be accessed by ad hoc queries and applications in the same way as base tables. Base tables and materialized views are stored in a time-partitioned format inside the PostgreSQL database, which we refer to as Continuous Tables (CT). Time partitioning makes it possible to insert new data without modifying the entire table; instead, only the newest partition is modified, leading to a significant performance increase.

A job defines how data are processed in DBStream, having one or more CTs as input, a single CT as output and an SQL query defining the processing task. An example job could be: `count the distinct
Figure 2.1: General overview of the DBStream architecture. DBStream combines on-the-fly data processing of DSMSs (Data Stream Management Systems) with the storage and analytic capabilities of DBMSs (DataBase Management System) and big data analysis systems such as Spark.

destination IPs in the last 10 minutes”. This job would be executed whenever 10 new minutes of data have been added to the input table (independently of the wall clock time) and stored in the corresponding CT.

Figure 2.1 gives a high-level overview of the DBStream architecture. DBStream consists of a set of modules running as separate operating system processes. The Scheduler defines the order in which jobs are executed, and besides avoiding resource contention, it ensures that data batches are processed in chronological order for any given table or view. Import modules may pre-process the raw data if necessary, and signal the availability of new data to the Scheduler. The scheduler then runs jobs that update the base tables with newly arrived data and create indices, followed by incrementally updating the materialized views. Each view update is done by running an SQL query that retrieves the previous state of the view and modifies it to account for newly arrived data; new results are then inserted into a new partition of the view, and indices are created for this partition. View Generation modules register jobs at the Scheduler.

Finally, the Retention module is responsible for implementing data retention policies. It monitors base tables and views, deleting old data based on predefined storage size quotas and other data retention policies. Since each base table and view is partitioned by time, deleting old data is simple: it suffices to drop the oldest partition(s).

The DBStream system is operated by an application server process called hydra, which reads the DBStream configuration file, starts all modules, and monitors them over time. Status information is fetched from those modules and made available in a centralized location. Modules can be placed on separate machines, and external programs can connect directly to DBStream modules by issuing
simple HTTP requests.

2.1.3 Deployment requirements

DBStream and the used libraries assume that you are using golang version 1.2.x ([https://golang.org/](https://golang.org/)). Therefore, for older versions of Ubuntu like e.g. 12.04 you might follow the instructions in this guide: [http://www.tuomotanskanen.fi/installing-go-1-2-on-ubuntu-12-04-lts/](http://www.tuomotanskanen.fi/installing-go-1-2-on-ubuntu-12-04-lts/). Next we provide a step-by-step description on how to install and run DBStream, as well as how DBStream is integrated in mPlane.

2.1.3.1 Installing DBStream

DBStream source code uses the go language; to compile the go source code of DBStream you have to install the go language:

```
apt-get install golang
```

DBStream also uses several open source libraries which you have to install in order to compile DBStream. First you need to create a directory where go code can be downloaded to, e.g:

```
mkdir ~/go
```

Next you need to export a new environment variable so go knows where to put the code, which at least in bash works like this:

```
export GOPATH=~/go
```

Now you can install the needed libraries with the following command:

```
go get github.com/lxn/go-pgsql
go get github.com/go-martini/martini
go get code.google.com/p/vitess/go/cgzip
```

Now go to the DBStream server directory e.g:

```
cd ~/source/dbstream/
```

and run the build script there:

```
./build.sh
```

The resulting executables will be placed in the `bin/` directory. The main executable is called hydra which starts the application server.
cd bin/
./hydra --config ../config/serverConfig.xml

Edit the server configuration and add the modules of DBStream you want to use. If you want to get some information about the application server you can run the command remote to monitor and control the server. This command shows the current status of the application server every second:

watch -n 1 ./remote

The default config also starts a CopyFile module. You can see that currently no files are being imported by checking:

http://localhost:3000/DBSImport

The next step before actually running DBStream is to set up Postgres as a DBStream backend. The first step is to install PostgreSQL. We where using versions from up to 8.4 for DBStream, but rather recommend to use newer versions, like e.g. 9.3. On ubuntu you can install PostgreSQL with the following command:

apt-get install postgresql-9.3

Then you have to create an operating system and database user for DBStream. From now on, we will assume that this user is called dbs_test but you can choose any other user name, just make sure that all parts of the configuration are adapted as well. This user has to be a postgres superuser.

sudo useradd -s /bin/bash -m dbs_test

Now you have to create a database with the name of that user; note that this database will also be used to store all data imported to and processed with DBStream.

sudo su - postgres
createuser -P -s dbs_test
createdb dbs_test
exit

DBStream uses a tablespaces to store data on disk, namely dbs_ts0. For testing purposes, we will locate them in the home folder of the dbs_test user, but in a real setup you probably want to set them to a large RAID-10 storage array.

sudo mkdir /home/dbs_test/dbs_ts0
sudo chown postgres /home/dbs_test/dbs_ts0

Now the newly created DBStream database needs to be initialized. Therefore, change to the test directory and login into the database you just created:
cd test
psql dbs_test  # note that you need to login with a database
superuser, so you might want to change to the dbs_test user first.

If you log correctly into the database you should see something like this:

```
psql (9.3.6)
Type "help" for help.
```

dbs_test=#

Now run the following command to initialize some DBStream internal tables.

```
\i initialize.sql
```

If all steps from this part were successfully completed you can go on and start DBStream for the first time.

### 2.1.3.2 Installing DBStream from Vagrant

Within mPlane, and to ease the installation of a DBStream instance without the burden of installing and configuring all its components, we provide a DBStream-Vagrant based image at [https://github.com/arbaer/dbstream/tree/master/vagrant](https://github.com/arbaer/dbstream/tree/master/vagrant). Vagrant is a tool for building complete development environments, aiming at lowering development environment setup time.

To run a DBStream instance using Vagrant, you need to follow these steps:

1. Install vagrant version 1.7.2 from [http://www.vagrantup.com/downloads.html](http://www.vagrantup.com/downloads.html)
2. Add the ubuntu/trusty64 box:

   ```
vagrant box add ubuntu/trusty64
   ```
3. Start the virtual machine:

   ```
vagrant up
   ```
4. Connect to the virtual machine:

   ```
vagrant ssh
   ```

In case you want to recompile DBStream set the GOPATH environment variable and execute the build.sh single threaded:

```
export GOPATH=~/go
./build.sh single
```
2.1.3.3 Running DBStream

Now that DBStream is already installed, follow these steps to start it:

First we need to change to the test directory.

```bash
cd test  # if you are comming here from vagrant, the directory is src/dbstream/test
```

Now you should see the executables in this directory (e.g. `hydra`, `math_probe`, `math_repo`, `scheduler` and `remote`). For this example it is the best to open three shells. In the first shell we will run `dbstream`, in the second we will run the `import source` and the third will be used for monitoring DBStream.

In the monitoring shell run the following command:

```bash
cd dbstream/test
watch -n 1 ./remote
```

In the `dbstream` shell execute the following command:

```bash
cd dbstream/test
./hydra --config sc_tstat.xml
```

In the `import source` shell run the following command:

```bash
cd dbstream/test
./math_probe --config math_probe.xml --repoUrl "localhost:3000" --startTime 2006-01-02T15:04:05
```

If all went well, you should now be able to log into postgres, and check some preloaded tables:

```bash
psql dbstream
select * from example_log_tcp_complete;
```

To cleanup the tables and run the example import again, inside postgres execute the following command:

```bash
select dbs_drop_table('example_log_tcp_complete');
slect dbs_drop_table('tstat_test');
```

and in the shell run:

```bash
rm -rf /tmp/target/
```
2.1.3.4 mPlane integration

In mPlane, DBStream is integrated and used together with the Tstat probe, storing and analyzing the data captured and exported by the probe. As described in Figure 2.2, the integration includes a mPlane proxy to the repository (RepoProxy) and a data transfer protocol which enables the Tstat probe to send bulk measurements to DBstream, and DBStream to import these measurements into tables for further analysis by the Analysis Modules it runs (e.g., anomaly detection). Data transfer is achieved through a customized protocol we have named MATH - Mplane Authorized Transfer via HTTP.

MATH is composed of two modules, math_probe and math_repo, the former runs together with the Tstat probe and it handles the transfer of Tstat logs to a mPlane repository, the latter runs together with DBStream and handles the importing of the received Tstat logs into the DBStream database. Both MATH modules come with .xml configuration files which extends the flexibility of the MATH protocol to be used with other probes and repositories.

To run the mPlane integration of DBStream with Tstat, you have to follow the next steps:

1. Download and install the Tstat and DBStream tools, both available at GitHub under https://github.com/fp7mplane/components/tree/master/tstat and https://github.com/arbaer/dbstream

2. Install the mPlane framework and probe and repository mPlane proxies

   
git clone https://github.com/fp7mplane/protocol-ri.git

   Enter the protocol-ri/mplane folder and rename (or remove) components. Then, check out the one available on github.
cd protocol-ri/mplane/
mv components components.orig (or rm -rf components)
git clone https://github.com/fp7mplane/components/

Add the following required capabilities at the Supervisor configuration files conf/supervisor.conf:

tstat-log_http_complete = guest,admin
tstat-exporter_log = guest,admin
repository-collect_log = guest,admin

3. Run the mPlane Supervisor:

   ./scripts/mpsup --config ./conf/supervisor.conf

4. Run the Tstat proxy:

   ./scripts/mpcom --config ./mplane/components/tstat/conf/tstat.conf

5. Run the Repository proxy:

   ./scripts/mpcom --config ./mplane/components/tstat/conf/tstatrepository.conf

6. Run the mPlane Client:

   ./scripts/mpcli --config ./conf/client.conf

7. Run both DBStream and the MATH importer module, math_repo:

   ./hydra --config sc_tstat.xml
   ./math_repo

8. Run Tstat and the MATH exporter module, math_probe, using the mPlane Client shell:

   |mplane| runcap tstat-log_tcp_complete-core
   |when| = now + inf

   |mplane| runcap tstat-exporter_log
   repository.url = localhost:3000

At this point in time, the Tstat proxy sends log files collected by Tstat to the repository proxy, and the log files are then stored in DBStream, where different analysis modules perform further analysis.
2.1.4 New features supported by the mPlane project

Thanks to the support of the mPlane project we extended DBStream functionalities with the following features:

- we extended the functionality of the CEL language, making it much easier to code an analysis job on top of DBStream.
- we added MATH (Mplane Authorized Transfer via HTTP), a protocol to export bulk data in the form of logs from a mPlane probe (e.g., Tstat) and to import it into DBStream.
- we added Machine Learning analysis capabilities to DBStream, by integrating a well know Machine Learning toolbox (WEKA) directly into the data processing of DBStream jobs.
- we improved the performance of DBStream in terms of complete job completion time by studying and developing different job scheduling approaches.

2.1.5 References

**Links to sources, binaries:** all DBStream and DBStream-related (e.g., MATH) sources and binaries are accessible on GitHub at https://github.com/arbaer/dbstream.

**Links to additional documentation:** additional documentation on DBStream can be found at https://github.com/arbaer/dbstream/blob/master/README.md

**Dissemination:** DBStream has been presented in 2014 in the following conferences [1, 2]: Wireless Communications and Mobile Computing Conference (IWCMC), and IEEE International Conference on Big Data (IEEE BigData).
2.2 EZRepo

2.2.1 Developer

NETVISOR, with contribution from FHA and FW

2.2.2 Description

2.2.2.1 Use case scenario

This Repository was developed in concordance with the corresponding Reasoner for the "WP5 Multimedia content" Use Case. In this scenario, the Reasoner is working in a troubleshooting/diagnostic role. The process is triggered by measured quality impairments, and the output of the diagnostics is the estimation of one or more causes responsible for the impairment. The Reasoner outlined here is used by the network operator (ISP) providing access for clients requesting multimedia (streaming) content. The operator is primarily concerned about if the bottleneck or problem is within the operator’s network or not.

1. If problem is out of his responsibility, the operator may notify and assist the content provider, and inform concerned customers.

2. If problem is within, then he should pinpoint the root cause of the problem - in this case, the precision of the diagnostics depends on the auxiliary information and measurements that is provided to augment content-delivery specific measurements. Network operators usually have diagnostics tools and procedures already in place to deal with internal network problems. Thus, in many cases the mPlane Reasoner just triggers this process by proving that a capacity or other problem is present in the network, and points at a location or component for starting the investigation.

In the following sections, we describe a basic setup that does not leverage on existing auxiliary measurements, carefully noting the points where the precision and usefulness (for the operator) of the reasoning process can be enhanced by providing additional measurements and data. Such data is very likely to be available, in a form specific to the operator’s existing monitoring systems.

Probes, measurements, pre-processing

The scenario uses measurements from both active and passive probes, with the following roles:

Active content probes simulating end-user transactions scheduled by the Reasoner. They are deployed at different locations within the ISP’s network, trying to achieve a good coverage of network paths between subscribers and the content provider. The probes are configured to request and download multimedia content (e.g. YouTube or OTT videos) and evaluate network, QoS (and partially QoE) characteristics. Their "playlists" are synchronized, so all probes are requesting the same content at the same time. Metrics include DNS resolution and server delay, best achievable bitrate, number of video stalls etc. Importantly, these probes also provide the names and IP addresses of content servers providing the requested media clips. Some of such probes are configured into
``crawling'' mode to query for different media clips in order to expand the list of known content servers.

Active "line" probes are used by the operator for performance testing of internal links (e.g. by transferring static HTTP content from known internal servers, or performing other simple measurements such as ping to evaluate line performance). These probes operate against known responders (or reference servers), also operated by the provider to help identify internal bottlenecks. The concept is to facilitate testing the performance of internal links or paths, and provide auxiliary measurements for evaluating the stream probes’ results. Obviously, if the provider has other means to obtain equivalent data (e.g. interface counters monitoring on routers), or a link utilization monitoring system, those measurements are also taken into account by the reasoning process in a similar fashion.

Passive tapping probes (e.g. Tstat, DATI) are deployed to central locations where they observe as much subscriber - content provider traffic as possible, and measure basic TCP transmission properties for each connection (e.g. average bitrate, volume transmitted, delay, retransmits etc.)

Using the server addresses obtained from active measurements, mPlane builds a list of known content servers, which is subsequently used to filter the passive probes’ logs for transactions. It must be noted that certain content providers (e.g. Google/YouTube) do not use dedicated servers for multimedia content, and this traffic is interspersed with other Web or Cloud traffic. Nevertheless, filtering these transactions by server IP is useful to detect overload issues caused by e.g. sudden spikes in demand.

It must also be noted that this scenario assumes that apart from IP addresses, port numbers and number of bytes transferred, no deeper information can be extracted by the passive probes. This models the case when the content is transferred over HTTPS or the passive probes’ DPI modules.
do not support the streaming protocol used.

Measurement readings are stored into Repositories, that perform the following functions:

- All measurement results are stored historically (along with metadata such as probe location and the media ID queried) for baselining

- Using the filtered list from passive logs, the Reasoner computes aggregated statistics characterizing traffic between the servers and subscribers served by the ISP. Such statistics include number of queries, approximates of total bandwidth served and min/max/avg server delay etc. for given intervals of time, and the computation is off-loaded to the repositories as much as possible.

In cases when subscriber addresses can be tied to geographical regions or network locations, (e.g. in the case of regional DHCP servers or address pools), the filtering and aggregation can be further refined by client (subscriber) address data, and a subsequent filtering pass can be made over the transfer logs provided by the passive probes.

2.2.2.2 Operation of the Repository

EZRepo architecture

EZRepo will collect measurement records from different probes (e.g. OTT probe, GLIMPSE, etc.), with different measurement scopes and capabilities.

EZRepo has 3 main modules, the Data Store, the Query Engine and the Classifier. Probes send their data into the EZRepo with UDP protocol. (Other protocols like HTTP and SCP are under consideration/development).

The Classifier module will make the evaluation ("grading") of the input data, based on the threshold data stored in a JSON file (Grading.json). We propose a 5 grades scale which complies to the common ("normal-warning-minor-major-critical") qualification and notification levels and conforms to the ITU standards as well. Thresholds can be published through the 'SetGrading' capability to the Supervisor.

The classified data, along with the original measurements are stored in the Data Store.

The Query Engine receives the specifications and returns the results via the Supervisor, using 'Query-ByCriteria' specification and result parameters.

In the example figure given, the information collected by the GLIMPSE probes are queried from EZRepo. The specification defines a query from all UDP protocol based measurements done via GLIMPSE, where destination IP address is 1.2.3.4, and the source IP and network paths can be arbitrary. We are looking for all the records within this set, where the bandwidth grade is in the range 3..5. The result values will show us that 11 records has been found with the given selection criteria, from which 3 was graded in the given quality range.

Periods and grading

The operation of the repository builds on the concept of Periods. Periods are tuples of attributes
describing the availability and access quality of some service (e.g. network connectivity or availability of some piece of content) during a certain period. Events are created based on some kind of qualification - i.e. "grading" - of some Probe measurements, along various criteria.

As an example an OTT probe measuring a piece of content may experience "EXCELLENT" quality for a long time, then the quality may go down to "NOT_AVAILBLE", possibly going through a "POOR" period before that. (We recommend simple grading with low number of options, like this 3-step classification.)

The structure of each Period is as follows:

- **MeasType**: the type of the probe/measurement represented by the Period
- **Begin**: start time specification for the Period
- **End**: end time specification for the Period
- **ProbeID**: the probe producing the measurement
- **ClientID**: the initiator of this measurement Period. For active probes, ClientID and ProbeID are equal.
- **ServerID**: the server or responder terminating the measurement
- **ContentID**: which program/content has been accessed
- **NodePath**: a list of relevant topological nodes on the path between the client and the server. It is not necessary to enumerate all routersswitches on the signal path individually, rather than the devices that somehow distribute or join the path from/to various services and vantage points. As an example, the typical access of a subscriber of provider 'A' to some video content CDN server of provider 'B' could span the following signal path: 1. The subscriber's home network, 2. The access node (DSLAM, OLT, CMTS) of Prov_A that serves the subscriber, 3.
OverallGrade: the overall quality classification for the Period. EXCELLENT | POOR | NOTAVAILABLE

BandwidthGrade: the classification of the bandwidth experienced through this Period, i.e. whether any slow downloads or slow IGMP responses were recorded. EXCELLENT | FAIR | POOR | NOTAVAILABLE

Periods are created from probe measurements by aggregating subsequent measurements into longer periods until any of classification criteria produces a different "grade". The example above shows two grade properties (for bandwidth and for overall quality), but we expect that periods will generally have not too many, i.e. up to 5 grade properties per period. This will avoid excessive segmentation of measurements into small periods.

It is to be emphasized that a Period only covers samples from repeated samples of a certain measurement (specification) on a certain probe.

The low number of grade properties and a reasonable classification system (offering a few grade values only for each property), should result in relatively few periods, i.e. a large number of measurements will be represented in a very concise way through Periods.

Criteria for Period selection

The concept of Criteria used in the searches has some key features that make them suitable for being used in diagnosis rules later in the Reasoner. In short, we define practical criteria grammar for each property type to cover all typical query scenarios.

- **Begin and End:** to compare a Period with a certain point in time T. The available comparison primitives are:
  - BB(<T>) [begins before]
  - BA(<T>) [begins after]
  - EB(<T>) [ends before]
  - EA(<T>) [ends after].

  T can be here an absolute time or (more frequently) a time relative to "Now" or "Today", like "Now - 20 mins". An example of a full time criteria for the latest 5-min period:

  BB(Now-5min) AND EA(Now)

- **ServerID, ProbeID, ClientID, ContentID, MeasType:** the supported filtering primitives are EQ(<id>) (for strict equality), MATCH(<regexp>) for partial textual matching (on the ID), and MEMBEROF(<set>) for matching based on (predefined) sets. For network nodes only, an additional primitive NETMATCH(<netaddr/netmask>) is also available for network address based matching.

- **Grades** can be compared for service level equality with the usual '<=' '<' '!=' '>' '>' comparison operators, which make use of the grades' metric nature, i.e. that they always represent a "ladder of qualities".
• **NodePath** matching primitive is `PASS(<node>)`, which allows to filter traffic passing any node. Using compound expression like `PASS(nodeA) AND !PASS(nodeB)`, we can filter all traffic running through nodeA, but branching before nodeB.

It is to be noted that criteria can be *parametrized*, i.e. a criteria can be applied for searches on traffic with serverA and serverB, through the selection of parameters. An example of "complete" real-world criteria could be

MeasType EQ("OTT") AND BandwidthGrade <= "POOR" AND PASS(\$node1) AND PASS(\$node2) AND BB(Now - 10m) and EA(Now)

which selects Periods for OTT measurements with poor or worse quality passing certain nodes (defined through parameters) fully covering the last 10 minutes.

**Capabilities**

```json
{
  "registry-format": "mplane-0",
  "registry-revision": 0,
  "includes": ["http://ict-mplane.eu/registry/core"],
  "registry-uri": "http://91.227.139.40/ezrepo-registry.json",
  "elements": [
    {
      "name": "type.repo",
      "prim": "string"
    },
    {
      "name": "overall.grade",
      "prim": "natural"
    },
    {
      "name": "range.grade",
      "prim": "string"
    },
    {
      "name": "select.grade",
      "prim": "string"
    },
    {
      "name": "results.repo",
      "prim": "string"
    },
    {
      "name": "metric.repo",
      "prim": "string"
    }
  ]
}```
2.2.3 Deployment requirements

EZRepo has been implemented in Python3 as an mPlane compliant component. All it needs for running is the mPlane RI framework.

2.2.3.1 Quick start

Installation

- Download EZRepo package from github: https://github.com/fp7mplane/components/tree/master/EZRepo
- Copy to /protocol-ri/components
- Change component.conf and the other config files accordingly (certificates, authorization and roles settings, etc)
- Register through the component module:

  ```
  $ scripts/mpsup --config <supervisor\_config>
  $ scripts/mpcom --config <component\_config>
  ```

- Access it through the clientshell:

  ```
  $ scripts/mpcli --config <client\_config>
  ```

Usage

```
$ runcap repo
|when| = 2009-02-20 13:02:15 ... 2016-04-04 04:27:19
select.grade = 2..4
metric.repo = all
type.repo = ping
```

This example will return all "ping" measurements within the given timeframe where the grade is between 4 and 1.

2.2.4 New features supported by the mPlane project

EZRepo has been developed entirely within the mPlane project.
2.2.5 References

Links to sources, binaries  https://github.com/fp7mplane/components/tree/master/EZRepo
3 Benchmarking
3.1 DBStream Benchmarking

Given the growing number of big data analysis platforms being proposed and used in the context of Network Traffic Monitoring and Analysis (NTMA), we conducted a performance benchmarking study of DBStream against the state-of-the-art Big Data processing frameworks. In particular, we considered Spark and the Hadoop-based data warehouse Hive. The analysis is performed using network traffic flows captured in an operational network. The benchmark considers different types of standard analysis jobs used in NTMA applications such as flow counts per Autonomous System, user counts, Round-Trip Time (RTT) statistics, etc., as well as more evolved and complex jobs considering incremental traffic analysis. The term incremental refers to the usage of measurements in the previous batches to compute statistics on the new batch of data.

3.1.1 Spark Introduction

Spark is an open-source MapReduce solution proposed by the UC Berkeley AMPLab. It exploits Resilient Distributed Datasets (RDDs), i.e., a distributed data abstraction which allows in-memory operations on large clusters in a fault-tolerant manner. This approach has been demonstrated to be particularly efficient, enabling both iterative and interactive applications in Scala, Java and Python. Moreover, an application does not strictly require the presence of a Hadoop cluster to take advantage of Spark. In fact, the system offers a resource manager and supports different data access mechanisms. However, it is most commonly used in combination with Hadoop and the HDFS. You can refer to [2] for a detailed description of the implementation of this benchmark in Spark and the reasons for selecting Spark without the Spark Streaming extension.

3.1.2 Hive Introduction

Hive is an open-source data warehousing solution developed at Facebook. It supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into map-reduce jobs executed on Hadoop. Hive performs batch processing and is one of the most popular Big Data SQL enabled framework available nowadays.

3.1.3 System Setup and Datasets

We installed Spark and Hive on a set of eleven machines of the following identical hardware: a 6 core XEON E5 2640, 32 GB of RAM and a 5 disks of 3 TB each. One of those eleven machines has been dedicated to DBStream, recombining 4 of the available disks in a RAID10. We use PostgreSQL in version 9.2.4 as the underlying DBMS. The remaining 10 machines compose a Hadoop cluster. The cluster runs CDH 4.6 with the MapReduce v1 Job Tracker enabled. On the cluster we also installed Spark v1.0.0 but we were only able to use the standalone resource manager.

All machines are located within the same rack connected through a 1Gb/s switch. The rack also contains a 40 TB NAS (Network-Attached Storage) device used to collect historical data. In particular, in the study we used four, 5 day-long datasets, each collected at a different Vantage Point (VP) in a real ISP network from the 3th till the 7th of February 2014. Each VP is instrumented with Tstat.
to produce per-flow text log files from monitoring the traffic of more than 20,000 ADSL households. For each TCP connection, Tstat reports more than 100 network statistics and generates a new log file each hour. Overall, each of the four datasets corresponds to approx. 160 GB of raw data, about 5 times the memory available on a single cluster node. In total, the four datasets sum up to approx. 650 GB, which is about twice as large as the total amount of memory available in the whole cluster.

3.1.4 Job Definition

Based on our experience in the design of network monitoring applications and benchmarks for large-scale data processing systems, we define a set of 7 jobs that are representative of the daily operations we perform on our production Hadoop cluster.

**Import** imports the data into the system from the NAS, where raw data is stored in files of one hour each.

**J1** for every 10 minutes i) map each destination IP address to its organization name through the Maxmind Orgname database (the Maxmind Orgname database provides a mapping between IPs and Organization names, see [www.maxmind.com](http://www.maxmind.com)) and ii) for each found organization, compute aggregated traffic statistics, i.e. min/max/avg RTT, number of distinct server IP addresses, total number of uploaded/downloaded bytes.

**J2** for every hour, i) compute the organization name-IP mapping as in J1, ii) collect all data having organization names related to the Akamai CDN, and iii) compute some statistics, i.e. min/max/average RTT, aggregated for the whole Akamai service.

**J3** for every hour, i) compute the organization name-IP mapping as in J1, and ii) select the top 10 organization names having the highest number of distinct IP addresses connecting to them.

**J4** for every hour, i) transform the destination IP address into a /24 subnet, and ii) select the top 10 /24 subnets having the highest number of flows.

**J5** for every minute, for each source IP address, compute the total number of uploaded/downloaded bytes and the number of flows.

**J6** for every minute, i) find the set of distinct destination IP addresses, and ii) use it to update the set of IP addresses that were active over the past 60 minutes.

**J7** for every minute, i) compute the total uploaded/downloaded bytes for each source IP address, and ii) compute the average over the past 60 minutes.

Overall, these jobs define network statistics related to CDN and other organizations (J1 to J4), statistics related to the monitored households (J5) and two incremental queries (J6 and J7) computing aggregated statistics over rolling sets of IP addresses.
3.1.5 DBStream Benchmark Implementation

All queries are implemented in DBStream CEL. The fact that the output of a job is stored on disk and can be used as input to another job is exploited to achieve increased processing performance. Figure 3.1 shows the resulting job dependencies, where the nodes represent the jobs and an arrow from e.g. job J1 to J2 means that the output of J1 is used as input to J2. The number next to each arrow indicates the size of the input window in minutes. For example, in order to compute the results of J6 we first gather the set of active IP addresses per minute in J6 prepare. Then, J6 uses J6 prepare and its own past output as input for the computation of the next output time window. This is indicated by the reflexive arrow starting from and going back into J6.

3.1.6 Improving DBStream Performance with Intelligent Scheduling

The main bottleneck of the DBStream system in this setup is the disk sub-system. Therefore, we tried to minimize the amount disk I/O by intelligent scheduling. Tasks are normally scheduled in FIFO order in DBStream. Since we set the number of parallel tasks to 64, FIFO effectively results in all tasks being executed as soon as the input data is ready. The effect of the FIFO scheduling is shown in Figure 3.2 (top), where each point in the plot corresponds to the execution of one window. The x-axis of this figure corresponds to the time after the start of the experiment at which a certain
task finished execution. The y-axis corresponds to the time when the data item was created by the vantage point, normalized to the start of the whole dataset.

Since some jobs process faster than others, in the FIFO case, the time distance between those jobs increases over the run of the experiment. The first step is the data import, which not only puts data into the disk, but also into the disk cache of the OS. As soon as the difference in progress of different jobs needing the same input gets too big, the data of the input drops out of the cache and has to be read from disk again. This increases the I/O overhead and, at the same time, decreases the overall system performance of DBStream.

Figure 3.2 (bottom) shows the execution of the same set of jobs using a "shared" scheduling strategy. In the "shared" case, a new hour is only imported if the difference in time between the imported hour $t_i$ and the hour $t_j$ for which all jobs have finished processing is smaller than $x$.

$$t_i - t_j < x$$  \hspace{1cm} (3.1)

In this case, data stays in the OS cache and fewer I/O operations are needed to complete the experiment. By setting $x = 1$, we are able to reduce the execution time of the 4 VPs by a factor of 45% from 808 minutes to 446 minutes.
3.1.7 Performance Evaluation and System Comparison

**DBStream vs. PostgreSQL:** In the first experiment, we show the efficiency of incrementally computing the set of active IP addresses as compared to re-computing the result from scratch for each individual minute. The results of this comparison are visualized in Figure 3.3. We use DBStream to compute the set of IPs active over a moving window (of e.g. 60 minutes) and compare its performance to a regular PostgreSQL implementation of the same job. We fix the primary window to be 1 minute long and vary the size of the secondary sliding window, evaluating three variants: finding all the active IP addresses within the past 10 minutes, 30 minutes and 60 minutes.

Intuitively, as the length of the sliding window referenced by the query increases, we expect the performance advantage of incremental processing to increase. For example, in case of a 60 minute window, computing a new partition of the view from scratch requires accessing 60 separate partitions of the base table. In contrast, incremental processing requires accessing only two partitions: the previous result and the new window of base data. The results of this experiment, shown in Figure 3.3, confirm the hypothesis. The provided numbers correspond to the execution time on 1 day of data. For a 10 minute time window, the incremental approach shows only a very slight advantage over the regular PostgreSQL approach. However, for 30 minute time windows, the incremental approach is noticeably faster, and for the 60 minute time window, the incremental approach is over three times faster than the regular approach.

**Hive Performance:** HiveQL offers primitives that can be used for incremental queries (e.g., \texttt{WINDOW}, \texttt{LEAD} and \texttt{LAG}) only from version 0.11\(^1\). Unfortunately, the production cluster has an older version of the Hive framework. The only way to implement J6 and J7 is by exploding each job in a set of queries, one for each window to process. This clearly result in a huge overhead (e.g., 14,400 queries for J6). Therefore, we limited our analysis of Hive to only the first 5 jobs of the benchmark.

For each VP, we imported the data in Hive leaving them in text format but creating a partition for

---

\(^1\)https://cwiki.apache.org/confluence/display/Hive/LanguageManual+WindowingAndAnalytics#LanguageManualWindowingAndAnalytics-WINDOWclause
Figure 3.4: Performance of Hive for Jobs J1 to J5.

Differently from DBStream, which progressively ingests data based on the computation speed, we decided to process whole datasets at once rather than in a file by file manner. This is motivated by the fact that previous benchmarks show not optimal performance for Hive.

Figure 3.4 shows the performance results for Hive. Since it was not possible to implement the jobs J6 and J7 within the language provided by Hive, we only implemented the jobs J1 to J5. We executed the jobs using 1, 2 or all 4 VPs, by first importing the whole data and then running all jobs in parallel over the whole imported data. The execution timings increase with the amount of data processed.

Spark Performance and DBStream: for the Spark implementation, we used a slightly different
approach. Here, hours are processed consecutively by the jobs. This made it possible to implement all jobs, also the incremental ones. As opposed to the DBStream implementation, all jobs start from processing the imported data. Although it is possible to take the output of a job as input to the next job, Spark does not offer a framework for specifying and resolving those dependencies.

Figure 3.5 shows the results we were able to achieve running Spark on our cluster of 10 machines. For the jobs J1 to J5 Spark offers great performance and the whole cluster is perfectly able to parallelize the processing, leading to very good results. However, jobs J6 and J7 are not processed very fast. This comes from two factors: one the one hand, especially J6 can not be parallelized very well, since data has to be synchronized and merged in one single location after each minute. On the other hand, distinct sets have to be computed, for which huge amounts of data have to be moved around in the reduce phase. Also for J7 the computation has to be synchronized for every minute, but here, the amount of data is smaller, since the output for every minute is only a single number.

Finally, Figure 3.6 compares the performance of Spark and DBStream in terms of makespan. In DBStream, the total execution time is measured from the start of the import of the first hour of data until all jobs finished processing the last hour of data. For Spark, all jobs were started at the same time in parallel. We report the total execution time of the job finishing last, which was J6 in this experiment. Since for Spark, the import is done before the jobs start processing the data, we also report the job processing time plus the time it takes to import the data separately.

For DBStream, the execution time increases nearly linearly with the number of VPs and therefore the amount of data to process. In contrast, for Spark the main bottleneck is the execution time of J6. The total execution time does not increase much with more VPs, since multiple instances of J6 run in parallel. Therefore, Spark is able to utilize its parallel nature better the more jobs are running, whereas DBStream shows better performance for incremental jobs. For the one VP case, Spark, running on a 10 node cluster, takes 2.6 times longer than DBStream, running on a single node of the same hardware, to finish importing and processing the data.
4 Schedulers
4.1 Cache-Oblivious Scheduling of Shared Workloads -- Results

In deliverable D3.3 we introduced a novel approach to schedule shared data-intensive workloads in a cache-oblivious way. In a nutshell, the proposed approach relies on a novel formulation of precedence constrained scheduling with the additional constraint that once a data item is in the cache, all tasks that require this data item should execute as soon as possible thereafter. The intuition behind this formulation is that the longer a data item remains in the cache, the more likely it is to be evicted regardless of the cache size.

Previously we gave an optimal ordering algorithm using A* search over the space of possible orderings, and proposed efficient and effective heuristics that obtain nearly-optimal results in much less time. The proposed techniques and algorithms were implemented as Schedule, a tool for cache-oblivious scheduling of shared workloads (check https://www.ict-mplane.eu/public/schedule); the code to run the algorithms presented in D3.3 can be found in Github at https://github.com/arbaer/schedule.

In this deliverable, we present experimental results on real-life data warehouse workloads as those processed by DBStream, as well as using well-known, publicly available benchmarks to validate the proposed solution. We recommend the interested reader to check all the details of the proposed solution in Section 3.2 of deliverable D3.3. Note that all the results presented next were obtained using the mPlane Schedule tool.

Results evaluate the effectiveness and efficiency of the algorithms presented in Section 3.2 of deliverable D3.3 and available in Schedule (A*, Baseline, Greedy and Heuristic), considering both the Total Maximum Bandwidth (TMB) and the Weighted TMB (WTMB) metrics (cf., Section 3.2, deliverable D3.3). We start with a description of our data sets and experimental environment (Sec. 4.1.1), followed by the results:

- In Sec. 4.1.2, we experiment with different precedence graphs as inputs, and we report the TMB/WTMB scores obtained by each algorithm as well as the time it took to generate the schedules. In general, we find that Heuristic obtains nearly-optimal schedules, but is slower than Greedy and Baseline (but still much faster than A*). Furthermore, both Greedy and Heuristic generate significantly better schedules than Baseline.

- In Sec. 4.1.3, we run the proposed algorithms (except A*) on very large random precedence graphs to see if they can efficiently compute schedules for complex workloads. We found that Heuristic does not scale as well as Baseline and Greedy, but can still handle large precedence graphs.

- In Sec. 4.1.4, we execute various workloads in PostgreSQL and show the real-world performance improvements due to our scheduling algorithms. Again, Heuristic and Greedy outperform Baseline.
4.1.1 Experimental Setup

We used a dual CPU Xeon E5-2630 machine, with 64 GB of RAM, and a 10-disk RAID10 storage subsystem. As a database we use PostgreSQL 9.2.4. We implemented all the algorithms in the Go language (http://golang.org).

The pgfincore (http://pgfoundry.org/projects/pgfincore/) library is used to advise the operating system to drop tables from the disk cache. We use this functionality to evict tables from the cache when they are no longer needed. We will explicitly state whenever we make use of this function.

We used the following groups of data sets. The number of nodes and edges in the corresponding precedence graphs is shown in Table 4.1, under the columns labeled |V| and |E|.

- **running** corresponds to the running example from Figure 4.1, described in D3.3.
- **test1, test2, test3** and **test5** correspond to small hand-crafted workloads with various features.
- **realworld1** and **realworld2** are two network monitoring workloads from data warehouses we are currently operating using DBStream. The tasks are to base table and materialized view updates.
- **tpc-ds-scan, tpc-ds-7q, tpc-ds-11q** and **tpc-ds-63q** are based on the TPC-DS decision support benchmark (http://www.tpc.org/tpcdis/). TPC-DS contains 24 base tables (7 fact tables and 17 dimension tables) and 99 predefined queries over the base tables. The number of tables required by a query ranges from one to 13, with an average of 4. We generated two versions of the benchmark: one with a scale factor of 10 and one with 100.

**tpc-ds-scan** is a workload of scan queries over the largest base tables: catalog_sales, web_sales and store_sales. These three tables account for 8.4GB in the scale-factor-10 version. The other three workloads contain 7, 11 and 63 queries from TPC-DS; the corresponding precedence graphs get progressively more complex. We do not use all 99 queries as not all of them are
Table 4.1: Performance comparison of the implemented algorithms. Times marked with * indicate that the experiment was stopped after 1 hour of wall clock time.

<table>
<thead>
<tr>
<th>Graph</th>
<th></th>
<th></th>
<th>Baseline</th>
<th>Greedy</th>
<th>Heuristic</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TMB</td>
<td>Mean</td>
<td>SD</td>
<td>TMB</td>
<td>Mean</td>
</tr>
<tr>
<td>running</td>
<td>6</td>
<td>6</td>
<td>0.00</td>
<td>0.010s</td>
<td>6</td>
<td>0.00</td>
</tr>
<tr>
<td>test1</td>
<td>11</td>
<td>10</td>
<td>16.16</td>
<td>0.64</td>
<td>16.16</td>
<td>1.32</td>
</tr>
<tr>
<td>test3</td>
<td>25</td>
<td>30</td>
<td>91.50</td>
<td>12.96</td>
<td>66.68</td>
<td>5.91</td>
</tr>
<tr>
<td>realworld1</td>
<td>47</td>
<td>44</td>
<td>297.4</td>
<td>24.6</td>
<td>120.4</td>
<td>10.3</td>
</tr>
<tr>
<td>tpc-ds-7q</td>
<td>14</td>
<td>25</td>
<td>62.0</td>
<td>2.9</td>
<td>52.8</td>
<td>3.7</td>
</tr>
<tr>
<td>tpc-ds-63q</td>
<td>85</td>
<td>310</td>
<td>1488.2</td>
<td>60.2</td>
<td>1162.5</td>
<td>78.7</td>
</tr>
</tbody>
</table>

data-intensive and benefit from caching. The precedence graph for tpc-ds-7q is shown in Figure 4.2.

- In one experiment, we also use large randomly-generated precedence graphs to test the scalability of our algorithms. These will be described later.

We use the small running and test workloads to test how close the solutions obtained by the heuristics are to an optimal solution; these are the only workloads on which it was feasible to run the A* algorithm. The realworld and TPC-DS workloads show that Greedy and Heuristic are scalable and outperform Baseline. We have the task output sizes for realworld and TPC-DS, so these are also used to test the WTMB version of our problem.

4.1.2 Comparison of Scheduling Algorithms

In the first set of experiments, we generate schedules using all the algorithms and report how long it takes to create the schedules and the TMB cost of the schedules (even though we know the output sizes for some workloads, we ignore them for now and will consider WTMB shortly). We do not actually run the workloads in a database system. Table 4.1 shows the results. Each row corresponds to a different workload (we omit tpc-ds-scan as it is only relevant to the PostgreSQL experiments later in this section). For Baseline, Greedy and Heuristic, we report the mean TMB score and the standard deviation (SD) over 100 runs, since these algorithms break ties randomly and therefore may return different schedules for the same input. Bold TMB numbers indicate the best algorithms. For Heuristic and A*, we also count the number of node visits during execution (the number of node visits for Baseline and Greedy is small and not reported). Note that A* finished running within one hour only on small problem instances and the number of nodes it visits is very large.

To summarize the results so far: for the workloads where A* was able to finish, we see that Heuristic gives a nearly-optimal schedule. Greedy also works well for some of the smaller problem instances. Baseline gives the most expensive schedules. On the other hand, Baseline and Greedy are extremely fast, Heuristic is still very fast, and A* is the slowest.

Figure 4.3 compares the TMB and WTMB costs of the schedules returned by Heuristic, Greedy and Baseline for the workloads that come with output sizes, namely real-world and TPC-DS (these workloads are too large for A* to handle). The average costs and error bars are included, based on 100 runs of each algorithm.
Figure 4.3: Comparison of a) TMB costs, b) WTMB costs assuming the algorithms are optimizing for TMB, and c) WTMB costs assuming the algorithms are optimizing for WTMB.

Figure 4.3(a) starts with the TMB costs that were already reported in Table 4.1, indicating that both Greedy and Heuristic are significant improvements over Baseline, and that Heuristic is the overall winner (but it takes longer to compute the schedules).

In Figure 4.3(b), we show the WTMB costs of the schedules from Figure 4.3(a), i.e., we have the algorithms optimize for TMB as before, but we compute the WTMB score of the resulting schedules by incorporating output sizes (which, of course, were not given to the algorithms). Heuristic continues to give the best and most stable results—note the wide error bars for Baseline and Greedy. In particular, different runs of Greedy may give widely different TMB results. For instance, if there are several base tables with various sizes but same TMB costs, Greedy randomly chooses the first table to update, regardless of the table size.

Note that the y-axis scales of Figure 4.3(a) and Figure 4.3(b) are different. TMB effectively assumes that each output size is one, whereas the WTMB scores are much higher because they reflect the true sizes of the inputs.

Figure 4.3(c) shows the WTMB scores assuming the algorithms know the output sizes and are actually optimizing for WTMB, not TMB. Comparing to Figure 4.3(b), which has the same y-axis scale, knowing the output sizes clearly helps to lower the WTMB score of the resulting schedules for Greedy and Heuristic. Interestingly, Greedy slightly outperforms Heuristic in this experiment, meaning that greedily selecting tasks with the lowest WTMB scores is a good strategy (and there are no more ties that Greedy has to break randomly, unless the output sizes are exactly the same). We hypothesize that Heuristic could be tuned for the WTMB problem, e.g., by incorporating output sizes in the deepest-first schedule generation, but even now it is not much worse than Greedy.

4.1.3 Scalability Comparison

In this experiment, we randomly generate very large precedence graphs (much larger than those used in the previous experiment) and measure the running time of Baseline, Greedy and Heuristic. Table 4.2 reports the number of nodes and edges for each random graph and the running times. Baseline and Greedy are very simple algorithms and scale extremely well. Heuristic does not scale as well, but can still handle graphs with up to 1000 nodes and edges in reasonable time (few or tens of minutes).
4.1.4 PostgreSQL Experiments

In this set of experiments, we execute various workloads under various schedules in the PostgreSQL database to measure the real-world performance improvements of our techniques. Here, we focus on the disk-RAM hierarchy.

Experiment 1

We start by running the simple tpc-ds-scan workload of three queries that scan three base TPC-DS tables:

Q1: select count(*) from catalog_sales;
Q2: select count(*) from web_sales;
Q3: select count(*) from store_sales;

In PostgreSQL, these queries result in full table scans, resulting in an I/O intensive workload. We also use the following three functions calls to drop tables from the cache:

X1: select drop_table_cache(catalog_sales);
X2: select drop_table_cache(catalog_sales);
X3: select drop_table_cache(catalog_sales);

We create three schedules, S1, S2 and S3, executing each query three times to demonstrate the differences in running time. In schedule S3, we also actively evict tables from the cache when they are not needed any more, indicated by the operations X1, X2 and X3.

S1: Q1,Q2,Q3, Q1,Q2,Q3, Q1,Q2,Q3
S2: Q1,Q1,Q1, Q2,Q2,Q2, Q3,Q3,Q3
S3: Q1,Q1,Q1,X1, Q2,Q2,Q2,X2, Q3,Q3,Q3,X3

In Figure 4.4(a) and 4.4(b), we show the processing time and read I/O, respectively, of the three schedules under varying amounts of available cache (RAM), ranging from 500MB to 10GB in steps.
of 100MB. We control this by running a program that allocates and fills a specific amount of memory, making that amount unavailable to the database. In each experimental iteration, we first execute one schedule, force the operating system to drop all caches and then execute the next schedule.

Figure 4.4(a) reports the processing times. *Schedule 1* is clearly the least efficient, so long as either no data fit into RAM (under 2GB) or all data fit into RAM (over 8.4GB). Additionally, *Schedule 3* achieves even better performance since tables that are not required any more are explicitly removed from the cache, simulating an optimal cache eviction strategy. The largest difference occurs at 4.5 GB of free RAM since now the biggest of the three tables fits entirely into the disk cache. At this point, *Schedule 1* finishes in 45 seconds while *Schedule 2* and *Schedule 3* in 33 seconds and 26 seconds, respectively. The resulting performance increase of *Schedule 3* over *Schedule 1* is 73 percent.

Figure 4.4(b) illustrates the amount of disk read I/O during the run of this experiment under the same varying available cache conditions. These results indicate that there is a correlation between the amount of disk read I/O during the execution of a schedule and its execution time.

*Schedule 1* needs to fit nearly all the tables into the cache before larger amounts of data can be reused. In *Schedule 2*, much more data can be reused through the cache and but even for larger amounts of available cache, between 4.5 and 8.4 GB, some data has to be fetched multiple times from disk. Finally, in *Schedule 3* since an optimal cache eviction strategy is applied, as soon as the biggest table fits entirely into the cache, data are only fetched once from disk.

We conclude from this experiment that changing the execution order of a workload can reduce the amount of disk I/O if not all the data fit into the cache, which also influences the execution time of a workload if it is I/O bound.
Experiment 2

Next, we show that the reduced amounts of disk read I/O are reproducible with queries from the TPC-DS benchmark. We use the *tpc-ds-7q* workload, consisting of 7 data-intensive queries, and execute them on TPC-DS tables generated using scale factor 100. This gives approximately 100GB of data. During the execution of a schedule, whenever a table is no longer needed in that schedule, it is evicted from the cache using the `drop_table_cache()` function, simulating an optimal cache eviction strategy.

We create schedules using Baseline, Greedy and Heuristic, each *not* considering the sizes of the outputs (tables), i.e., optimizing for TMB, not WTMB. We run the experiment four times and reduce the total amount of available system memory from 64 GB to 16 GB in steps of 16 GB, simulating machines with different amounts of available memory. We do this in the same manner as in the previous experiment.

Figure 4.5(a) shows the results, with disk read I/O on the y-axis. For 64 and 48 GB, only the Baseline schedule shows increased amounts of disk I/O. At 32 GB of available RAM, Heuristic performs better than Baseline. Although Greedy is also better than Baseline, it does not perform as well as
Table 4.3: Average and Maximum cache usage for different schedules of the tpc-ds-63q workload.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>10.439</td>
<td>100%</td>
<td>15.025</td>
<td>100%</td>
</tr>
<tr>
<td>Greedy TMB</td>
<td>7.514</td>
<td>71%</td>
<td>13.217</td>
<td>88%</td>
</tr>
<tr>
<td>Greedy WTMB</td>
<td>4.430</td>
<td>42%</td>
<td>9.529</td>
<td>63%</td>
</tr>
<tr>
<td>Heuristic TMB</td>
<td>9.190</td>
<td>88%</td>
<td>9.586</td>
<td>63%</td>
</tr>
<tr>
<td>Heuristic WTMB</td>
<td>7.113</td>
<td>68%</td>
<td>10.511</td>
<td>70%</td>
</tr>
</tbody>
</table>

Finally, for the 16 GB case all schedules perform nearly the same.

Figure 4.5(b) shows the results of a similar experiment, but one in which we also give the algorithms the table sizes, allowing the algorithms to optimize for WTMB. The main difference compared to Figure 4.5(a) are the reduced I/O reads for the 32 GB run for Greedy and Heuristic. Even in the 16 GB case, these algorithms perform better than Baseline.

If we extrapolate the memory needs linearly, the Greedy and Heuristic schedules would need about 320 GB to execute without additional read I/O for the larger TPC-DS scale factor of 1000.

Experiment 3

In the last experiment, we study the effect of different schedules on cache usage over time. We use the tpc-ds-63q workload, which contains the 63 most data-intensive queries. We use scale TPC-DS scale factor 10. Only the raw data are imported into the database, without creating any further auxiliary data structures such as indices. We do not optimize the queries in terms for processing time since our focus is on I/O optimization via scheduling to exploit the cache. Before each experimental run, we first clear the cache and then we execute the tpc-ds-63q workload in the order specified by the scheduling algorithm. Every second, we sample the disk cache usage. As soon as a table is no longer needed by any other query in the schedule, we remove it from the cache using the drop_table_cache() function. Note that we never actively load any data into the cache, but instead tables are loaded automatically when they are accessed by a query.

Figure 4.6(a) shows the RAM usage (on the y-axis) as a function of time since the start of the experiment (x-axis) for Baseline, Greedy and Heuristic optimizing for TMB (i.e., the algorithms do not know the table sizes). Figure 4.6(b) shows the results for WTMB. In both cases, the cache usage over time is greatly reduced by Greedy and Heuristic as compared to Baseline.

Finally, we compare the algorithms by their areas under the cache usage curves. If $c_s(t)$ denotes the cache usage of schedule $s$ at any given point in time $t$, we can formulate the average cache usage for the total execution time $\Delta T$ as $\overline{c}_{s}$ as:

$$\overline{c}_s = \frac{\int_{t_0}^{t_0+\Delta T} c_s(t) \, dt}{\Delta T}.$$  \hfill (4.1)

The resulting cache usages are shown in Table 4.3. The schedule of the Greedy algorithm needs on
average only 42% of RAM of the Baseline schedule. This leaves either more RAM for other concurrently running applications or results in less read I/O if RAM is scarce.
5 Web Page Reports
The Blockmon distributed architecture consists of an overlay made up of Blockmon nodes, which are coordinated by the Blockmon Controller (BC).

The BC is implemented in Python and the communication between the BC and the nodes, as well as the communication between the BC and the outside world takes place by means of JSON-RPC. Furthermore, the BC stores information about nodes and running templates on an internal SQLite database.

The code of the BC, together with instruction on how to run it, are released open source on GitHub at: Blockmon Controller.

The following figure outlines the overall architecture of the Blockmon controller and its interaction with the other components, either external or the Blockmon nodes.

**Interaction between the Blockmon Controller and the external components**

This interaction allows the external components to store template definitions on the BC, to invoke template instances and execute applications distributed across Blockmon nodes. Additionally, the BC makes available functions that allow the other components to retrieve information from the nodes about the available topologies and blocks, as well as the status of a variable of a running application. The BC stores all information about blocks, nodes and templates in a database.

The supported APIs that the BC makes available to accomplish those tasks are the following:

- **put_template(templatedef xml)**: Uploads the XML for a template definition to the BC.
- **get_templates()**: Returns a list of templates definitions (i.e., XML files) already defined at the BC.
- **remove_template(templatedef ID)**: Removes the template definition with the given ID from the list of defined at the BC.
- **expand_template(templateinstance xml)**: Expands a given template instance file based on a previously-uploaded (i.e., through put_template) template definition. Note that the template instance file contains information about which template definition is relevant.
- **invoke_template(templateinstance xml)**: Invokes (i.e., installs) an application on the overlay based on a previously-uploaded (i.e., through put_template) template definition and a given template instance file. Note that the template instance file contains information about which template definition is relevant.
- **stop_template(template ID)**: Stops an application on the overlay based on a previously-uploaded (i.e., through put_template) template definition and a given template ID.
- **get_supported_blocks()**: Returns a list of the blocks currently installed on the overlay.
- **get_supported_topologies()**: Returns a list with descriptions of the topologies currently supported by Blockmon.

https://www.ict-mplane.eu/public/blockmon-controller
Blockmon Controller | Building an Intelligent Measurement Plane for the Internet

- **get_variable(dist comp ID, comp ID, block ID, var ID)**: Returns the value of a variable belonging to a block in a composition.
- **save_datafile(filename, datafile)**: Uploads the data file needed to run a given template instance to the BC.

All the above functions return 0 upon success or -1 upon failure.

**Interaction between the Blockmon Controller and the Blockmon nodes: Node Membership Management and Capabilities Discovery**

This interaction allows the Blockmon node to join the Blockmon network and show its availability to run distributed applications. During this process, the node sends information such as hardware and software specs to inform the BC of its capability. Further, the BC will make this information available, where permitted, to advanced users again via a JSON-RPC call. The BC stores information about nodes into a database.

The supported APIs that the BC makes available to accomplish those tasks are the following:

- **register(IP address, listening_port)**: accepts a request to join the overlay sent by the node with the given IP address and listening_port.
- **unregister(node ID)**: accepts a request to unregister from the overlay sent by the node with the given ID.
- **keepalive(node ID, status_information)**: accepts a keepalive message sent by the node with the given ID and updates statistics about its status.

Upon success of the register function, the BC sends back to the node an ID to identify it within the overlay. The ID is an integer number, computed as an hash of the IP address and listening_port of the node. Upon success of the keepalive and register functions, a success code 0 is returned. All the above functions return -1 upon failure.

**Interaction between the Blockmon Controller and the Blockmon nodes: Running compositions and Installing data file**

This interaction allows the BC to control compositions that are running on a given Blockmon node. This includes the ability of the BC to send data files to the node if the composition needs it and reading or overwriting the value of a given variable.

The supported APIs that the Blockmon nodes make available to accomplish those tasks are the following:

- **start_composition(composition xml, datafiles)**: accepts a request to start a composition and stores the data files needed to run the composition.
- **update_composition(composition xml)**: updates an already running composition.
- **stop_composition(composition ID)**: accepts a request to stop the running composition with the given ID.
- **read_variables(composition ID, variables list)**: reads the value of given variables.
- **write_variables(composition ID, variables list)**: overwrites the value of given variables.
- **get_composition_ids()**: returns the IDs of all compositions currently running.
- **get_running_compositions(composition IDs)**: returns the composition XML for the given IDs.

All the above functions return -1 upon failure.

**Blockmon Controller as mPlane-compliant component**

Blockmon Controller is able to communicate with other mPlane components, thanks to the interface that makes it mPlane compliant. Please refer to the following GitHub repository for the latest version of the interface:

```sh
git clone https://github.com/fp7mplane/components.git COMPONENTS_DIR
```

Once Blockmon Controller is properly installed, get the latest version of the mPlane protocol RI at the following GitHub repository:

```sh
git clone https://github.com/fp7mplane/protocol-ri PROTOCOL_RI_DIR
```

The following instructions assume you are in the `[COMPONENTS_DIR]` folder.
1. Set the parameters in the file `blockmon-controller/blockmonController.conf` (e.g., path to certificates, supervisor address, client port and address, and roles)

2. Set the environment variable `MPLANE_RI` to point to `[PROTOCOL_RI_DIR]`

   ```shell```
   $ export MPLANE_RI=[PROTOCOL_RI_DIR]
   ```shell```

3. Set the following parameters in the file `blockmon-controller/blockmonController.py` to connect to the Blockmon controller

   ```python```
   _controller_port = Blockmon Controller_port
   _controller_address = Blockmon Controller address
   ```python```

4. Run Blockmon Controller

   ```shell```
   $ python3 blockmonController.py --config blockmonController.conf
   ```shell```
Description

DBStream is a flexible, scalable and easy to use Data Stream Warehouse (DSW) designed and implemented at FTW. The main purpose of DBStream is to store and analyze large amounts of network monitoring data. Indeed, DBStream is tailored to tackle the requirements of Network Traffic Monitoring and Analysis (NTMA) applications, both in terms of storage and near real time data processing and analysis. DBStream is a repository system capable of ingesting data streams coming from a wide variety of sources (e.g., passive network traffic data, active measurements, router logs and alerts, etc.) and performing complex continuous analysis, aggregation and filtering jobs on them. DBStream can store tens of terabytes of heterogeneous data, and allows both real-time queries on recent data as well as deep analysis of historical data.

DBStream is implemented as a middle-ware layer on top of PostgreSQL. Whereas all data processing is done in PostgreSQL, DBStream offers the ability to receive, store and process multiple data streams in parallel. As we have shown in a recently published benchmark study [2], DBStream is at least on par with recent large-scale data processing frameworks such as Hadoop and Spark.

One of the main assets of DBStream is the flexibility it provides to rapidly implement new NTMA applications, through the usage of a novel stream processing language tailored to continuous network analytics, called CEL (Continuous Execution Language), this declarative, SQL-based language is highly precise yet very easy to use. Using CEL, advanced analytics can be programmed to run in parallel and continuously over time, using just a few lines of code.

The near real time data analysis is performed through the online processing of time-length configurable batches of data (e.g., batches of one minute of passive traffic measurements), which are then combined with historical collections to keep a persistent collection of the output. Moreover, the processed data can then be easily integrated into visualization tools (e.g., web portals).

In DBStream, base tables store the raw data imported into the system, and materialized views (or views for short) store the results of queries such as aggregates and other analytics — which may then be accessed by ad hoc queries and applications in the same way as base tables. Base tables and materialized views are stored in a time-partitioned format inside the PostgreSQL database, which we refer to as Continuous Tables (CT). Time partitioning makes it possible to insert new data without modifying the entire table; instead, only the newest partition is modified, leading to a significant performance increase.

A job defines how data are processed in DBStream, having one or more CTs as input, a single CT as output and an SQL query defining the processing task. An example job could be: "count the distinct destination IPs in the last 10 minutes". This job would be executed whenever 10 new minutes of data have been added to the input table (independently of the wall clock time) and stored in the corresponding CT.

DBStream consists of a set of modules running as separate operating system processes. The Scheduler defines the order in which jobs are executed, and besides avoiding resource contention, it ensures that data batches are processed in
chronological order for any given table or view. **Import** modules may pre-process the raw data if necessary, and signal the availability of new data to the **Scheduler**. The scheduler then runs jobs that update the base tables with newly arrived data and create indices, followed by incrementally updating the materialized views. Each view update is done by running an SQL query that retrieves the previous state of the view and modifies it to account for newly arrived data; new results are then inserted into a new partition of the view, and indices are created for this partition. **View Generation** modules register jobs at the **Scheduler**.

Finally, the **Retention** module is responsible for implementing data retention policies. It monitors base tables and views, deleting old data based on predefined storage size quotas and other data retention policies. Since each base table and view is partitioned by time, deleting old data is simple: it suffices to drop the oldest partition(s).

The **DBStream** system is operated by an application server process called **hydra**, which reads the **DBStream** configuration file, starts all modules, and monitors them over time. Status information is fetched from those modules and made available in a centralized location. Modules can be placed on separate machines, and external programs can connect directly to **DBStream** modules by issuing simple HTTP requests.

**Deployment Requirements and Execution**

**DBStream** and the used libraries assume that you are using golang version 1.2.x (https://golang.org/). Therefore, for older versions of Ubuntu like e.g. 12.04 you might follow the instructions in this guide: http://www.tuomasrikkenen.fi/Installing-go-1-2-on-Ubuntu-12-04-1s. Next we provide a step-by-step description on how to install and run **DBStream**, as well as how **DBStream** is integrated in moFane.

**la - Installing DBStream**

**DBStream** source code uses the go language; to compile the go source code of **DBStream** you have to install the go language:

**apt-get install go**

**DBStream** also uses several open source libraries which you have to install in order to compile **DBStream**. First you need to create a directory where go code can be downloaded to, e.g.:

```
mkdir ~/go
```

Next you need to export a new environment variable so go knows where to put the code, which at least in bash works like this:

```
export GOPATH=~/go
```

Now you can install the needed libraries with the following command:

```
go get github.com/lin/go-pgsql
  go get github.com/go-martini/martini
  go get code.google.com/p/vitess/go/cgzip
```

Now go to the **DBStream** source directory e.g.:

```
cd ~/source/dbstream/
```

and run the build script there:

```
./build.sh
```

The resulting executables will be placed in the `\texttt{bin/}` directory. The main executable is called **hydra** which starts the application server.

```
cd bin/
./hydra --config ..//config/serverConfig.xml
```

Edit the server configuration and add the modules of **DBStream** you want to use. If you want to get some information about the application server you can run the command remote to monitor and control the server. This command shows the current status of the application server every second:

```
watch -n 1 ./remote
```

The default config also starts a CopyFile module. You can see that currently no files are being imported by checking:

```
http://localhost:3000/DBImport
```

The next step before actually running **DBStream** is to set up Postgres as a **DBStream** backend. The first step is to install **PostgreSQL**. We where using versions from up to 8.4 for **DBStream**, but rather recommend to use newer versions, like e.g.
9.3. On Ubuntu you can install PostgreSQL with the following command:

```
apt-get install postgresql-9.3
```

Then you have to create an operating system and database user for DBStream. From now on, we will assume that this user is called `dbs_test` but you can choose any other user name, just make sure that all parts of the configuration are adapted as well. This user has to be a postgres superuser.

```
sudo useradd -s /bin/bash -m dbs_test
```

Now you have to create a database with the name of that user; note that this database will also be used to store all data imported to and processed with DBStream.

```
sudo su - postgres # change to the postgres user
createnewuser -P -s dbs_test # create new user with superuser rights and set password
createdb dbs_test # create a database with the same name
exit # close the postgres user session
```

DBStream uses a tablespaces to store data on disk, namely `dbs_ts0`. For testing purposes, we will locate them in the home folder of the `dbs_test` user, but in a real setup you probably want to set them to a large RAID-10 storage array.

```
sudo mkdir /home/dbs_test/dbs_ts0 # create data0
sudo chown postgres /home/dbs_test/dbs_ts0 # This directory must be accessible by the postgres system user
```

Now the newly created DBStream database needs to be initialized. Therefore, change to the test directory and login into the database you just created:

```
cd test
psql dbs_test # Please note that you need to login with a database superuser, so you might want to change to the dbs_test user first.
```

If you log correctly into the database you should see something like this:

```
psql (9.3.6)
Type "help" for help.

dbs_test=#
```

Now run the following command to initialize some DBStream internal tables.

```
\i initialize.sql
```

If all steps from this part were successfully completed you can go on and start DBStream for the first time.

**lb - Installing DBStream from Vagrant**

Within mPlane, and to ease the installation of a DBStream instance without the burden of installing and configuring all its components, we provide a DBStream-Vagrant based image at [https://github.com/arbaer/dbstream/tree/master/vagrant](https://github.com/arbaer/dbstream/tree/master/vagrant). Vagrant is a tool for building complete development environments, aiming at lowering development environment setup time.

To run a DBStream instance using Vagrant, you need to follow these steps:


Add the ubuntu/trusty64 box:

```
vagrant box add ubuntu/trusty64
```

Start the virtual machine:

```
vagrant up
```

Connect to the virtual machine:

```
vagrant ssh
```

In case you want to recompile DBStream set the GOPATH environment variable and execute the `build.sh single` threaded:

```
export GOPATH=~/go
./build.sh single
```
II - Running DBStream

Now that DBStream is already installed, follow these steps to start it:

First we need to change to the test directory.

```bash
cd test  # if you are coming here from vagrant, the directory is src/dbstream/test
```

Now you should see the executables in this directory (e.g. `hydra`, `mathprobe`, `mathrepo`, `scheduler` and `remote`). For this example it is the best to open three shells. In the first shell we will run `dbstream`, in the second we will run the `import source` and the third will be used for `monitoring` DBStream.

In the `monitoring` shell run the following command:

```bash
cd dbstream/test
watch -n 1 ./remote
```

In the `dbstream` shell execute the following command:

```bash
cd dbstream/test
./hydra --config sc_tstat.xml
```

In the `import source` shell run the following command:

```bash
cd dbstream/test
./mathprobe --config math_probe.xml --repoUrl "localhost:3000" --startTime 2006-01-02T15:04:05
```

If all went well, you should now be able to log into postgres, and check some preloaded tables:

```sql
psql dbstream
select * from example_log_tcp_complete;
```

To cleanup the tables and run the example import again, inside postgres execute the following command:

```sql
select dbs_drop_table('example_log_tcp_complete');
select dbs_drop_table('tstat_test');
```

and in the shell run:

```bash
rm -rf /tmp/target/
```

III - mPlane Integration

In mPlane, DBStream is integrated and used together with the Tstat probe, storing and analyzing the data captured and exported by the probe. The integration includes a mPlane proxy to the repository (`RepoProxy`) and a data transfer protocol which enables the Tstat probe to send bulk measurements to DBstream, and DBStream to import these measurements into tables for further analysis by the Analysis Modules it runs (e.g., anomaly detection). Data transfer is achieved through a customized protocol we have named MATH - mPlane Authorized Transfer via HTTP.
MATH is composed of two modules, `math_probe` and `math_repo`, the former runs together with the Tstat probe and it handles the transfer of Tstat logs to a mPlane repository, the latter runs together with DBStream and handles the importing of the received Tstat logs into the DBStream database. Both MATH modules come with XML configuration files which extends the flexibility of the MATH protocol to be used with other probes and repositories.

To run the mPlane integration of DBStream with Tstat, you have to follow the next steps:

Download and install the Tstat and DBStream tools, both available at Github under https://github.com/fp7mplane/components/tree/master/tstat and https://github.com/artbw/dbstream, respectively.

Install the mPlane framework and probe and repository mPlane proxies

```
git clone https://github.com/fp7mplane/protocol-ri.git
```

Enter the protocol-ri/mplane folder and rename (or remove) components. Then, check out the one available on github.

```
cd protocol-ri/mplane/
mv components components.orig (or rm -rf components)
git clone https://github.com/fp7mplane/components/
cd ..
```

Add the following required capabilities at the Supervisor configuration files `conf/supervisor.conf`:

```
tstat-log_http_complete = guest,admin
tstat-exporter_log = guest,admin
repository-collect_log = guest,admin
```

Run the mPlane Supervisor:

```
./scripts/mpsup --config ./conf/supervisor.conf
```

Run the Tstat proxy:

```
./scripts/mpcom --config ./mplane/components/tstat/conf/tstat.conf
```

Run the Repository proxy:

```
./scripts/mpcom --config ./mplane/components/tstat/conf/tstatrepository.conf
```

Run the mPlane Client:

```
./scripts/mpcli --config ./conf/client.conf
```

Run both DBStream and the MATH importer module, `math_repo`:

```
./hydra --config sc_tstat.xml
./math_repo
```

Run Tstat and the MATH exporter module, `math_probe`, using the mPlane Client shell:
New Features supported by the mPlane project

Thanks to the support of the mPlane project we extended DBStream functionalities with the following features:

- **CEL Extension**: we extended the functionality of the CEL language, making it much easier to code an analysis job on top of DBStream.
- **MATH**: we added MATH (Mplane Authorized Transfer via HTTP), a protocol to export bulk data in the form of logs from a mPlane probe (e.g., Tstat) and to import it into DBStream.
- **Machine Learning @DBStream**: we added Machine Learning analysis capabilities to DBStream, by integrating a well-known Machine Learning toolbox (WEKA) directly into the data processing of DBStream jobs.
- **Better Performance through Scheduling**: we improved the performance of DBStream in terms of complete job completion time by studying and developing different job scheduling approaches.

References

All DBStream and DBStream-related (e.g., MATH) sources and binaries are accessible on GitHub at https://github.com/arbaer/dbstream.

Additional documentation on DBStream can be found at https://github.com/arbaer/dbstream/blob/master/README.md.

Further information about DBStream can be found in the following research papers:


If you are using DBStream for any research purpose we would highly appreciate if you would reference [2].

DBStream is open source software and the full source code along with a detailed installation description are available under the AGPL license on github.com at: https://github.com/arbaer/dbstream.
Description

**EZRepo** is a repository Component, fully compatible with the mPlane software framework. This means that this repository is controlled and queried through the mPlane protocol.

The data processing/collapsing functionality provided by EZRepo makes it suitable for trouble/root cause analysis tasks in the first place.

The operation of the repository builds on the concept of *periods*. Periods are tuples of attributes describing the availability and access quality of some service (e.g., network connectivity or availability of some piece of content) during a certain period. Events are created based on some kind of qualification - i.e., "grading" - of some probe measurements, along various criteria.

As an example, an OTT probe measuring a piece of content may experience "EXCELLENT" quality for a long time, then the quality may go down to "NOT AVAILABLE", possibly going through a "POOR" period before that. (We recommend simple grading with only a few number of options (3..5).

Queries against EZRepo will look for measurement records which match some network criteria (e.g., measurement probe, measurement type, source IP, destination IP, network path, etc.) and also match some grade selection criteria (bandwidth or overall grade, grade level, etc.).

Results returned by EZRepo will provide the number of records which match the network selection criteria, and the number of records which match both the network and the grade selection criteria.

In the example figure given, the information collected by the GLIMPSE probes are queried from EZRepo. The specification defines a query from all UDP protocol based measurements done via GLIMPSE, where destination IP address is 1.2.3.4, and the source IP and network paths can be arbitrary. We are looking for all the records within this set, where the bandwidth grade is in the range 3..5.

The result values will show us that 11 records has been found with the given selection criteria, from which 3 was graded in the given quality range.

Architecture

EZRepo will collect measurement records from different probes (e.g., OTT probe, GLIMPSE, etc.), with different measurement scopes and capabilities.
EZ Repo has 3 main modules, the **Data Store**, the **Query Engine** and the **Classifier**.

Probes send their data into the EZRepo with UDP protocol. (Other protocols like HTTP and SCP are under consideration/development).

The **Classifier** module will make the evaluation ("grading") of the input data, based on the threshold data stored in a JSON file ("Grading.json"). For practical reasons we use a 5 grades scale which complies to the common (‘normal-warning-minor-major-critical’) qualification and notification levels and conforms to the ITU standards as well. Thresholds can be published through the ‘SetGrading’ capability to the Supervisor.

The classified data, along with the original measurements are stored in the **Data Store**.

The **Query Engine** receives the specifications (’QueryByCriteria’) and returns the matching results.

**Grading.json**

The **Grading.json**, which is downloadable from [GitHub](https://github.com/fp7mplane/components/tree/master/EZRepo), contains the definitions for the different grades for the different measurement types. As en example, for GLIMPSE probe based ping measurements, the definitions are looking like that:

```json
{
    "grading": [
    {
        "gradeName": "overall",
        "appliesTo": {
            "recordType": "GLIMPSE",
            "recordSource": "*",
            "recordDest": "*",
            "recordContent": "ICMPECHO"
        },
        "gradingRules": [
            {
                "grade": "5",
                "if": [
                    {
                        "metric": "rtt.ms.avg",
                        "rel": "<",
                        "limit": "20"
                    },
                    {
                        "metric": "rtt.ms.max",
                        "rel": "<",
                        "limit": "50"
                    }
                ]
            }
        ]
    }...
```

Which resolves as: for the 'overall' grade we are searching ICMPECHO (i.e. ping) measurement records collected with GLIMPSE, with any source, destination; and ‘grade’ will result value "5" if the value of 'rtt.ms.avg' is smaller than 25 and the value of 'rtt.ms.avg' is smaller than 50.

**Quick start**

Download EZRepo package from github: [https://github.com/fp7mplane/components/tree/master/EZRepo](https://github.com/fp7mplane/components/tree/master/EZRepo)

Copy to ~/protocol-ri/components

Change component.conf and the other config files accordingly if needed (certificates, authorization and roles settings, etc)

Register through the component module:

```
$ scripts/mpsup --config <supervisor\_config>
$ scripts/mpcom --config <component\_config>
```

Access it through the clientshell:

https://www.ict-mplane.eu/public/ez-repo
$ scripts/mpcli --config <client\_config>

References
Links to sources, binaries
https://github.com/fp7mplane/components/tree/master/EZRepo

The information available on this website is property of the contributing authors from the mPlane Consortium (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission. The information in this website is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Privacy
MATH, Mplane Authorized Transfer via HTTP, is a tool developed by FTW to export bulk data in the form of logs from a mPlane probe (e.g., Tstat) and to import it into DBStream.

MATH is composed of two modules, `math_probe` and `math_repo`, the former runs together with the probe and it handles the transfer of logs to a mPlane repository, the latter runs together with DBStream and handles the importing of the received logs into the DBStream database. Both MATH modules come with XML configuration files which extend the flexibility of the MATH protocol to be used with other probes and repositories.

The MATH source code and binaries are available on GitHub at [https://github.com/ariege/dbstream/tree/master/src/modules/math_import](https://github.com/ariege/dbstream/tree/master/src/modules/math_import)

Below you can find a description of the included files:

`math_probe` -- handles the probe part, i.e. it provides the ability to download files following the MATH protocol.

`math_repo` -- handles the download of files.

`math_probe.xml` -- is the configuration file for the probe. The most important part is the directory XML attribute to configure the directory to copy.

`math_repo.xml` -- is the configuration file for the repository. The most important part here is the output XML attribute of the filetransferConfig XML element, which is used as the target for the files being copied.

To start the transfer of file using MATH you first need to configure and start the repository part:

```
./math_repo
```

Then, in another shell or in the other machine where data is coming from you need to start the probe part:

```
./math_probe --repoUrl "localhost:3000" --startTime 2006-01-02T15:04:05
```

Note that the `math_repo` can be used in combination with the DBStream hydra server process or as a stand alone executable.
Repository mPlane interfaces for Tstat is a set of mPlane protocol-based tools to import logs and RRDs generated by Tstat (developed by POLITO and FTW) into mPlane compliant repositories. The code is located in the github page of mPlane.

Currently the repository interfaces supports three different indirect importing approaches:

1. Log bulk importer
2. Log streaming importer
3. RRD importer

The repository interface plays the role of a proxy which imports data from Tstat probes using the mPlane protocol. The repository proxy register itself to the supervisor and enables its indirect export capabilities at the registration phase. Simple ad-hoc protocols have been developed to setup the asynchronous and indirect export between the Tstat proxy, and the repository proxies.

To activate the indirect import capabilities on Tstat logs and RRDs, it is recommended to activate the `tstatrepository.py` as well as the `tstat.py`, as described in the Tstat page.

You can run the mPlane repository interface by entering into protocol-ri folder and execute:

```bash
>>> export PYTHONPATH=.
>>> ./scripts/mpcom --config ./mplane/components/tstat/conf/tstatrepository.conf
```

The tstatrepository component in the protocol-ri supports all the aforementioned indirect export techniques. We detail their some use example in the following.

### Log bulk importer

The log bulk importer uses MATH (Mplane Authorized Transfer via HTTP) developed by FTW to import data from Tstat and store them in DBStream. It can be customized to dump logs to different databases, or to simply copy and store them on the repository filesystem.

In mPlane, client can run the specification from now to unlimited future. For instance, in the example below, we instruct Tstat proxy to start exporting data from the probe to a repository proxy, which is running on the same host (localhost).

```
|mplane| runcap tstat-exporter_log
|when| = now + inf
|repository.url| = localhost:3000

OK
```

NOTE: The `repository.url` contains the IP address of repository and the port value associated to `repository_log_port`.

### Log streaming importer

The repository interface enables the streaming of logs collected in real-time by Tstat. The code contains in `repository_streaming_importer.py` acts as a simple endpoint server which receives the streamed log and redirect them to the stdout. It is possible to customize the code of the file `repository_streaming_importer.py` to redirect the data to other consumers such as, e.g., a WeBrowse module.

For instance, to activate the streaming indirect export of log_tcp_complete for 1 day, execute (as above, repository and
...running both on localhost):

```bash
mplane| runcap tstat-exporter_streaming
|when| = now + 1d
log.folder = /path/to/log/folder/  # insert here the path where the logs are
log.time = 60
log.type = log_tcp_complete
repository.url = localhost:9001
ok
```

NOTE: The `repository.url` contains the IP address of the repository and the port value associated to `repository_streaming_port` in `tstatrepository.py`.

## RRD importer

The repository proxy imports the RRD files collected by the Tstat. The RRD files are then sent to Graphite for storage and graphical presentation.

For instance, to activate the RRD indirect export form now to 1 hour, run at the client (as above, repository and probe running both on localhost):

```bash
mplane| runcap tstat-exporter_rrd
|when| = now + 1h
repository.url = localhost:9000
ok
```

NOTE: The `repository.url` contains the IP address of the repository and the port value associated to `repository_rrd_port` in `tstatrepository.py`.

---

**Official version**

- July 10th, 2015 - frozen release for D3.4
- Tstatrepository-proxy [tar.gz] -- NEW -- this archive includes tstatrepository.py, all `repository_importers` (bulk log, streaming and rrd), and the corresponding configuration files. However, we suggest to use the GitHub version. See above.

---

The information available on this website is property of the contributing authors from the mPlane Consortium (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission. The information in this website is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Privacy
Description:
The mobile probe stores data in a MongoDB database in json format.

For this purpose a MongoDB to mPlane reference implementation proxy has been created.
The proxy is not specific to the mobile proxy and, therefore, it can be configured to extract data from any MongoDB schema.

Repository Description:
The repository consists of a MongoDB database and a node.js interface. The Mobile Probe sends HTTP requests from the device to the server with the aggregated measurements. The node.js script is in turn responsible for receiving, parsing and storing the incoming data to the database. Each measurement is inserted with fields such as timestamp, the device's unique ID and IP address. In this way, the repository can handle concurrent reports from multiple devices. As can be seen in the provided sample below, a report contains the required performance metrics from the different layers that the probe is monitoring.

```json
{
    "report":{
        "measurementsArray":[
            {
                "timestamp": "2015-03-31 13:19:57.608",
                "LOCATION":{
                    "timeStamp": 1427800797607,
                    "provider": "NO_LOCATION",
                    "measurementType": "location",
                    "contentType": "Measurement"
                },
                "extras": "mplane",
                "deviceID": "352605059221028",
                "ip": "83.53.30.1",
                "version": "2"
            },
            {
                "CELL_INFO":{
                    "measurementReason": "onSignalStrengthsChanged",
                    "networkOperatorName": "",
                    "currentCellLocationCID": 125067479,
                    "timeStamp": 1427800798013,
                    "measurementType": "cellInfo",
                    "contentType": "Measurement",
                    "neighborCellInfo": [
                        {
                            "currentCellLocationLAC": 864,
                            "dataActivity": ",
                            "networkOperatorCountryCode": "es",
                            "latestGSMBitError": 99,
                            "currentCellLocation": "[864,125067479,-1]"
                        }
                    ]
                }
            }
        ]
    }
}
```
"getNetworkType":3,
"currentCellLocationPSC":-1,
"dataState":,
"latestGSMSignalStrength":30,
"networkOperatorCode":"21407"
},
"timestamp":"2015-03-31 13:19:58.050",
"extras":"mplane",
"deviceID":"352605059221028",
"ip":"83.53.30.1",
"version":"2"
},
{"timestamp":"2015-03-31 13:19:58.851",
"HW":{
  "network_type":"WiFi",
  "wifi_RSSI":-37,
  "Cpu Tot":":1,
  "MemFree":":20908"
},
"extras":"mplane",
"deviceID":"352605059221028",
"ip":"83.53.30.1",
"version":"2"
},
{"timestamp":"2015-03-31 14:54:55.716",
"LogCat":"I/AwesomePlayer( 1854): setDataSource_l(URL suppressed)",
"version":"2",
"deviceID":"358848043406974",
"ip":"83.53.30.1",
"extras":"mplane"
},
{"timestamp":"2015-03-31 14:54:56.546",
"LogCat":"I/AwesomePlayer( 1854): Could not offload audio decode, try pcm offload",
"version":"2",
"deviceID":"358848043406974",
"ip":"83.53.30.1",
"extras":"mplane"
},
{"tcp_complete":{
  "c_sack_opt":":1",
  "s_rtt_avg":":0.914270",
  "s_cwin_max":":59856",
  "c_win_min":":14600",
  "s_last":":11838.654000",
  "s_pks_dup":":0",
  "s_rtt_cnt":":965",
  "s_rtt_max":":36.791000",
  "c_pks_dup":", "c_bytes_all":":16916",
  "c_pks_data":":2522",
  "s_pks_unrto":":0",
  "c_pks_reor":":0",
  "s_syn_retx":":0",
  "c_win_max":":522880",
  "p2p_st":":0",
  "ed2k_c2s":":0",
  "c_pks_push":":17",
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_ssl</td>
<td>&quot;3---sn-h5q7ened.googlevideo.com&quot;</td>
</tr>
<tr>
<td>c_pkts_rto</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_rtt_cnt</td>
<td>&quot;18&quot;</td>
</tr>
<tr>
<td>s_sack_opt</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>ed2k_c2c</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_bytes_retx</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_mss</td>
<td>&quot;1460&quot;</td>
</tr>
<tr>
<td>c_bytes_uniq</td>
<td>&quot;16916&quot;</td>
</tr>
<tr>
<td>ed2k_uniq</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_bytes_all</td>
<td>&quot;3481036&quot;</td>
</tr>
<tr>
<td>c_pkts_data</td>
<td>&quot;17&quot;</td>
</tr>
<tr>
<td>c_cwin_min</td>
<td>&quot;235&quot;</td>
</tr>
<tr>
<td>s_pkts_ooo</td>
<td>&quot;19&quot;</td>
</tr>
<tr>
<td>c_pkts_retx</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_rtt_std</td>
<td>&quot;60.187121&quot;</td>
</tr>
<tr>
<td>durat</td>
<td>&quot;41905.741000&quot;</td>
</tr>
<tr>
<td>c_win_scl</td>
<td>&quot;6&quot;</td>
</tr>
<tr>
<td>c_rtt_min</td>
<td>&quot;21.642000&quot;</td>
</tr>
<tr>
<td>s_pkts_fc</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>http_f</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_rtt_max</td>
<td>&quot;187.524000&quot;</td>
</tr>
<tr>
<td>s_ssl</td>
<td>&quot;<em>.</em>.c.docs.google.com&quot;</td>
</tr>
<tr>
<td>s_first_ack</td>
<td>&quot;218.297000&quot;</td>
</tr>
<tr>
<td>ed2k_chat</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_rst_cnt</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_ack_cnt_p</td>
<td>&quot;1195&quot;</td>
</tr>
<tr>
<td>first_abs</td>
<td>&quot;1427836180809.872070&quot;</td>
</tr>
<tr>
<td>s_win_0</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_pkts_fs</td>
<td>&quot;17&quot;</td>
</tr>
<tr>
<td>c_cwin_max</td>
<td>&quot;1093&quot;</td>
</tr>
<tr>
<td>s_cwin_ini</td>
<td>&quot;1440&quot;</td>
</tr>
<tr>
<td>c_rtt_avg</td>
<td>&quot;73.632758&quot;</td>
</tr>
<tr>
<td>s_bytes_retx</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_mss</td>
<td>&quot;1460&quot;</td>
</tr>
<tr>
<td>c_ip</td>
<td>&quot;192.168.1.243&quot;</td>
</tr>
<tr>
<td>c_pkts_unfs</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_win_scl</td>
<td>&quot;7&quot;</td>
</tr>
<tr>
<td>last</td>
<td>&quot;249797.323000&quot;</td>
</tr>
<tr>
<td>c_pkts_dup</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_pkts_unrto</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_fin_cnt</td>
<td>&quot;4&quot;</td>
</tr>
<tr>
<td>s_pkts_unfs</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_sack_cnt</td>
<td>&quot;235&quot;</td>
</tr>
<tr>
<td>s_rtt_std</td>
<td>&quot;2.552807&quot;</td>
</tr>
<tr>
<td>s_win_max</td>
<td>&quot;63872&quot;</td>
</tr>
<tr>
<td>s_ack_cnt</td>
<td>&quot;2529&quot;</td>
</tr>
<tr>
<td>s_pkts_retx</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_win_min</td>
<td>&quot;28960&quot;</td>
</tr>
<tr>
<td>c_mss_min</td>
<td>&quot;235&quot;</td>
</tr>
<tr>
<td>s_port</td>
<td>&quot;443&quot;</td>
</tr>
<tr>
<td>s_pkts_unk</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>s_ttl_max</td>
<td>&quot;57&quot;</td>
</tr>
<tr>
<td>s_pkts_rto</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>s_ack_cnt_p</td>
<td>&quot;5&quot;</td>
</tr>
<tr>
<td>c_port</td>
<td>&quot;40886&quot;</td>
</tr>
<tr>
<td>c_isint</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>s_sack_cnt</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>c_ttl_max</td>
<td>&quot;64&quot;</td>
</tr>
<tr>
<td>c_tm_opt</td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>
mPlane proxy interface

The proxy can be found here: https://github.com/fp7mplane/components/blob/master/mongoDB-mobileProbe/

The proxy Interface is written in python and it sits on top of MongoDB. It exports the collections of mongoDB as capabilities using the Reference Implementation. Afterwards, the mPlane clients can pull the requested data.

The mongoDB proxy can be used by any component that has data stored in a mongoDB database. For details on how to install and run the probe visit the github help file:

https://github.com/fp7mplane/components/blob/master/mongoDB-mobileProbe/

Installation

Copy files from the mPlane interface (from the components GitHub) into protocol-ri/:

https://www.ict-mplane.eu/public/mongo-db
mongo-db | Building an Intelligent Measurement Plane for the Internet
Page 5 of 7
https://www.ict-mplane.eu/public/mongo-db

- mobile_probe_settings.json The registry.json file, copy it into protocol-ri/mplane/components/.
- mongo.py The Python interface, copy it into protocol-ri/mplane/components/.
- supervisor.conf, component.conf and client.conf The configuration files, copy them into protocol-ri/conf/.

Configuration

The mongoDB proxy can be used by any component that has data stored in a mongoDB database. The configuration is given at a json file of the following format:

```json
{
    "measurements":{
        "mobile_probe_hardware":{
            "collection":"mobileMeasurements",
            "search":"HW",
            "return":{
                "observer.link":"HW.network_type",
                "snr":"HW.wifi_RSSI",
                "cpuload":"HW.Cpu Tot: ",
                "memload":"HW.MemFree"
            }
        },
        "mobile_probe_cellInfo":{
            "collection":"mobileMeasurements",
            "search":"CELL_INFO",
            "return":{
                "snr":"lCELL_INFO.atestGSMSignalStrength",
                "observer.link":"CELL_INFO.currentCellLocation"
            }
        },
        "mobile_probe_videoLog":{
            "collection":"mobileMeasurements",
            "capability_name":"mobile_probe_videoLog",
            "search":"LogCat",
            "return":{
                "measurement.identifier":"LogCat"
            }
        }
    }
}
```

The settings above will pull data from db mplane and collection mobileMeasurements. The field "devicID" will be used as the source field when making a query for a specific device. Furthermore, the timestamp field is "date".

The measurements dictionary contains information about the offered capabilities and the association of the return values between the mongoDB database and the mPlane reference implementation. For instance, the mongoDB entry "HW.wifi_RSSI" will be returned as the mPlane's RI snr field.

The settings above will generate three capabilities.

query (mobile_probe_hardware)
query (mobile_probe_videoLog)
query (mobile_probe_cellInfo)

Launching the data extractor

After installing the RI, in a terminal window start supervisor:
cd ~/protocol-ri
python3 -m scripts/mpsup --config ./conf/supervisor.conf

alternatively:

cd ~/protocol-ri
python3 -m mplane.supervisor --config ./conf/supervisor.conf

In another terminal start the mobile probe as a component:

cd ~/protocol-ri
python3 -m scripts/mpcom --config ./conf/component.conf

The expected output for this example should be:

...  
Capability registration outcome:
mobile_probe_hardware: Ok
mobile_probe_videoLog: Ok
mobile_probe_cellInfo: Ok

Checking for Specifications...

Retreiving data

You need to have mongoDB setup and running on your machine. You should also have the settings.json configured in order to access and translate the mongoDB field into the mPlanes RI entries (see above).

Start a client to test the component:

mpcli --config ./conf/client.conf

The expected output should be:

ok
mPlane client shell (rev 20.1.2015, sdk branch)
Type help or ? to list commands. ^D to exit.

Now check that the mongoDB capabilities is registered:

<table>
<thead>
<tr>
<th>mplane</th>
<th>listcap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability mobile_probe_cellInfo (token cd6b5fa2dc4ddf3d7c4f16898bf1b60)</td>
<td></td>
</tr>
<tr>
<td>Capability mobile_probe_hardware (token f3a12891623783d20df63f01e86104d)</td>
<td></td>
</tr>
<tr>
<td>Capability mobile_probe_videoLog (token 57a0a9484788520090bf3e1c97aa6df8)</td>
<td></td>
</tr>
</tbody>
</table>

And run a test capability. In this example we ask for any mongoDB hardware entries for mobile device with IMEI 352605059221028 between 2013-09-20 and 2015-5-5:

<table>
<thead>
<tr>
<th>mplane</th>
<th>runcap mobile_probe_hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>when = 2013-09-20 ... 2015-5-5</td>
<td></td>
</tr>
<tr>
<td>source.device = 352605059221028</td>
<td></td>
</tr>
</tbody>
</table>

ok

If everything works ok the data should be pulled from the mongoDB database and transformed into an mPlane result:

<table>
<thead>
<tr>
<th>mplane</th>
<th>listmeas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result mobile_probe_hardware-0 (token 6ed23ada2c8921f7cf9de917402a4283):</td>
<td></td>
</tr>
</tbody>
</table>

2015-03-31 11:19:58.851000 ... 2015-03-31 11:20:04.377000

You can now access the meassurments:

| mplane | showmeas 6ed23ada2c8921f7cf9de917402a4283 |
result: query
label: mobile_probe_hardware-0
token: 6ed3ada2c892f7cf9de917402a4283
when: 2015-03-31 11:19:58.851000 ... 2015-03-31 11:20:04.377000
registry: http://ict-mplane.eu/registry/demo
parameters (1):
source.device: 352605059221028
metadata (1):
System_version: 1.0
resultvalues (2):
result 0:
cpuload: 1.0
memload: 20908.0
observer.link: WiFi
snr: -37.0
result 1:
cpuload: 0.6805555820465088
memload: 44084.0
observer.link: WiFi
snr: -39.0
repoSim is an ns2-based simulator aimed at assisting the fine-tuning of mPlane repository performance. The overall goal would be to use simulation as a preliminary, necessary step to investigate a broad spectrum of solutions, to find candidate solutions worth implementing in real operational mPlane repositories.

Motivations

The need for such a tool can be clarified considering the following picture, that represent a general mPlane workflow, valid for both active or passive measurements. The picture show a reasoner, or intelligent user interacting with mPlane through a supervisor, triggering WP2 active/passive measurement nodes [yellow arrows], that generate a workflow that will solicitate WP3 repositories.

Low level viewpoint

Specifically, as emerges from the figure above, a mixture of flows insist on mPlane repository: i.e., flows that enter or exit the repository, or even flow that are confined within the repository "data center" network. Such flow include:

- store raw data (eg CSV, binary, ...) [black]
- access raw data (eg FTP, HTTP, ...) [black]
- export raw data (eg IPFIX, ...) [black]
- cooking data to some extent (e.g., MapReduce, or other algorithms) [red]
- generate results and events (i.e., outcome of the above) [black]
- state all the above (i.e., capability) [blue]

To simplify, we see that WP3 large-scale data analysis involves several types of concurrent data flows, that are either confined within the Repository itself, or cross its interface toward other parts of mPlane infrastructure (or external networks). From the architectural viewpoint, it implies: firstly, multiple tools may possibly share the same repository; secondly, even for a single
tool, its control and data workflows are intermingled. Our network resource is multiplexed by different flows of type, size, and load, both within and enter/exit the repository infrastructure.

This has possible consequences not only on the timeliness of the results (e.g., results stuck behind a large transfer), but also possibly about the accuracy of the results themselves (e.g., control messages in iterative drill-down analysis slowed down by fat process transfer) and need careful investigation. Therefore we are facing the challenge to: design, implement and evaluate scheduling protocols for the efficient and fair allocation of networking resources to network data analysis jobs. Such scheduling should consider not only internal data process workloads, but also cooperate data flows coming in and out of the mPlane infrastructure and external network, to support data storage, query, analysis and export.

An even more detailed viewpoint concerning the Repository is shown below, where we use arrows thickness to represent the expected heterogeneity as far as the volume of the exchanges are concerned:

![Diagram of repository structure]

**High level viewpoint**

We are now study and optimize the repo "data center" network performance. For the sake of readability and generality, we argue that it is possible to simplify the above mPlane-centric view to get broadly applicable insights, that also apply to mPlane, by cutting some high frequency details that just add noise in the picture. The simplification comes into considering that there are basically two classes of flows: short or "mice" flows (e.g., events, specification, capabilities) vs fat data or "elephant" transfers (e.g., results, indirect exports, map, etc.)

Under this light, an important observation is that unless proper actions are taken, competing elephant flows would slow down the performance of mice flows, which results in a downgrade of overall Repository/mPlane performance. We are investigating into the design of scheduling protocols to mainly satisfy:

- Sustained throughput to avoid slowdown of data cooking (e.g. elephant MapReduce data transfer in a map phase)
- Low-delay communication for short transactions (e.g. mice control flows)

It appears that the issue is larger than the scope of what can be done within MapReduce schedulers, and rather call for a more systematic analysis at all levels, including general purpose data center solution that may be engineered at the application, transport or network layers.

The ns2 simulator allow to compare effective yet practical solutions that can be implemented in mPlane repository, with state of the art data center solutions that would require a much more involved deployment.

Tuning for mPlane repository simply involves defining a workload size distribution and flow arrival process gathered from operational mPlane repositories, that will be available from the testplant. In the meanwhile, realistic distributions taken from Hadoop clusters are in use for a more general fitting.

**Quick start:**

The installation instruction are detailed in the D33 tarball.

**mPlane proxy interface**

none

**Official version**

https://www.ict-mplane.eu/public/reposim
The information available on this website is property of the contributing authors from the mPlane Consortium (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission. The information in this website is provided "as is"; and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Privacy
Description:

The Hadoop Fair Sojourn Protocol Scheduler (HFSP, available on [github](https://github.com)) is a size-based scheduler for Hadoop that exploits approximate job size estimation to obtain response times that outperform the current state of the art (e.g., processor sharing).

Size-based scheduling with aging has, for long, been recognized as an effective approach to guarantee fairness and near-optimal system response times. HFSP introduces this technique to a real, multi-server, complex and widely used system such as Hadoop. Size-based scheduling requires a priori job size information, which is not available in Hadoop: HFSP builds such knowledge by estimating it on-line during job execution and it is largely tolerant to job size estimation errors.

HFSP can deal with estimation errors by leveraging on the fact that the Hadoop framework provides information about job progression: therefore, HFSP starts with a first rough estimation of job size based simply on the number of map/reduce tasks in a job, but refines this estimation once a few tasks for that job are completed.

**SchedSim** is a simulator for evaluating the impact of errors in estimating the size when performing size-based scheduling in big-data workloads. Details in a technical report.

Please, refer to Deliverable D3.3 for a full description of the HFSP Scheduler.

Quick start:

Please, refer to the quickstart on [github](https://github.com) for the scheduler and on [bitbucket](https://bitbucket.org) for the simulator.
New features supported by the mPlane project

The HFSP Scheduler has been completely built in projects mPlane and BigFoot.

mPlane proxy interface

None

Official version


File:

HFSP Scheduler
Cache-Oblivious Scheduling of Shared Workloads

You can find the code at github.

---

**Pre-Build**

First you have to install the Go language (http://golang.org). In Ubuntu this can be done like this:

```
apt-get install golang
```

In order to compile the code with go you have to set GOPATH enviroment variable e.g. like this:

```
export GOPATH=$GOPATH:/path_to_the_repo/schedule
```

**Build**

Just run the command:

```
./build.sh
```

to build the programs.

**Graph Visualization**

The tool con2gv can be used to generate GraphViz (http://www.graphviz.org) files for given configurations. With

```
./con2gv --config config/running.xml --style name_id > running.gv
```

you can create a gv file, which can then be transformed into your preferred image format using the dot commandline tool.

```
dot -Tsvg running.gv > running.svg
```

**Scheduling**

The tool schedule can be used to generate schedules of a given configuration. Those schedules will full-fill all precedence constraints and, depending on the algorithm you choose, will be optimized for cache usage.

There are four algorithms:

- baseline: Schedules the jobs in a breadth first like way.
- greedy: Tries to always optimize the next step according to the total maximum bandwith costs,
- heuristic: Uses several heuristics to find a suitable schedule.
- a_star: Uses the A* algorithm to find the optimal schedule. Please be aware that this algorithm might consume a lot of RAM and to complete.

To create a schedule for your workload, just run the following command:

```
./schedule --config config/running.xml --algo greedy
```
If your configuration also contains sizes you can use the \texttt{--size} option to utilize them also for the execution of the algorithm.

File:
- 954schedule-master.zip

---

The information available on this website is property of the contributing authors from the mPlane Consortium (project FP7-ICT-312927) and does not necessarily reflect the view of the European Commission. The information on this website is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.
This is an illustrative set of Spark jobs, implemented in Scala, that can be run on a HDFS repository storing JSON files. The main goal of these jobs is to compute statistical values (e.g., mean, median, std) and to perform basic analysis on raw data coming from browsing session recorded by the Firelog probe.

**Job 1:** Calculate mean, median, standard deviation, top5 of the Page Load Time in all stored sessions.

**Job 2:** Calculate the correlation between the Page Load Time and other properties (e.g., page size, http times, and so on)

**Job 3:** Find the most contacted servers, excluding the DNS resolved one, for all stored sessions.

**Job 4:** For all probes, find the longest common path in traceroutes towards the same destination.

For each job, a capability on the repository component has to be registered. For example, for Job 1:

```
| mplane | runcap page-load-time-stats
| when | = now + 5m
hdfs.root = /path/to/root/json/folder/ ok
```

**File:**

```
| Scala : Spark jobs
```

---

The information available on this website is property of the contributing authors from the mPlane Consortium (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission. The information in this website is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.
References
