Conclusions on Webjive performance spike

Giorgio Brajnik - Interaction Design Solutions
2019-10-16

Introduction

The aim of this short document is to provide a conclusion regarding the performance benchmarking that was recently done by the BUTTONS team on webjive suite as per feature https://jira.skatelescope.org/browse/SP-296.
I'd like to say that the BUTTONS team has done a really excellent piece of work.

Please refer to the more detailed documents that the team has produced:

- analysis of the Webjive frontend https://docs.google.com/document/d/1XJFmw9XJAoXqP7bPkOGdOfvIrzg0Iq32-3hNgFcinZs/edit?usp=sharing
- analysis of TangoGQL https://docs.google.com/document/d/1sPm8wmMq8q8SoZ7Rx18369xHPLmSsBCUm12BWV9M58Q/edit?usp=sharing and
- https://docs.google.com/document/d/16kVP6mk8W76ge2Z2GhKqYnaSyqRen7Na8ZB-4eRemXM/edit?usp=sharing

For the purpose of this benchmark, Webjive Suite can be seen as the composition of two components:

- **backend**: a first one that queries a Tango server and produces data frames, one for each data to be displayed/rendered in some widget of some dashboard. This is currently implemented by TangoGQL, in python and graphQL. A data frame is a javascript structure that includes the data to display, the name of the attribute, of the device, etc.
- **frontend**: a second one that runs within the user browser that handles dashboards, widgets and gets data frames and populates widgets.

The benchmark was not focused on the performance of the dashboard editor.
To do the benchmark the team dissected Webjive and analysed separately TangoGQL and the frontend. To test the frontend a fast and ad-hoc backend was written (in C++ and using a websocket).

The desired performance limits are

- "We also want UIs that can open new screens and populate them with data very quickly. In less than 2 sec a new screen is launched."
- "Each screen should be able to handle an update rate of **1000 updated items/sec**."

Performance of TangoGQL

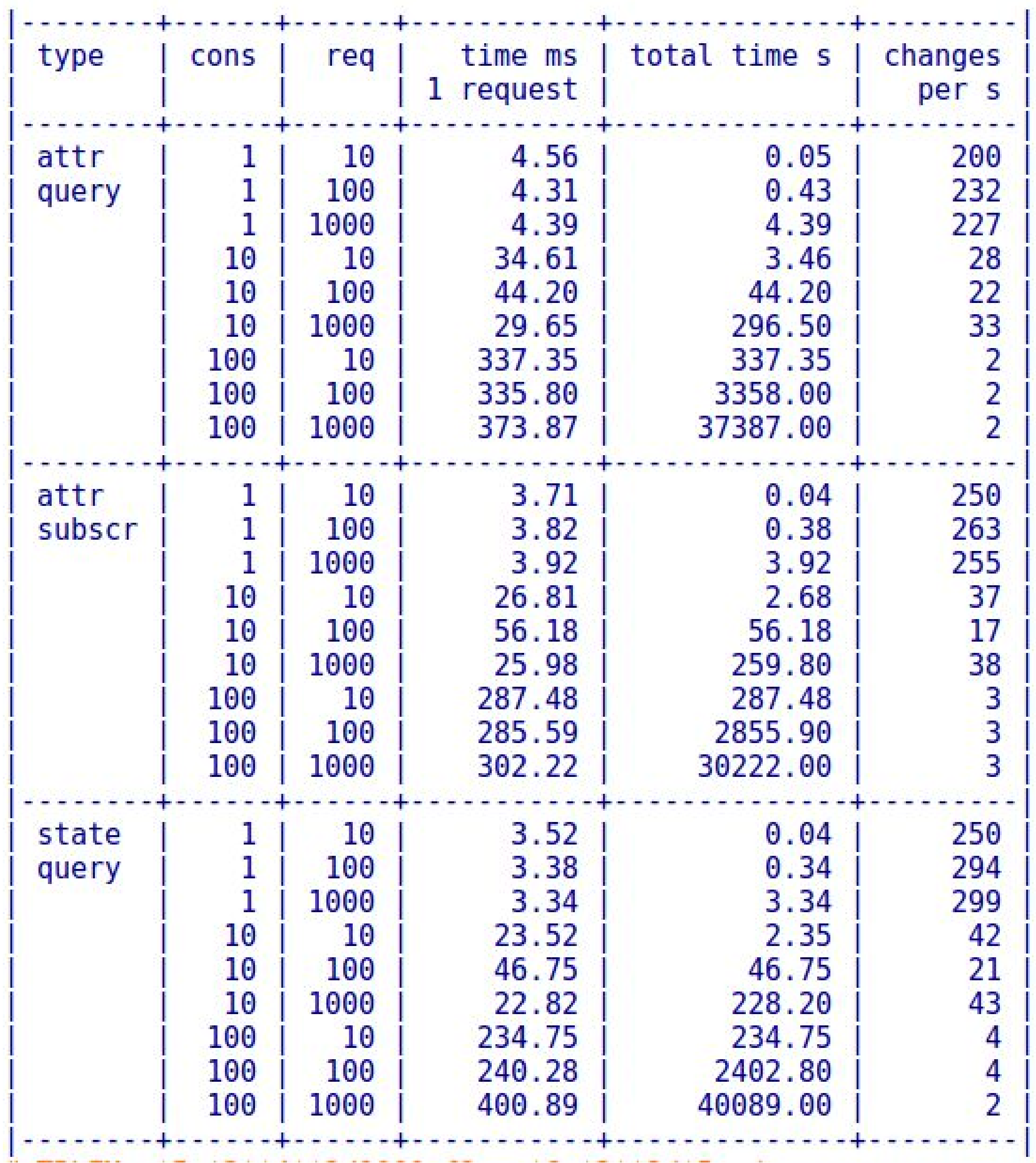
Profiler analysys

Based on the profiling report of TangoGQL the following conclusions can be drawn.
The test was done by submitting to TangoGQL different kinds of requests:

- attribute query: Querying an attribute and retrieving a value
- attribute subscription: Subscribing to an attribute and waiting for the initial result to be returned
- Device Command: Send a simple command to the device
- Query State: Get the current state of the device

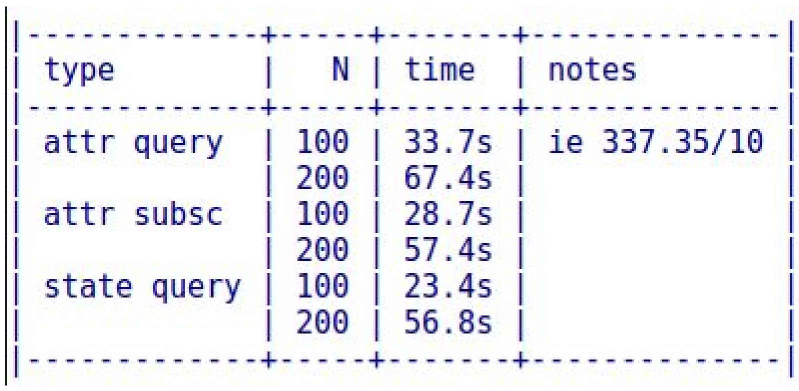
For our top-level performance requirements sending a command to a device is not a relevant task. And in our context each consumer can be meant as being a particular widget instance in the active dashboard.

Twenty runs of 10, 100, 1000 requests were launched using a single consumer; then the same runs were executed using 10 concurrent consumers; and then using 100 concurrent consumers. The following table summarises the median value of the time needed by TangoGQL to process each single request and to ask the device server for the data.



```
|--------+------+------+-----------+--------------+---------|
| type   | cons |  req |   time ms | total time s | changes |
|        |      |      | 1 request |              |   per s |
|--------+------+------+-----------+--------------+---------|
| attr   |    1 |   10 |      4.56 |         0.05 |     200 |
| query  |    1 |  100 |      4.31 |         0.43 |     232 |
|        |    1 | 1000 |      4.39 |         4.39 |     227 |
|        |   10 |   10 |     34.61 |         3.46 |      28 |
|        |   10 |  100 |     44.20 |        44.20 |      22 |
|        |   10 | 1000 |     29.65 |       296.50 |      33 |
|        |  100 |   10 |    337.35 |       337.35 |       2 |
|        |  100 |  100 |    335.80 |      3358.00 |       2 |
|        |  100 | 1000 |    373.87 |     37387.00 |       2 |
|--------+------+------+-----------+--------------+---------|
| attr   |    1 |   10 |      3.71 |         0.04 |     250 |
| subscr |    1 |  100 |      3.82 |         0.38 |     263 |
|        |    1 | 1000 |      3.92 |         3.92 |     255 |
|        |   10 |   10 |     26.81 |         2.68 |      37 |
|        |   10 |  100 |     56.18 |        56.18 |      17 |
|        |   10 | 1000 |     25.98 |       259.80 |      38 |
|        |  100 |   10 |    287.48 |       287.48 |       3 |
|        |  100 |  100 |    285.59 |      2855.90 |       3 |
|        |  100 | 1000 |    302.22 |     30222.00 |       3 |
|--------+------+------+-----------+--------------+---------|
| state  |    1 |   10 |      3.52 |         0.04 |     250 |
| query  |    1 |  100 |      3.38 |         0.34 |     294 |
|        |    1 | 1000 |      3.34 |         3.34 |     299 |
|        |   10 |   10 |     23.52 |         2.35 |      42 |
|        |   10 |  100 |     46.75 |        46.75 |      21 |
|        |   10 | 1000 |     22.82 |       228.20 |      43 |
|        |  100 |   10 |    234.75 |       234.75 |       4 |
|        |  100 |  100 |    240.28 |      2402.80 |       4 |
|        |  100 | 1000 |    400.89 |     40089.00 |       2 |
|--------+------+------+-----------+--------------+---------|
```

Therefore we should expect (by gross interpolation) that N consumers launching 1 request each might require:



```
|-------------+-----+-------+-------------|
| type        |   N | time  | notes       |
|-------------+-----+-------+-------------|
| attr query  | 100 | 33.7s | ie 337.35/10 |
|             | 200 | 67.4s |             |
| attr subsc  | 100 | 28.7s |             |
|             | 200 | 57.4s |             |
| state query | 100 | 23.4s |             |
|             | 200 | 56.8s |             |
|-------------+-----+-------+-------------|
```

This is way above the 2s threshold for loading time of dashboards.

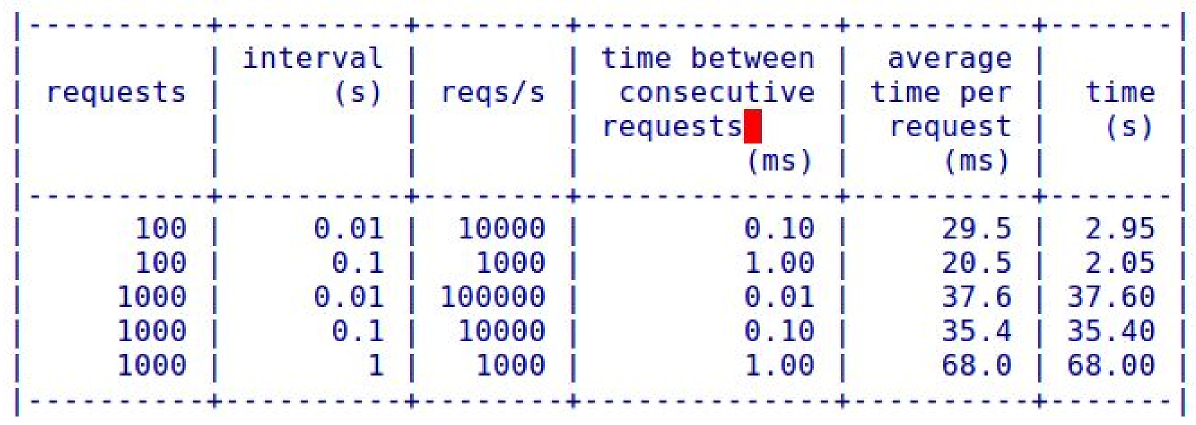
Regarding the refresh rate, the above table shows that the number of changes per second supported by TangoGQL is also way below the required threshold (1000).
Thus the current implementation of TangoGQL does not meet the required thresholds (and does not even come close to meeting them).

It might be the case that by replacing the polling mechanism that is currently used with an event-driven solution TangoGQL gets closer to the required threshold. In fact, with 1 consumer running 100 requests the total time is less than 0.43s: not ideal but much better.

Jmeter analysis

Based on the jmeter analysis of the response time of TangoGQL, the test was conducted by sending 1, 10, 100, 1000 requests to TangoGQL within specific time intervals (0.01, 0.1, 0.5, 1s) and measuring the time duration between just before sending the request to just after the first response has been received.

The following table summarises some of the average of the median times that were collected using jmeter when in a time interval of X sec jmeter sent Y requests.



```
|----------+----------+--------+-------------+----------+-------|
|          | interval |        | time between|  average |       |
| requests |      (s) | reqs/s |  consecutive| time per |  time |
|          |          |        | requests    |  request |   (s) |
|          |          |        |         (ms)|     (ms) |       |
|----------+----------+--------+-------------+----------+-------|
|      100 |     0.01 |  10000 |         0.10|     29.5 |  2.95 |
|      100 |      0.1 |   1000 |         1.00|     20.5 |  2.05 |
|     1000 |     0.01 | 100000 |         0.01|     37.6 | 37.60 |
|     1000 |      0.1 |  10000 |         0.10|     35.4 | 35.40 |
|     1000 |        1 |   1000 |         1.00|     68.0 | 68.00 |
|----------+----------+--------+-------------+----------+-------|
```

There a few unexpected aspects of the data: in fact in the original data table there are some values which are outliers, and it is not clear what causes them (I removed them from my analysis).

Conclusions regarding the load < 2s:

- 100 widgets might send 100 requests in 10ms; the total response time of TangoGQL is 2.95s
- 100 widgets might send 100 requests in 100ms: total time is 2.05s
- 1000 widgets might send 1000 requests in 10ms: total time is 37.49s
- 1000 widgets might send 1000 requests in 100ms: total time is 35.40s
- 1000 widgets might send 1000 requests in 1s: total time is 68s

Obviously the reference threshold is far from being met.

Conclusions regarding the 1000 changes/s:

- if we assume that the frontend sends in 100 requests every 100ms, then tangoGQL takes 2.05s to process them
- assuming that the frontend sends in 1000 requests every 1s, tangoGQL takes 68s to process them.

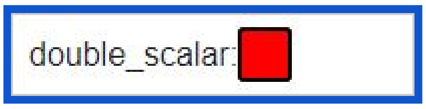
Again TangoGQL is way below the threshold and is not able to sustain that change rate in dashboards.
TangoGQL is capable, though, of handling all the requests that were sent, even at the highest rates (like 10^5 requests/s). The order in which the responses were produced differed from the order in which requests were sent, but all of them were processed.

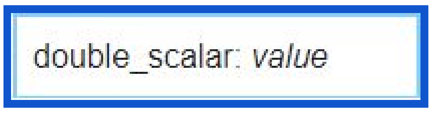
Based on the findings of the profiling and jmeter activity I recommend that:

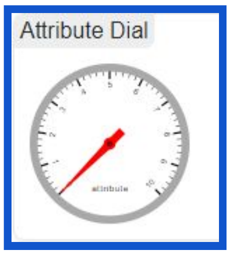
- the pub/sub mechanism is implemented to get rid of the delay incurred because of the polling mechanism
- the same tests are repeated (also with 100 and 200 consumers launching 1 request each)
- additional tests are run to explore the impact that the caching mechanism might introduce
- if this fails to improve TangoGQL performance, some other sort of solution not based on GraphQL needs to be design - maybe for the problem of getting data from tango devices and moving them to a frontend, GraphQL is not the best technology. Or perhaps a pool of websockets instead of a single one could make a difference.

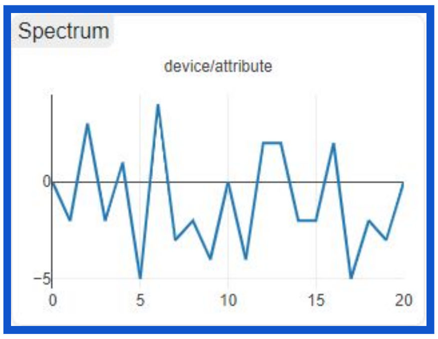
Performance of the frontend

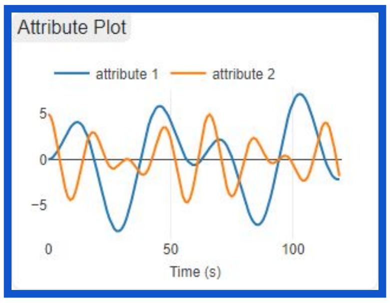
The following widgets were used to create dashboards with 1, 10, 100, 1000 copies of the same widget:

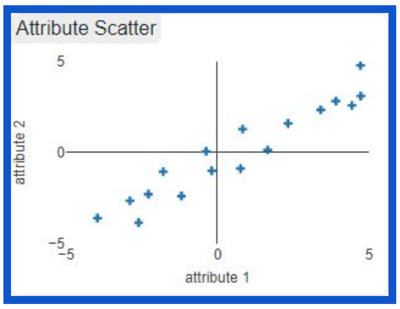
- LED widget 

- attribute display 

- dial 

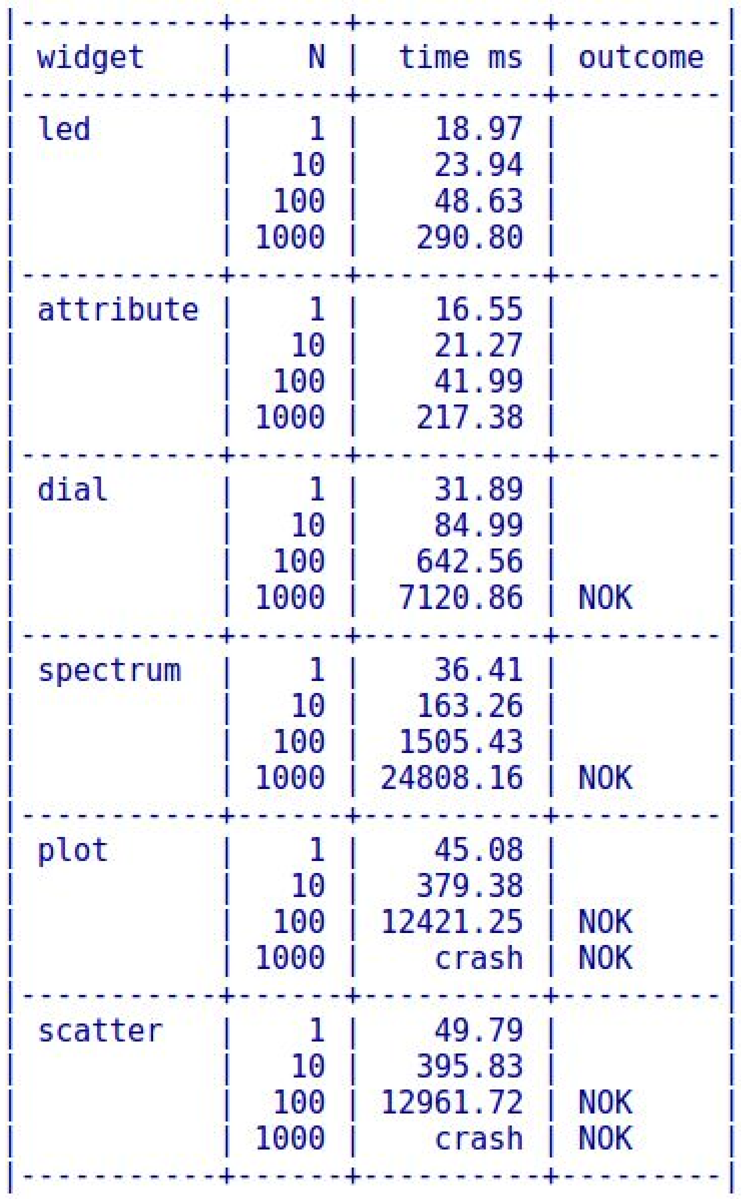
- spectrum 

- plot 

- scatter 

Load time

The following table reports the **load time**, namely the time needed from the moment a user clicks the “start” button on the dashboard until the moment that all the widgets are visible on the dashboard and Webjive has sent all the subscription information to the backend (either tangoGQL or a websocket-based daemon written for testing purposes). In particular the time reported below does not include the time spent by Tangogql to process the query and to transfer the data.



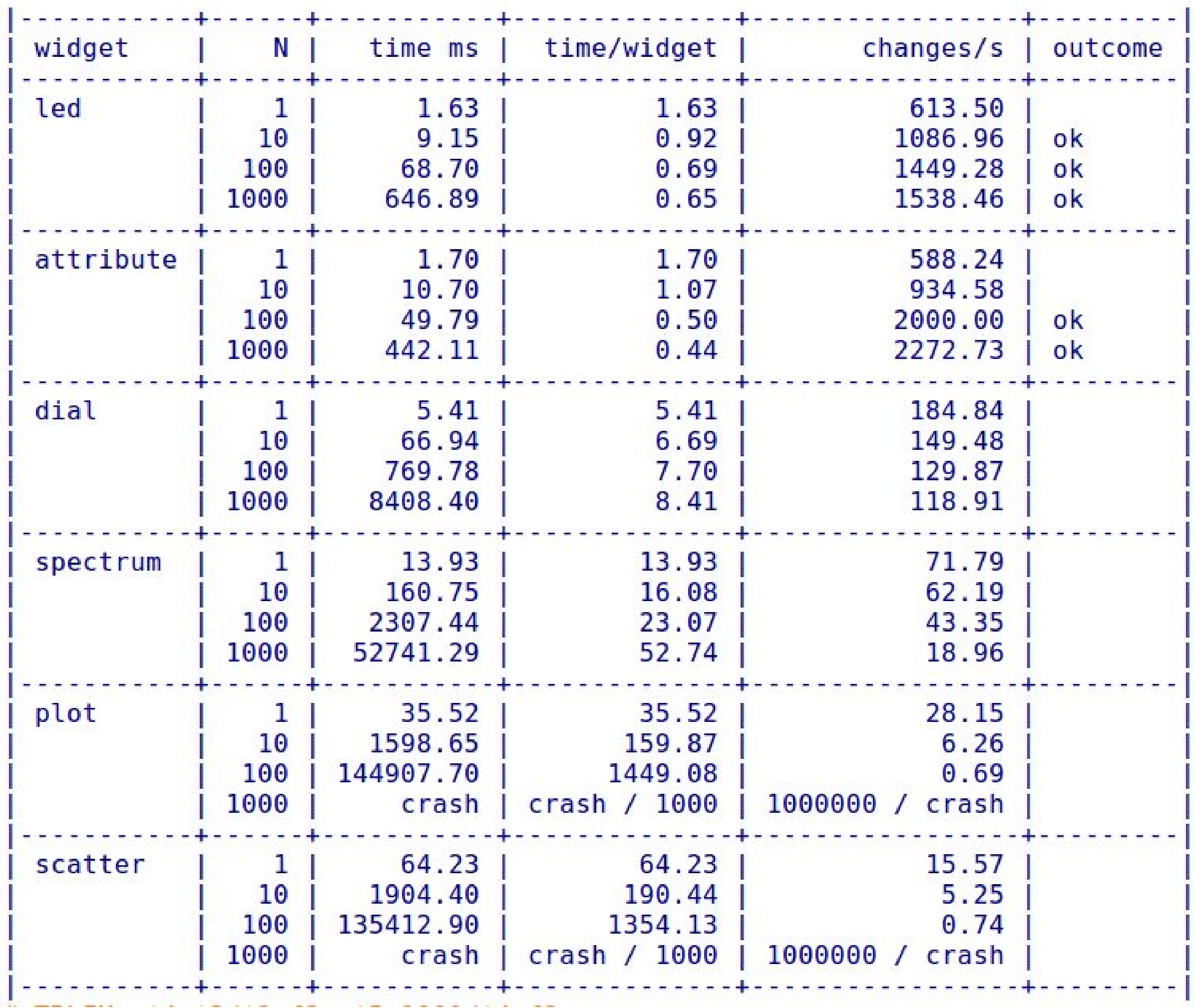
```
|-----------+------+----------+---------|
| widget    |    N |  time ms | outcome |
|-----------+------+----------+---------|
| led       |    1 |    18.97 |         |
|           |   10 |    23.94 |         |
|           |  100 |    48.63 |         |
|           | 1000 |   290.80 |         |
|-----------+------+----------+---------|
| attribute |    1 |    16.55 |         |
|           |   10 |    21.27 |         |
|           |  100 |    41.99 |         |
|           | 1000 |   217.38 |         |
|-----------+------+----------+---------|
| dial      |    1 |    31.89 |         |
|           |   10 |    84.99 |         |
|           |  100 |   642.56 |         |
|           | 1000 |  7120.86 | NOK     |
|-----------+------+----------+---------|
| spectrum  |    1 |    36.41 |         |
|           |   10 |   163.26 |         |
|           |  100 |  1505.43 |         |
|           | 1000 | 24808.16 | NOK     |
|-----------+------+----------+---------|
| plot      |    1 |    45.08 |         |
|           |   10 |   379.38 |         |
|           |  100 | 12421.25 | NOK     |
|           | 1000 |    crash | NOK     |
|-----------+------+----------+---------|
| scatter   |    1 |    49.79 |         |
|           |   10 |   395.83 |         |
|           |  100 | 12961.72 | NOK     |
|           | 1000 |    crash | NOK     |
|-----------+------+----------+---------|
```

It can be seen from the above table that the webjive frontend **meets the threshold** (dashboard displayed in less than 2 s) for large dashboards (up to a few hundred widgets) that have simple widgets (LEDs, attribute display, and to some extent dials).

For complex widgets (plots, spectrum, scatter) the performance is much poorer. While there is margin for improvement (those widgets rely on third party's libraries and are likely not to be optimized for performance), because of the quantity of data that they need to display the rendering time of a dashboard with hundreds of these widgets will take significantly more than 2s.

Refresh time

The following table reports the number of **widgets changed per second**, based on the recorded time needed to process webjive data frames and dispatch them to the active dashboard and the appropriate widget.



```
|-----------+-----+-----------+-------------+-----------------+---------|
| widget    |   N |   time ms | time/widget |       changes/s | outcome |
|-----------+-----+-----------+-------------+-----------------+---------|
| led       |   1 |      1.63 |        1.63 |          613.50 |         |
|           |  10 |      9.15 |        0.92 |         1086.96 | ok      |
|           | 100 |     68.70 |        0.69 |         1449.28 | ok      |
|           | 1000|    646.89 |        0.65 |         1538.46 | ok      |
|-----------+-----+-----------+-------------+-----------------+---------|
| attribute |   1 |      1.70 |        1.70 |          588.24 |         |
|           |  10 |     10.70 |        1.07 |          934.58 |         |
|           | 100 |     49.79 |        0.50 |         2000.00 | ok      |
|           | 1000|    442.11 |        0.44 |         2272.73 | ok      |
|-----------+-----+-----------+-------------+-----------------+---------|
| dial      |   1 |      5.41 |        5.41 |          184.84 |         |
|           |  10 |     66.94 |        6.69 |          149.48 |         |
|           | 100 |    769.78 |        7.70 |          129.87 |         |
|           | 1000|   8408.40 |        8.41 |          118.91 |         |
|-----------+-----+-----------+-------------+-----------------+---------|
| spectrum  |   1 |     13.93 |       13.93 |           71.79 |         |
|           |  10 |    160.75 |       16.08 |           62.19 |         |
|           | 100 |   2307.44 |       23.07 |           43.35 |         |
|           | 1000|  52741.29 |       52.74 |           18.96 |         |
|-----------+-----+-----------+-------------+-----------------+---------|
| plot      |   1 |     35.52 |       35.52 |           28.15 |         |
|           |  10 |   1598.65 |      159.87 |            6.26 |         |
|           | 100 | 144907.70 |     1449.08 |            0.69 |         |
|           | 1000|     crash | crash / 1000| 1000000 / crash |         |
|-----------+-----+-----------+-------------+-----------------+---------|
| scatter   |   1 |     64.23 |       64.23 |           15.57 |         |
|           |  10 |   1904.40 |      190.44 |            5.25 |         |
|           | 100 | 135412.90 |     1354.13 |            0.74 |         |
|           | 1000|     crash | crash / 1000| 1000000 / crash |         |
|-----------+-----+-----------+-------------+-----------------+---------|
```

The reference threshold here is supporting at least 1000 changes/s.

For large dashboards (a couple of hundreds of widgets) made of simple widgets (LEDs and attribute displays), the webjive frontend **meets the threshold** (even with more than 1000 widgets). With dials the performance gets below 1/5 of the threshold, and it degrades a lot for the other widgets.

There is margin for improvement, both in refactoring the frontend code that dispatches data frames to dashboards and to widgets and in refactoring the more complex widgets (as said above) to improve their performance.

Overall conclusions

WebJive Suite was designed to display relatively simple dashboards, with 10-20 widgets, not complex UIs for controlling devices. It was not designed to achieve a high performance level either.

It is remarkable that WebJive frontend is already matching the desired performance limits, at least for large dashboards with simple widgets.

Consider that most of the screens of existing control UIs are based on widgets that are the same or similar to the simpler ones considered in this benchmark (LEDs, attribute values, dials).

Recommendations

My recommendation is to keep considering Webjive Suite as the reference implementation framework for web-based engineering UIs to be used for SKA.

Additional work is needed to mature the Webjive Suite from the performance aspect; these are some of the activities that should be planned during the next PIs:

- implement the event-driven behavior of TangoGQL;
- refactor the dispatching code within the Webjive frontend and base it on asynchronous code;
- reassess the performance results of Webjive Suite based on the refactored dispatcher and TangoGQL;
- refactor some of the complex widgets (dial, plot, scatter, spectrum) when the need arises (i.e., when some UI needs that kind of widget and its performance bottleneck has a significant negative impact on users)
- identify the causes of the crashes mentioned above and then prioritize these bugs accordingly
- further improve the performance of the TangoGQL (maybe also by considering to replace TangoGQL with something that is simpler and more efficient).