Design and Implementation of a Dynamic Extension of the ATLAS Run Query Service

Alexandru Buliga, Polytechnic University of Bucharest
Supervisors: Jörg Stelzer and Andreas Hoecker

July-August 2015
Abstract

The ATLAS RunQuery is a primarily web-based service for the ATLAS community to access meta information about the data taking in a concise format. In order to provide a better user experience, the service was moved to use a new technology, involving concepts such as: Web Sockets, on demand data, client-side scripting, memory caching and parallelizing execution.
Chapter 1

Introduction

The purpose of this document is to offer the people working on Atlas Run Query a clear understanding of the way the service works. Since the last version, a series of changes has occurred. Here are the most relevant ones:

- introducing the **WebSocket protocol**\(^1\) through **Python’s Tornado**\(^2\) framework in order to implement long living duplex connections between the client and the server; this implies that both the client and the server can initiate data transmission;

- a **self-made message passing system** based on long bidirectional communication between the server and the browser;

- the implementation of the popular mechanism of **on demand data serving**; this is largely used within Atlas Run Query when it comes to rendering essential information first as soon as it is available;

- parallelizing independent tasks using **thread pools and processes**;

- caching information using the **Redis data structure database**\(^3\), and the client it provides in Python, **Redis py**\(^4\).

In addition to this, this text serves another goal, at least equally important: to help people easily build new features on top of the existing structure and improve different modules of the project.

---


\(^3\) [http://redis.io/](http://redis.io/)

Chapter 2

Client - server communication

2.1 Why Web Sockets?

WebSocket is a protocol providing a low-latency, bi-directional, full-duplex and long-running connection between a browser and server. This is a very strong sentence that comprises the main big advantages this protocol has over existing communication techniques on the web.

2.2 Tornado mechanism explained

Tornado\(^1\) is a Python framework that offers support for server-side Web Sockets communication. In figure 2.1, I have outlined the structure it follows and the handlers it provides:

- **HTTP Server**: takes a port number and an application context and a number of server instances (fig: HTTP Process 1, 2, ...) equal to the number of cores the machine has; each of those instances uses a specific class to handle the request they serve (fig: Handler Instance 1, 2, ...);

- **ExecQueryHandler** is a class that implements the method required by the WebSocket protocol (onopen, onmessage, etc.); more details in 2.3 The message passing system section;

- **Application context**: this objects has specific handlers and their role is to tell the HTTP Server how to interpret different type of connections; for instance:
  - a request at http:HOME-URL/index is always expected to be an HTTP request because the class handler bound to this URL, IndexPageHandler, serves an HTML page;
  - on the other hand, requests at http:HOME-URL/execQuery have to implement the WebSocket protocol given the fact the class bound to this URL, ExecQueryHandler, implements the WebSocket protocol (through the already well known functions);

- **Tornado IO event loop**: This specific tornado object, ioloop, is a mediator between the incoming web browser connections and the HTTP server that processes them; it offers a fairly simple interface (start and stop methods) and runs on a dedicated thread, which particularly fast.

\(^{1}\text{http://tornado.readthedocs.org/en/latest/websocket.html}\)
- **Web browser**: communicates with the server through the WebSocket interface; the client-side of the WebSocket protocol is implemented in **JavaScript**.

Figure 2.1: Tornado Overview
2.3 The message passing system

As mentioned before, the interface the WebSocket protocol provides is suitable for direct message passing between server and client. Still, we need a way to distinguish between different message types. The idea behind the protocol I came up with is uncomplicated:

- firstly, I created an encapsulation of the message object that goes down the wire; it is a stringified JSON object containing 2 attributes: type (an integer suggesting the type of the message) and data (the actual data, which can be pretty much anything); the data field can be anything because knowing the type field of the message, we are aware of what to expect in the data field;

- then, I made sure the types of the message were consistent between client and server side; the trivial way to guarantee this was to keep 2 copies of the same dictionary with message types, one on the client, the other on the server;

- if one wants to extend the functionality of this interface has to follow this steps:
  1. decide upon a name and a value for the new message type and add them the both copies of the MsgType dictionary;
  2. on server-side: call the function write_message from file arq_server.py with the correct parameters (type and data);
  3. on server-side: go to file /server/html/new_js/atlas-runquery-main.js, function ARQ_MAIN.updateUI add to the switch/case your new message type as a case and process the data by updating the UI components.

Note: the dictionary is called MsgType and I left instructions on where to find it
Chapter 3

How the new Atlas Run Query works

This section’s aim is to offer a general idea of how the new Atlas Run Query actually works. We know by now that there is a long-running client-server connection, a simple message passing communication system and that both the client and the server can initiate messages. We also known there are special handlers on the server side for the type of connection it receives.

Now I am going to explain a ‘start to finish’ scenario which is exemplified by figure 3.1; those are the stages:

1. visit the Atlas Run Query homepage: this triggers a HTTP web request that serves the index page via `IndexPageHandler`\(^1\); it also creates a JavaScript WebSocket object that opens the connection through the handshake protocol;

2. perform a query search: triggers the function that sends a message to the server; this message is encapsulated in the well known object with the 2 keys (type and data), encoded, and sent to the server;

3. the server-side handler for WebSocket connections, `ExecQueryHandler`\(^2\), implements the protocol specific method `on_message`; this is the method that receives the data sent from the browser;

4. following the structure of the `ExecQueryHandler` in figure 3.1, we can observe it controls a thread pool; `on_message` only submits a job to this pool, which will be executed by the `process_message` function in the context of a new thread; this ensures us the tornado IO event loop is not blocked and can ‘listen’ uninterruptedly to new requests;

5. in the context of the new thread (run by `process_message`) I spawned a new custom process, which actually calls all the functions of the old version of Atlas Run Query up to a point (discussed bellow); creating a worker process (subprocess) is necessary because of the code structure of the old project: it needs to be run from a new process context (at every new WebSocket message), otherwise data will get corrupted (there are several class objects that remain mutated between separate runs of the code, which is not the expected behavior).

6. the main process and the subprocess communicate via a special queue, that encapsulates as a result an object similar to the task received from the browser (one dictionary with 2 attributes: type and data);

7. after the task is sent to the worker process, the main process blocks into a while loop, waiting for results in its queue;

\(^1\) `IndexPageHandler` - implemented in main server script `arq_server.py`
\(^2\) `ExecQueryHandler` - implemented in main server script `arq_server.py`
8. at this point, all the operations in the original project are performed: parsing the query input, fetching data from the databases, filtering data, etc; the difference comes from the fact that, once we have a piece of information that can be sent to the browser to be rendered, all we need to do is to put the intermediate results in the queue; by doing that, the main process gets the result and sends it to the browser via write_message function, also implemented in ExecQueryHandler;

9. the browser received the info in its own on_message method and renders the info according to the message type it receives;

10. meanwhile, the server continues the execution until it gets all the information it needs in order to put up the final results;

11. another significant difference resides in the way the actual HTML table containing the run data is created; because at this point, assembling HTML data for each run is an independent step, everything can be parallelized; I achieved this by creating a pool of processes with a limit in the number of processes and submitting jobs to this pool;

12. each job gets a run and builds the HTML data it needs; all the remaining database requests are now parallelized, which translates into a considerable boost in performance; once again, the information is sent to the browser as soon as it is available; an essential aspect of this separation is that building data that usually takes longer (plots) and is not necessary right-away (tool tips) can be delayed in favor of the paramount information.
Figure 3.1: ARQ Demo
Chapter 4

Future work

4.1 Caching for results table cells

Some of the data Atlas Run Query is building throughout its complex processes can be cached. One idea for the people that are going to work on the project is to cache the whole HTML string at the contained by one cell in each of the results table rows (corresponding to a run). In this way, the cache can be useful even between queries retrieving different data for the same run: at least the data that overlaps between those queries can be obtained from the cache.

4.2 Ongoing run bug?

I have implemented a feature allowing the web page to update the ongoing run automatically every five minutes if it is present in the initial query. The way it works is by setting a JavaScript timeout function that performs an identical query (in terms of what info to show) but only for the last run (find run last 1 / show ...).

Unfortunately, I had to disable the feature because of this unexpected behavior: querying for a list of runs with a specific set of data in the /show clause returns a table of results. Depending on the run, some of the data the user requests is not available, which caused the specific cell to be marked with a n.a. symbol.

However, if we perform the same query (with the same /show clause) only on one run and the results do not contain some of the data, we do not get an empty n.a. cell anymore; the cell is just not added to the table header because there is no other run that has this information. It is my belief this is unexpected behavior as the user might prefer to know what data is not available instead of finding out by exclusion.
4.3 Jquery UI Autocomplete using Tries

I have added a special jQuery UI component (input text with drop-down list) that imitates Google Search behavior: suggesting lists of possible queries in a drop-down element. The purpose was to improve the overall user experience.

Nevertheless, I did not have enough time to make it smarter. Right now, it gets all the suggestions from a static list of possible queries defined in /server/html/autocomplete-query-choices.js by matching the queries starting with the value inside the input text.

The aim is to create a list of rules on how the queries need to be created. I would see those rules as objects specifying a pair formed by a key-word and a list of all the words that can follow the key-word. Also, a good algorithm needs a Trie data structure in order to perform the optimal string matching. Here are the steps one needs to implement for the query matching algorithm and creation of the suggestions list:

1. get the current input of the user;
2. break it into a list of words;
3. get the first word (let’s call it prefix) and use the trie (it needs to implement a function that returns a list of words that begin with the prefix) to get a list of the matches from the list of key-words in the rules object;
4. then create a partial query suggestions list; its elements start with the words in the list of matches;
5. get the second word of the query and match it against the list of words created from appending all the lists representing words that can follow the key-words from the previous steps;
6. update the partial query suggestions list with the results accordingly;
7. continue until there are no more words in the input.

This algorithm has to be called each time the input text value changes. In order to avoid thrashing I set up a delay for 200ms before recalling the matching function.