SP-586 Refactor Taranta to improve performance

Taranta front-end performance report

Hélder Ribeiro, Ajaykumar Dubey, Matteo Canzari, Valentina Alberti

Introduction

The work presented in this document has been performed with the goal of improving performances of running Taranta dashboards as indicated in the feature: “Refactor Taranta to improve performance” [SP-586]. The results of the benchmarking conducted in [SP-296] showed that the frontend is mostly responsible for slowing down the performances of the tool. In particular, what appeared to be critical was the part of the code that dispatches the information coming from the backend to the frontend: process which was sequential. For each frame of information, the following chain of functions is executed in subsequent steps. This causes long latencies given that a new frame can be handled only after all the steps have been completed.

Frame handling sequence of functions: handleNewFrame() -> recordAttribute() -> recordHistory() -> setNewValues() -> setState() -> render()

This observation is at the basis of the first definition of [SP-586]. While trying to solve the problem by introducing asynchronous code, performance tests revealed that the actual blocker wasn't the dispatcher (i.e. recordAttribute step) as previously thought but rather the rendering part. For this reason, in agreement with the PO and FO the team decided to focus on benchmarking the graphical library used by Taranta, plotly, and see if it is possible to improve its performance or find an alternative library to use.

The revised acceptance criteria for the feature are as follows:

- a report is produced describing the current bottleneck hypothesis and how it was confirmed, with the list of affected widgets and widgets not affected by it
- a description (maybe a section of the above report) describing which plotly part is responsible for the bottleneck
- reimplementation of at least 1 widget so that the problem disappears; the new implementation should maintain the current functionality, configurability, look and feel of the widget to minimize changes to existing dashboards.

We would like to stress the importance of distinguishing between the following 3 areas of the code which are responsible for performances at the different stages of Taranta use:

1. **Edit** dashboard section
    - This section of code relates to the user experience while editing an existing dashboard, by dragging, copying, adding new widgets and so on.
2. **Loading** dashboard section
    - This section of code relates to the time it takes from a user clicking on the start button on a dashboard till the moment in which the dashboard is fully available and displays the desired data.
3. **Refreshing** dashboard section

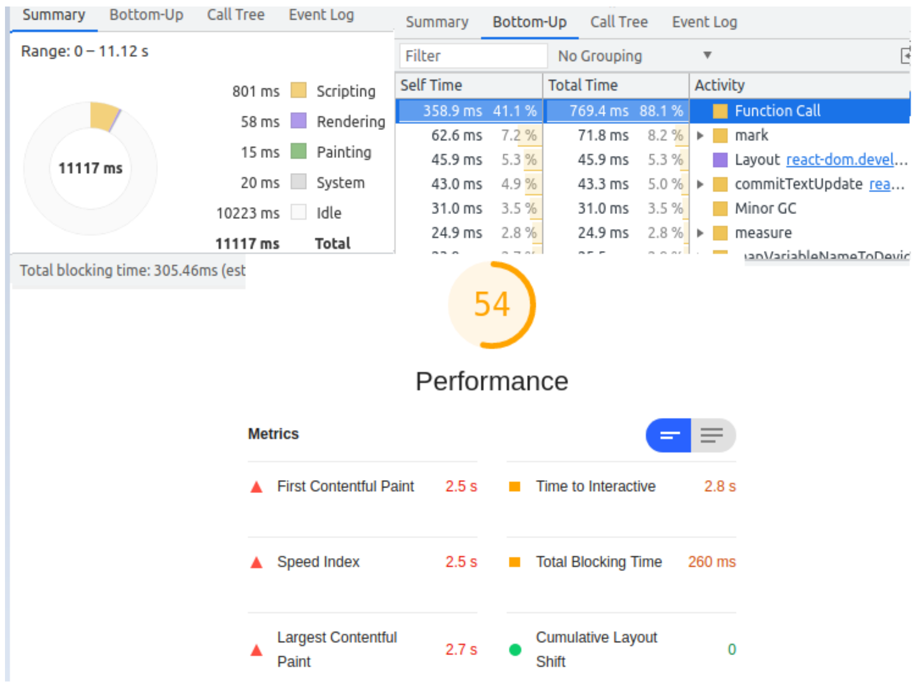
- This section of code relates to the time it takes for a fully working/started dashboard to update it's values, for example if we consider a dashboard with 25 widgets, all displaying a single attribute of value 10, the refreshing time is the time it takes to update their value to 20 after it changed in the backend.

The results of the study performed within the scope of SP-586 and summarised in this document mainly refers to the last point, Refreshing dashboard.

Taranta performance analysis

Multithreading as a way to improve Taranta performances

The initial thought was that the sequential behaviour of Taranta dispatcher caused most of the performance issues seen while using the tool. Multithreading is the alternative that has been investigated to speed up the process. The best approach we could find in literature is to use Web Workers. We implemented a prototype and performed a benchmark to verify and quantify the improvement with respect to the current implementation of the code.
Performance at runtime has been evaluated using Chrome Developer tools. They allow, among other functionalities, the retrieval of both a summary that highlights the time spent by the dashboard in performing different tasks during the duration of the test and of a more extensive report via the Lighthouse tool [Lighthouse]. The summary highlights the time spent by the running dashboard while performing different tasks such as running scripts or rendering. An example of this summary can be seen in Figure 1. On the other hand, Lighthouse calculates the performance based on a set of metrics that have been identified to take into account how the user perceives performance when using the web. This is a set of six metrics that includes for example the time taken to load the page's main content (Largest Contentful Paint, LCP) or the time taken to render the first piece of DOM content after a user navigates to a certain page (see [Lighthouse] for more details on the metrics and how they are weighted).



Range: 0 – 11.12 s

801 ms	🟨	Scripting
58 ms	🟪	Rendering
15 ms	🟩	Painting
20 ms	⬜	System
10223 ms	⬜	Idle
11117 ms		Total

Total blocking time: 305.46ms (est

Filter | No Grouping

Self Time		Total Time		Activity
358.9 ms	41.1 %	769.4 ms	88.1 %	Function Call
62.6 ms	7.2 %	71.8 ms	8.2 %	mark
45.9 ms	5.3 %	45.9 ms	5.3 %	Layout react-dom.devel…
43.0 ms	4.9 %	43.3 ms	5.0 %	commitTextUpdate rea…
31.0 ms	3.5 %	31.0 ms	3.5 %	Minor GC
24.9 ms	2.8 %	24.9 ms	2.8 %	measure

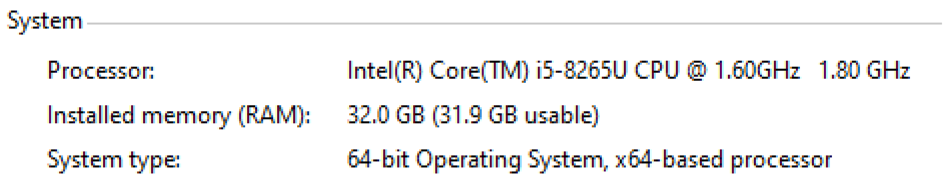
▲	First Contentful Paint	2.5 s	■	Time to Interactive	2.8 s
▲	Speed Index	2.5 s	■	Total Blocking Time	260 ms
▲	Largest Contentful Paint	2.7 s	●	Cumulative Layout Shift	0

Figure 1: Developer tool output example. Lighthouse metrics results are displayed at the bottom.

The benchmark has been performed as follows:

- We used dashboards with an increasing number of widgets. Each dashboard had multiple copies of the same widget;
- We investigated both widgets that require plotly and some that don't;
- We analysed the output of the used developer tools to assess the impact of implementing Web Workers.

The conclusion is that implementing Web workers doesn't significantly improve performances for any of the benchmarked widgets [CT-341]. The lack of multithreading isn't the current bottleneck for Taranta, and rather we realized the rendering is. The study also shows that most of the time spent in rendering is consumed by the Recalculate style functionality of the plotly library and that increasing the number of plots increases the rendering time.



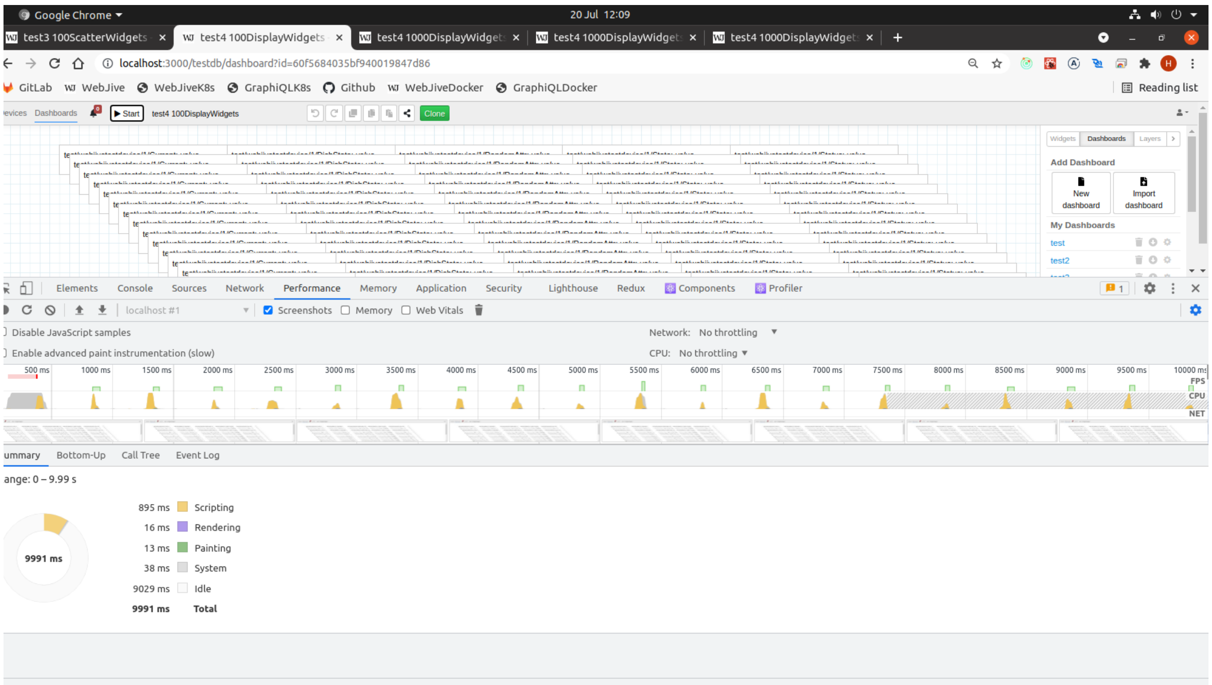
System

Processor:	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
Installed memory (RAM):	32.0 GB (31.9 GB usable)
System type:	64-bit Operating System, x64-based processor

Figure 2: Configuration of the computer used for the benchmarking of Web workers implementation.

Improving the plotly library

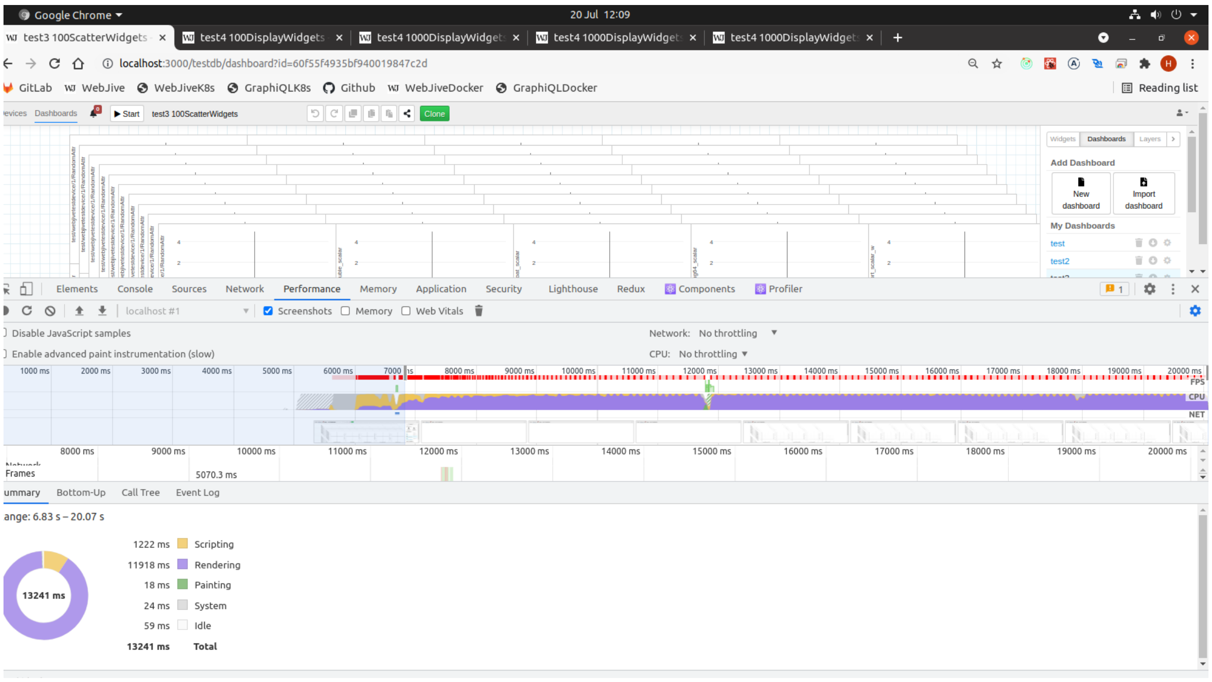
Plotly is the library used by Taranta to create and render graphical widgets such as the Timeline widget or the Scatter plot widget. It is a quite complete library, built on top of D3.js, that allows for some customization and functionalities such as zooming and panning. Despite its usefulness, the performed benchmark identified it as the current bottleneck responsible for reducing performances or running Taranta dashboards. To highlight this, we analysed the results obtained from the Chrome Developer tools in the case of dashboards with 100 or 1000 non graphical widgets and dashboards with 100 plots on it. We iterated on the graphical widgets to cover them all. When graphical widgets are involved, performances degrade significantly and rendering is the step that requires the longest time to be completed, see for example Figure 3 and Figure 4.



Range: 0 – 9.99 s

895 ms	🟨	Scripting
16 ms	🟪	Rendering
13 ms	🟩	Painting
38 ms	⬜	System
9029 ms	⬜	Idle
9991 ms		**Total**

Figure 3: Performance test results for a dashboard with 100 Attribute Display widgets (doesn't use plotly). The idle time value indicates the time during which no processing is happening and the browser is ready to update widgets with new data coming from the backend.



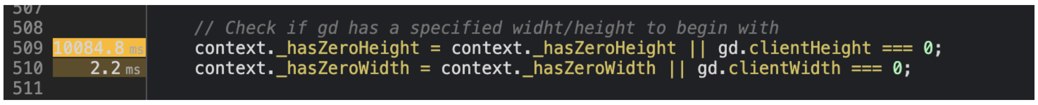
Range: 6.83 s – 20.07 s

1222 ms	🟨	Scripting
11918 ms	🟪	Rendering
18 ms	🟩	Painting
24 ms	⬜	System
59 ms	⬜	Idle
13241 ms		**Total**

Figure 4: Performance test for a dashboard containing 100 Scatter plot widgets (uses plotly). Rendering takes 90% of the total duration of the test (identified by the Total label) while scripting takes less than 10%. As a result, the idle time is very short and updating widgets with new data takes longer.

Functions in the plotly library that are mainly responsible for slowing down widgets have been identified by benchmarking dashboards with 10 and 50 Attribute Plot widgets. The identified lines of code are:

- `/node_modules/plotly.js/src/plot_api/plot_api.js`. This is the function which checks if initial width/height are set and slows down the “Recalculate Style” activity, see below.

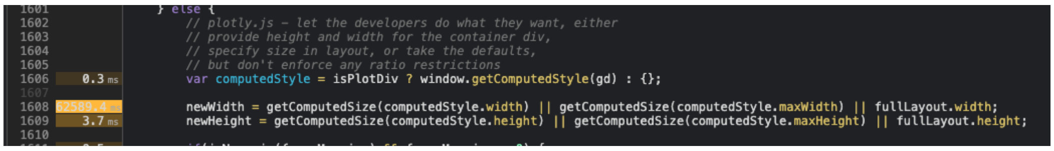


```
508                   // Check if gd has a specified widht/height to begin with
509  10084.8 ms       context._hasZeroHeight = context._hasZeroHeight || gd.clientHeight === 0;
510      2.2 ms       context._hasZeroWidth = context._hasZeroWidth || gd.clientWidth === 0;
511
```

  Moreover, increasing the number of widgets on the dashboard from 10 to 50 the benchmark shows that performances don’t scale linearly, see below table:

Number of widgets	% of time spent in scripting	% of time spent in rendering
10 Attribute plots	19%	42%
50 Attribute plots	10%	87%

- `plotly.js/src/plots/plots.js`



```
1601                } else {
1602                    // plotly.js - let the developers do what they want, either
1603                    // provide height and width for the container div,
1604                    // specify size in layout, or take the defaults,
1605                    // but don't enforce any ratio restrictions
1606      0.3 ms        var computedStyle = isPlotDiv ? window.getComputedStyle(gd) : {};
1607
1608  62589.4 ms        newWidth = getComputedSize(computedStyle.width) || getComputedSize(computedStyle.maxWidth) || fullLayout.width;
1609      3.7 ms        newHeight = getComputedSize(computedStyle.height) || getComputedSize(computedStyle.maxHeight) || fullLayout.height;
1610
```

This information was useful to identify the next steps in tackling the issue.
Three different strategies have been tried, namely:

1. To assign fix values to width and height in the identified lines of the code [CT-398];
2. To test performances of other libraries to decide whether to use a different one inTaranta [CT-412];
3. To develop a widget from scratch using svg images or canvas.

1 Results from assigning fix values to width and height in the identified lines of the code

In `plot_api.js` we modified the code by setting the above mentioned lines 509 and 510 to `context._hasZeroHeight = 0;` and `context._hasZeroWidth = 0;` respectively. This change highlighted the problem in `plots.js` which can be solved by replacing the code in line 1608 and 1609 with `newWidth = 100;`and `newHeight = 100;` respectively.
The benchmark conducted after implementing these changes didn’t give clear results because, after resolving the first bottleneck, a list of other possible ones have been identified in the dependent libraries.For this reason we decided to change our approach.

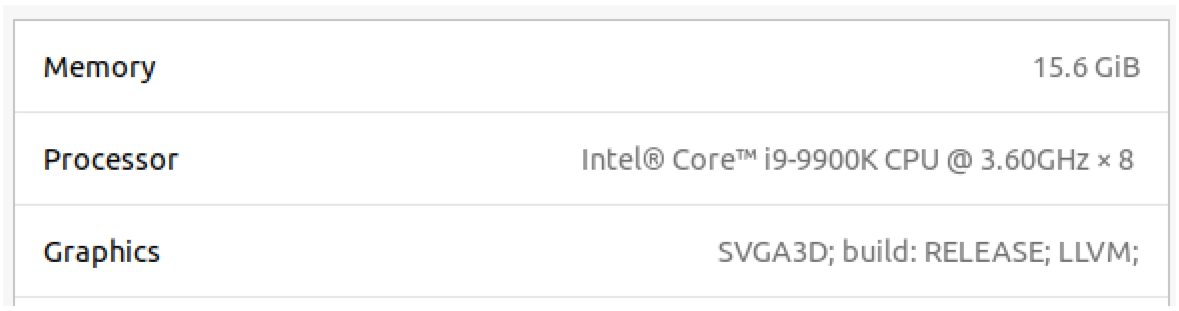
2 Results from benchmarking other available graph libraries

Different libraries/technologies have been tested with the goal of identifying a possible alternative to plotly to be used in Taranta. A comparison between the performance of plotly and of each of the

investigated alternatives is shown in Table 1. Canvas and SVG options are discussed in more detail in the dedicated subsection. All the other libraries adapt D3.js to React.
Values in Table 1 have been obtained by benchmarking different libraries while running a dashboard with 20 Spectrum widgets. The libraries to be included in the study have been selected among the best ones reported in literature. The computer setup used for conducting the test is reported in Figure 6.
The collected data clearly show that none of the tested libraries improves performances significantly. For this reason substituting plotly with one of those doesn't seem a sensible solution.

Table 1: Benchmark results: comparison between different solutions. Percentages indicate the ratio between time spent while performing a task, such as rendering, compared to the total duration of the test. The line containing plotly data has been highlighted as a reference along with the percentage of idle time which is a good indicator of highly performing solutions (the longer the idle time the highest the refreshing speed).

	Scripting	Rendering	Painting	System	Idle	Loading
Canvas	1.95%	0.01%	0.02%	0.24%	97.78%	0%
SVG	1.97%	0.14%	0.03%	0.56%	97.3%	0%
Plotly	18.8%	2.1%	0.1%	0.5%	78.5%	0%
TimeSeries	19.4%	1.47%	0.13%	0.32%	78.68%	0%
D3	39.37%	0.66%	0.16%	0.34%	59.47%	0%
Viser	50.2%	0.22%	0.16%	5.39%	42.9%	1.13%
ChartJS	56.7%	0.009%	0.02%	20.07%	23.2%	0%
Rechart	82.4%	4.9%	0.4%	1.4%	10.9%	0%



Memory	15.6 GiB
Processor	Intel® Core™ i9-9900K CPU @ 3.60GHz × 8
Graphics	SVGA3D; build: RELEASE; LLVM;

Figure 6: Computer setup parameters.

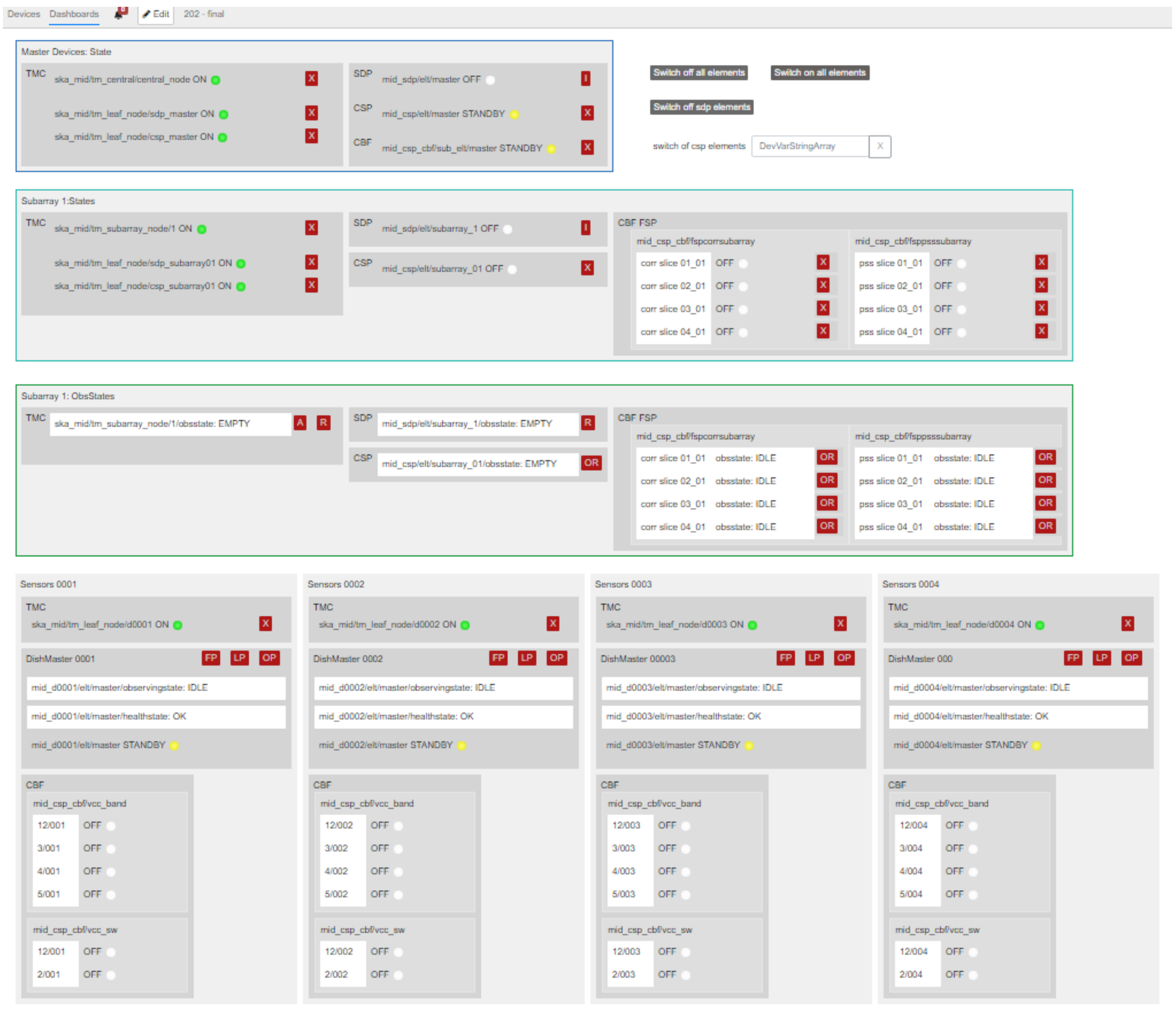
3 Results after implementing the Spectrum Attribute widget from scratch

As a final test we tried to implement the simplest plot widget available in Taranta, the Spectrum widget, from scratch, with the goal of evaluating both potential improvements in performances and the feasibility of starting a new library based on html and … instead of relying on D3.js.
Benchmark results are reported in Table 1 (row one and two to be compared with line three) and show that this solution would be the most performant one.

The drawback is that we won’t be able to rely on stable working code and will have to implement all the charts needed by SKA and the widget customization from scratch.

Improve the loading time for the “Resources Readiness” dashboard

The final part of the study has been devoted to reducing the loading time for a specific dashboard, See Figure 7.
Problem statement: after pressing the “start” button in Taranta, the user has to wait about 10 seconds before the dashboard is fully loaded.
The issue was caused by a specific widget, the State widget, which was sequentially fetching the metadata of every device linked to it. The dashboard checks the state of a relatively high number of different devices and this triggered a long queue of calls to the `FetchDeviceMetadata` function which caused the long loading time.
To improve this behaviour we relied on the ability of TangoGQL to fetch all the deployed devices at once. The change in performance is substantial, and brought the loading time down to about 1s.



Master Devices: State

TMC
ska_mid/tm_central/central_node ON ●
ska_mid/tm_leaf_node/sdp_master ON ●
ska_mid/tm_leaf_node/csp_master ON ●

SDP mid_sdp/elt/master OFF ●
CSP mid_csp/elt/master STANDBY ●
CBF mid_csp_cbf/sub_elt/master STANDBY ●

Subarray 1:States

TMC
ska_mid/tm_subarray_node/1 ON ●
ska_mid/tm_leaf_node/sdp_subarray01 ON ●
ska_mid/tm_leaf_node/csp_subarray01 ON ●

SDP mid_sdp/elt/subarray_1 OFF ●
CSP mid_csp/elt/subarray_01 OFF ●

CBF FSP

mid_csp_cbf/fspcorrsubarray		mid_csp_cbf/fsppsssubarray	
corr slice 01_01	OFF ●	pss slice 01_01	OFF ●
corr slice 02_01	OFF ●	pss slice 02_01	OFF ●
corr slice 03_01	OFF ●	pss slice 03_01	OFF ●
corr slice 04_01	OFF ●	pss slice 04_01	OFF ●

Subarray 1: ObsStates

TMC ska_mid/tm_subarray_node/1/obsstate: EMPTY
SDP mid_sdp/elt/subarray_1/obsstate: EMPTY
CSP mid_csp/elt/subarray_01/obsstate: EMPTY

CBF FSP

mid_csp_cbf/fspcorrsubarray		mid_csp_cbf/fsppsssubarray	
corr slice 01_01	obsstate: IDLE	pss slice 01_01	obsstate: IDLE
corr slice 02_01	obsstate: IDLE	pss slice 02_01	obsstate: IDLE
corr slice 03_01	obsstate: IDLE	pss slice 03_01	obsstate: IDLE
corr slice 04_01	obsstate: IDLE	pss slice 04_01	obsstate: IDLE

Sensors 0001

TMC
ska_mid/tm_leaf_node/d0001 ON ●

DishMaster 0001
mid_d0001/elt/master/observingstate: IDLE
mid_d0001/elt/master/healthstate: OK
mid_d0001/elt/master STANDBY ●

CBF

mid_csp_cbf/vcc_band	
12/001	OFF ●
3/001	OFF ●
4/001	OFF ●
5/001	OFF ●

mid_csp_cbf/vcc_sw	
12/001	OFF ●
2/001	OFF ●

Sensors 0002

TMC
ska_mid/tm_leaf_node/d0002 ON ●

DishMaster 0002
mid_d0002/elt/master/observingstate: IDLE
mid_d0002/elt/master/healthstate: OK
mid_d0002/elt/master STANDBY ●

CBF

mid_csp_cbf/vcc_band	
12/002	OFF ●
3/002	OFF ●
4/002	OFF ●
5/002	OFF ●

mid_csp_cbf/vcc_sw	
12/002	OFF ●
2/002	OFF ●

Sensors 0003

TMC
ska_mid/tm_leaf_node/d0003 ON ●

DishMaster 00003
mid_d0003/elt/master/observingstate: IDLE
mid_d0003/elt/master/healthstate: OK
mid_d0003/elt/master STANDBY ●

CBF

mid_csp_cbf/vcc_band	
12/003	OFF ●
3/003	OFF ●
4/003	OFF ●
5/003	OFF ●

mid_csp_cbf/vcc_sw	
12/003	OFF ●
2/003	OFF ●

Sensors 0004

TMC
ska_mid/tm_leaf_node/d0004 ON ●

DishMaster 000
mid_d0004/elt/master/observingstate: IDLE
mid_d0004/elt/master/healthstate: OK
mid_d0004/elt/master STANDBY ●

CBF

mid_csp_cbf/vcc_band	
12/004	OFF ●
3/004	OFF ●
4/004	OFF ●
5/004	OFF ●

mid_csp_cbf/vcc_sw	
12/004	OFF ●
2/004	OFF ●

Figure 7: Resources Readiness dashboard.

Conclusions and suggestions for further improvements in Taranta performances

This study focused on the analysis of performance bottlenecks present in Taranta that slow the performances of the tool when a dashboard is running. We found that the graphical library has the highest impact in slowing down performances and investigated possible solutions to improve this area. In particular we 1) Refined the problem moving the focus from the dispatcher to the graphical rendering of widgets; 2) analysed the plotly library and explored three possible ways of increasing performances of graphical widgets (by directly modifying the library code, by changing the library itself and by developing charts from scratch).
Finally, the cause of the long loading time of the “Resource Readiness” dashboard has been debugged and the problem has been resolved resulting in a better experience for the user of the dashboard.

The extensive benchmark study performed by the team also provided a better idea for further improvements of Taranta code. They could be:

For the Edit part:

- The graphical library currently used, plotly, renders widgets everytime they are dragged across the canvas. This behaviour could be optimised;
- The way in which information is stored on the database can be optimized.

For the Load part:

- Three different queries are performed while loading a dashboard. They retrieve all the deployed devices/attributes and then filter them to display those used by the dashboard. This mechanism could be trimmed to query only the devices/attributes used in the dashboard from the start. Such a development would include changes in TangoGQL.

For the Refresh part:

- The library currently in use by graphical widgets is not optimised with regards to performances. Custom solutions have been prototyped and proved to improve the rendering speed. It has to be noticed though that adopting this solution will mean to develop from scratch the graphical widgets needed in SKA and all the related interactions modalities (zoom in and out, pan, etc.) and functionalities.
- Optimise queries to the back-end.

References

[SP-586] “Refactor Taranta to improve performance” https://jira.skatelescope.org/browse/SP-586
[SP-296] “Spike to benchmark performance of WebJive”,
https://jira.skatelescope.org/browse/SP-296, See the linked documentation, in particular:
https://drive.google.com/file/d/14W-oajXd8n_TpUw_S6BsmIHiKFaIHHYc/view
[CT-341] “Benchmark js workers” https://jira.skatelescope.org/browse/CT-341
[CT-398] “Identify what function(s) is(are) slowing down the widgets” ,
https://jira.skatelescope.org/browse/CT-398

[Lighthouse] Lighthouse https://developers.google.com/web/tools/lighthouse