Data TV

Author:
Martí Bosch Padrós

Supervisor:
Xavier Espinal Curull

September 11th 2015, Meyrin, Switzerland

Contents

1 Introduction

This report describes the work carried out during a CERN Summer Student project named DataTV, whose objective is to create a web application that displays the real time physics raw data throughputs from CERN's experiments to the CERN storage facilites (Castor and EOS). For such data flows, the integrated values for a given period of time (the last 24 hours, the last week, and the LHC run 2). On the other hand, the amounts of data that are migrated from disk to tape (in CASTOR) for long term storage, are also shown in the plot.

The project can be split in two main parts, which are (1) the deduction of the data that is being transferred from the experiment to the storage facilities, and (2) its display in a web browser. The first part is covered in Section 2 whereas Section 3 deals with the second part. A complementary related work that has also been done as part of the project is the collection of Xrootd formatted logs in the CASTOR Cockpit, which is detailed in Section 2.4.

I would like to thank my supervisor Xavier Espinal for considering me an appropiate person to initiate the development of the project, as well as the rest of the IT-DSS team for their support. On a more personal note, I must also thank the rest of the CERN Summer Students that I have been lucky to share time with.

2 Collecting Data from the Storage Facilities

In the context of this project, two storage facilities are considered: CASTOR and EOS. CASTOR is a Hierarchical Storage Management System born in 1999 that handles both disk and tape layers. On the other hand, EOS started in 2011 and is focused on analysis and fast data processing with very low access latency. By October 2013, CASTOR and EOS holded 92PB and 20PB of data which were respectively distributed in 350M and 158M files respectively [2].

2.1 Global Structure

The different experiments that are carried out at CERN perform their write operations happen in an *asynchronous* way, but for the purposes of this project, the updates in the web server will be forced to happen *synchronously*, with a *period* that will be noted as ΔT_U, and will be of 1 minute. This means that in both CASTOR and EOS Cockpits, a UNIX `crontab`[1] is programmed to send the messages to DataTV every ΔT_U. The described structure is represented in Figure 1.

The messaging between the Cockpits and DataTV is implemented via the messaging library ØMQ[2], and for both CASTOR and EOS Cockpits the messages will be formatted in JSON as in Object 1.

[1] See `http://www.gnu.org/software/crontab/manual/crontab.html` for more details
[2] See `http://zeromq.org/` for the library's web site

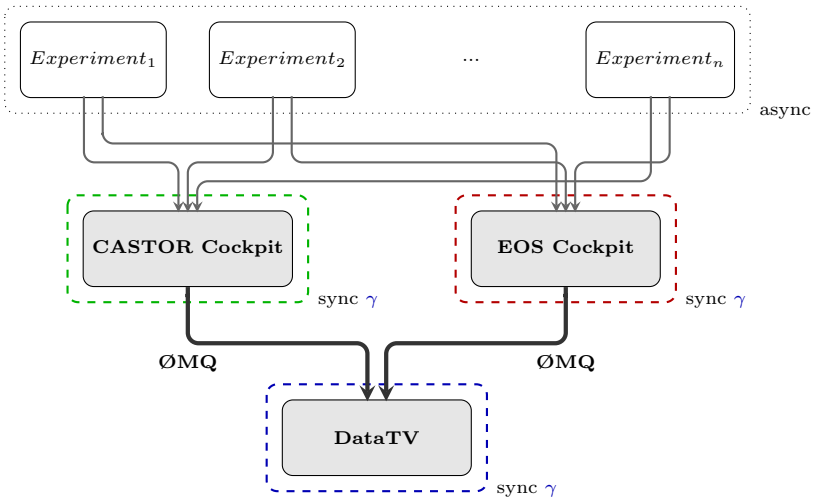

Figure 1: Structure of the messaging between machines

```
{ storage: storage,
  flows: { experiment_i: v_i, experiment_j: v_j ... },
  timestamp: t_msg }
```

Object 1: Message received from the cockpit

2.2 CASTOR Data

The determination of the amounts of data that the different experiments have persisted in CASTOR is not implemented by the time of this report. Such implementation will consist in defining *monitoring metrics* in the CASTOR Cockpit aimed to extract the amount of bytes that the different users (that correspond each to a given experiment), and then using the *Command-Line Interface* implemented in the Python library `rpyc` in a *sequential* script that will build a JSON message of the form of Object 1 that will be sent to DataTV every ΔT_U.

2.3 EOS Data

For the EOS storage facility, the amount of bytes transferred by each experiment can be extracted from the different `iostat` files located in `/scratch/eosquota/`. This task is performed by a *sequential* script named `eos_send_flows.py`, which uses the UNIX command `grep`[3] to extract the written bytes of every experiment. With the output of the `grep` and some string processing a JSON message of the form shown in Object 1 is built and sent to DataTV as explained in Section 2.1.

[3]See `http://www.gnu.org/software/grep/manual/grep.html` for the command's manual

2.4 Implementation of Xrootd Log Collection

The throughputs of the experiments to CASTOR are collected in the Xrootd logs which *were not sent* to the CASTOR Cockpit before the project started. As part of the project's work, the collection of Xrootd logs on CASTOR Cockpit was *developed*, which consisted in the implementation of three new plugins for the Simple Log Producer:

1. `castor_xrootdfile_voidheader_parser.py` discards *log lines* with no information (those that do not contain the key: value `level=INFO`
2. `castor_xrootdfile_header_parser.py` parses the *line header* (highlighted in blue in Listing 2)
3. `castor_xrootdfile_body_parser.py` decomposes and parses the `opaque` *field*. For the purposes of DataTV the `opaque` subfields `oss.asize` and `castor.sfn` (highlighted in green in Listing 2) permitted the identification of certain experiments' throughput to CASTOR

Listing 2: Example of an Xrootd log line

```
150904 10:47:06 time=1441356426.671963 func=open level=INFO ... tident=
alicedaq.47974:1154@aldaqtdsm03 path=/castor/cern.ch/alice/raw/ad
/2015/09/03/19/15000234925004.107.root, opaque=mode=644&oss.asize=37720243&
castor.sfn=/castor/cern.ch/alice/raw/ad/2015/09/03/19/15000234925004.107.root
&castor.pfn1=/srv/castor/04/18/1455174018@castorns.21033176386&castor.id=
alicedaq.47974:214@aldaqtdsm03&castor.client_sec_uid=13798&castor.
client_sec_gid=1395&castor.accessop=3&castor.exptime=1441356486&castor.
manager=c2alicesrv201.cern.ch:1094&castor.txtype=user&, isRW=1, open_mode
=4102, file_ptr=0x7f1070010910
```

3 The Web Server: DataTV

3.1 Storage and Integration of Values

In order to keep track of the amounts of data that have been transferred during a previous given ΔT which could be the last 24 hours or the last week, DataTV stores on MongoDB[4] the list of all the flows and its information. Each list's item is a JSON-like *flow object* as in Object 3.

[4]Project's web site: `https://www.mongodb.org/`

```
flow = { source: experiment,
         dest: storage,
         flowData: {
           values: [v_t, ... ,v_{t-week}],
           last24h: S_{24h},
           lastWeek: S_{week},
           run2: S_{run2} }
       }
```

Object 3: Flow object stored in the MongoDB

The fields `last24h`, `lastWeek` and `run2` store an integer value each, that corresponds to the number of bytes sent during the fields' respective period of time. On the other hand, `values` stores the number of bytes transferred during each one of the previous ΔT_U: v_t corresponds to the number of bytes transferred during the last ΔT_U, whereas v_{t-week} represents the amount transfered in the ΔT_U that occurred one week before the last update.

For the purposes of this project, `values` stores only the amounts of bytes transferred during the ΔT_U that correspond to the last week time frame. Nevertheless, this can be generalized if `values` is considered as a *FIFO circular buffer*[5], which is a data structure behaving like a *FIFO queue* with a fixed size.

Every ΔT_U DataTV *receives a message* with the amounts of bytes v_t transferred during that last ΔT_U. When that happens, for *every path* the persisted flow object must be updated. Let `UPDATES_24H` and `UPDATES_WEEK` be the number of ΔT_U in 24 hours and one week respectively, and `vLast` $= v_t$. This is summarized in Procedure 1.

Procedure 1: updateFlow(v'_t, `flow`)

```
1 update in DB flow.flowData as in
2 |  values.enqueue(v'_t);
3 |  last24h += v'_t - values[UPDATES_24H - 1];
4 |  lastWeek += v'_t - values[UPDATES_WEEK - 1];
5 |  values.dequeue();
```

Note that after the update, the database will have the array `values = [`v'_t`,` v_t`, ... , `$v_{t-week-\Delta T_U}$`]`, and so its length is still `UPDATES_WEEK`.

3.2 Enforcing Data Consistency

The structure described in Section 2.1 ensures that if no machine goes down, $\forall\ \{t_0, t_1\} \mid t_1 = t_0 + \Delta T_U$ the server DataTV will receive *one and only one* update message from *both* EOS and CASTOR Cockpits.

[5] See `https://en.wikipedia.org/wiki/Circular_buffer` for more inofrmation

However, it is possible that in certain ΔT_U some flows do not appear in the *messages*. Consider again how the values are integrated (as explained in Section 3.1). Since every position of `values` corresponds to a number of bytes transferred during the last ΔT_U, to ensure that values are integrated properly for the periods *last minute*, *last 24 hours* and *last week*, the flow must be updated anyway for this ΔT_U but with a value of 0 (which corresponds to calling `updateData` from Procedure 1 with `vLast = 0`).

To make sure that this happens, DataTV stores a list of *all the data flows* that are tracked in MongoDB. Then, when the two messages from the last ΔT_U are available, Procedure 2 updates *each* flow with either 0 or the value found in the corresponding message. This way the data displayed is guaranteed to be consistent with its time frame (last 24 hours, last week, LHC run2).

Procedure 2: updateFromMsgs(`castorMsg`, `eosMsg`)

```
1  foreach flow ∈ flows do
2  |  foreach message ∈ castorMsg, eosMsg do
3  |  |  v'_t = 0;
4  |  |  if message has key flow.source then
5  |  |  |  lastFlows[flow.source][msg.storage] = msg[flow.source];
6  |  |  end
7  |  |  updateFlow(v'_t, flow);
8  |  end
9  end
10 updateDBTimeStamp();
```

3.3 Pull Socket

The *pull socket* is responsible of collecting in the DataTV server the messages received from both CASTOR and EOS cockpits as shown in Figure 1. The behaviour of the *pull socket* can be represented with an *Extended Finite-State Machine (EFSM)*, with the notation `trigger [guard] / action` to depict the transitions between states. The `trigger` *event* starts the transition, but in *EFSM*s the transition will only occur *iff* the condition expressed by the `guard` evaluates to `true`. Under such circumstances, the corresponding `action` is executed, and the machine moves to the next state thus completing the transition. The behaviour of the pull socket is then represented as an *EFSM* in Figure **??**.

DataTV will start from the state *init* and start a timer which will trigger the event *timeout* after ΔT_U. The event triggers *castorMsg* and *eosMsg* correspond to the reception of a message from CASTOR and EOS respectively, and the *update* corresponds to the action of updating the database according to the information received in the messages.

Nevertheless, the *coherency* mecanism described in Section 3.2 will only work with the automaton of Figure 2 under the assumption that all the message passing shown in Figure 1 works properly, which means that CASTOR and EOS

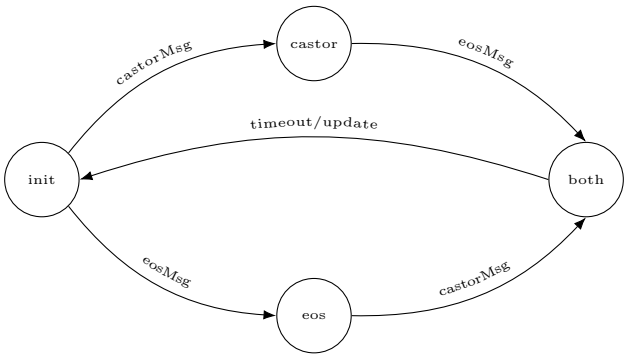

Figure 2: *EFSM* of the *pull socket*

cockpits *both* send *one and only one* message every ΔT_U. This ideal scenario can be altered as two problems might arise in Figure 1:

1. One of the cockpits might go down and *not* be able to *deliver* one or more messages.

2. DataTV might go down and *not* be able to *receive* one or more messages.

Then a protocol must be defined in order to still ensure the *data consistency* under either of the two listed possible situations. Consider Figure 3 as the extension of the *pull socket* automaton from Figure 2.

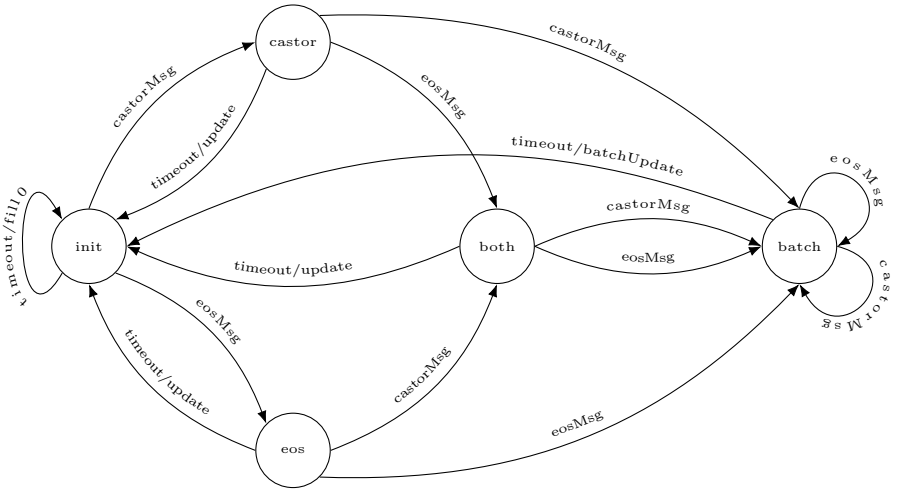

Figure 3: *EFSM* of the *pull socket* with the *coherence protocol*

In Figure 3, DataTV starts from the state *init*, and if no message is received at the end of the timer, a self-transition will occur and the action *fill0* will be

executed. Such action will fill *every* flow from the database with *zero values* for the data flows corresponding to that particular ΔT_U.

When the first message is received, the automaton moves to the corresponding state (*castor* or *eos*) without executing any action. Under these circumstances, if no other message is received when *timeout* is triggered, the machine returns to the *init* state the action *update* is executed with the data of only *one* message. On the other hand, if the message form the other Cockpit is received, the system moves to the state *both*, that will wait for the *timeout* trigger, and when it arrives *update* the database with the content of *both* messages.

If DataTV is working properly, as stated before, it will never receive two messages from the same machine in the same ΔT_U. However if the messages are sent from a Cockpit and DataTV is not able to receive them, they will be stored in a *message queue* and sent to DataTV as soon as it is up and running again. In this case, DataTV might receive a *batch* of messages from the *message queue* in only one ΔT_U. When that happens, the state *batch* is reached. In such state, several messages are received and stored in an ordenate way according to their *timestamps*. Then when *timeout* is triggered, the action *batchUpdate* will insert in the database all the values that correspond to the period where DataTV was not able to receive the messages.

3.4 Visual Layout

To display the active data flows in CERN, a *weighted directed graph* structure is plotted. Two kinds of *nodes* can be distinguished: *pools* (experiments) and *destination* (storages). Such *nodes* are plotted over a figure of the CERN accelerator complex[6] according to the actual physical location of their infrastructure. The *paths* between the *pools* and the *destinations* are plotted in a blue-orange *color scale*[7]. In addition to the *color scale*, bullets that follow the *path* might be plotted as well, with a speed that is adjusted to the flows' magnitude.

Such plot is created using Scalable Vector Graphics (SVG) [1], that are dynamically modified by JavaScript using the D3.js[8] library. To dynamically modify elements of the user interface, the JavaScript library JQuery[9] is used.

3.5 Front-End

The interaction between the plot described in Section 3.4, the user events and the server events is plotted in an *EFSM* in Figure 4.

In this project, the *automaton* is composed of 4 *states* (excluding the initial one): L_{min}, L_{24h}, L_{week}, Run_2 which show the number of bytes transferred

[6]See `https://commons.wikimedia.org/wiki/File:Cern-accelerator-complex.svg` for the source of the file

[7]The blue-orange color scale are more suitable for people with color blindness diseases (deuteranopia, protanopia and tritanopia) than traditional red-blue color scale

[8]Library's main site `http://d3js.org/`

[9]Library's main site `https://jquery.com/`

during 4 different time frames respectively: *last minute*, *last 24 hours*, *last week* and *LHC season 2*, which corresponds to a LCH second three-year run[10].

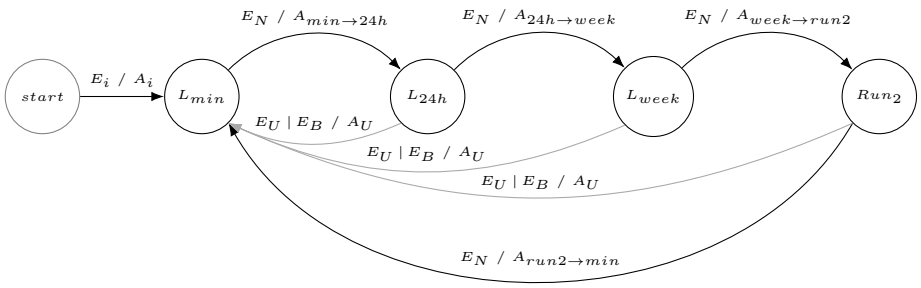

Figure 4: *Extended Finite State Machine* representation of the Web Site

A representation of such *automaton* is shown in Figure 4. The *events* that might trigger transitions are:

- E_i is the initial user request to the server
- E_N is the event that the server sends repeatedly every certain interval ΔT to indicate that the *automaton* should move to the next state
- E_U is triggered every ΔT_U to automatically retrieve the latest data from the database
- E_B represents the user's click to the *Update Button*

On the other hand, the different *actions* that are associated to the *events* share a lot of common funcitonalites. Lets then define the following functions:

- $f_{labels}(S_i, S_f)$ hides the *text labels* corresponding to the state S_i and shows those of S_f
- $f_{colors}(S_f)$ changes the *color* of the paths (in the blue-orange scale) according to their values in the state S_f
- f_{balls} toggles the visibility of the balls that follow the paths

Then the *actions* from Figure 4 can be described as:

- A_i is the initial action that consists in (1) generating *all* the SVG elements according to the data provided by the server, and (2) toggling the visibility of the SVG elements so the data of the state L_{min} is displayed
- $A_{min \to 24h}$ calls $f_{labels}(L_{min}, L_{24h})$, $f_{colors}(L_{24h})$, and f_{balls} (to *hide* the balls)

[10] See `https://run2-13tev.web.cern.ch/` for more details

- $A_{24h \to week}$ calls $f_{labels}(L_{24h}, L_{week})$, and $f_{colors}(L_{week})$
- $A_{week \to run2}$ calls $f_{labels}(L_{week}, L_{run2})$, and $f_{colors}(L_{run2})$
- $A_{run2 \to min}$ calls $f_{labels}(L_{run2}, L_{min})$, $f_{colors}(L_{min})$, and f_{balls} (to *show* the balls)
- A_U is performed when new data comes from the server, so according with the new data and for *every state*, it (1) replaces the *text labels*, (2) rescales the paths' *colors*, as well as (3) readjusts the speed of the paths' *balls*

4 Perspectives and Future Work

The DataTV project has been started from scratch as a Summer Student project during 11 weeks the summer of 2015. By the end of the project, there are three main missing features:

1. The extraction of of data throughputs stored in CASTOR Disk is not implemented yet, nor the extraction of the amounts of data that are migrated from Disk to Tape for long-term storage
2. The ΔT_U is of 300 seconds, but shall be dropped to 60 seconds as the `iostat` files of EOS Cockpit can already support that
3. The *batch mode* of the automaton of Figure 3 is not implemented yet, although the high level idea is already detailed in Section 3.3

On the other hand, the User Interface can be improved in many ways. This is rather subjective, nevertheless some ideas are listed here:

- Create total counters that sum the throughput of *all* the experiments
- Display a *status bar* that indicates when the displayed data is updated (every ΔT_U)
- Refine the color scaling to show better looking flow colors in the plot according to the nature of the experiments' amounts of written bytes

References

[1] Erik Dahlström and ed. Scalable Vector Graphics (SVG) 1.1 Specification (Second Edition), 2011.

[2] X Espinal, G Adde, B Chan, J Iven, G Lo Presti, M Lamanna, L Mascetti, A Pace, A Peters, S Ponce, and E Sindrilaru. Disk storage at cern: Handling lhc data and beyond. *Journal of Physics: Conference Series*, 513(4):042017, 2014.