ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP044

# IMPROVING THE PERFORMANCE OF TARANTA: ANALYSIS OF MEMORY REQUESTS AND IMPLEMENTATION OF THE SOLUTION*

Matteo Canzari† , INAF - Osservatorio Astronomico d'Abruzzo, Teramo, Italy
Hélder Ribeiro, Atlar Innovation, Rua Rangel de Lima, Portugal
Ajaykumar Dubey, PSL, Pune, India
Athos Georgiou, CGI Scotland, Scotland
Valentina Alberti, INAF - Osservatorio Astronomico di Trieste, Trieste, Italy
Yimeng Li, Vincent Hardion, Mikel Eguiraun, Johan Forsberg, Max-IV Institute, Lund, Sweden

*Abstract*

Taranta is a software suite for generating graphical interfaces for Tango Controls software, currently adopted by MaxIV for scientific experiment usage, SKA during the current construction phase for the development of engineering interfaces for device debugging, and other institutions. A key feature of Taranta is the ability to create customizable dashboards without writing code, making it easy to create and share views among users by linking the dashboards to their own tango devices. However, due to the simplicity and capabilities of Taranta's widgets, more and more users are creating complex dashboards, which can cause client-side resource problems. Through an analysis of dashboards, we have found that excessive memory requests are generated by a large amount of data. In this article, we report on the process we believe will help us solve this performance issue. Starting with an analysis of the existing architecture, the issues encountered, and performance tests, we identify the causes of these problems. We then study a new architecture exploiting all the potential of the Javascript framework React on which Taranta is built, before moving on to implementation of the solution.

## INTRODUCTION

Taranta [1] is a tool for creating user interfaces for Tango devices without the need for coding [2]. In the Problem Impact section, we will analyze how the use of Taranta for creating complex dashboards has led to performance and system stability issues, resulting in a loss of confidence within the community. We then explored an architectural solution to the problem described in the Problem Analysis section.

In the Implementation section, we describe the implementation of this solution, which was released in the latest version of Taranta [3], and its results are detailed in the Benchmark Test section. Finally, in the conclusions, we highlight the architectural improvements introduced into the system through the implemented changes and discuss potential enhancements.

* Work supported by the financial support by the Italian Government (MEF - Ministero dell'Economia e delle Finanze, MIUR - Ministero dell'Istruzione, dell'Università e della Ricerca)
† matteo.canzari@inaf.it

## PROBLEM IMPACT

The issue reported by Taranta users involved a gradual slowdown of dashboards containing numerous widgets, eventually leading to a complete halt in updates. Additionally, there was a progressive increase in browser RAM usage. This problem was highly inconvenient and, at times, detrimental for several reasons. Primarily, users were unaware that the values from Tango were not being updated at the correct frequency due to the lack of a runtime notification mechanism. Worse still, when the slowdown was severe enough to freeze the dashboard, it provided no feedback to the user. In this state, the user viewing the dashboard assumed that updates from Tango were not coming through when in reality, they had a way to ascertain the current state of the control system. Moreover, the system might not respond to commands, meaning a user could issue a command to change an attribute or state and not observe the result. This situation must be avoided, as the user may need to discern whether the issue stems from malfunctioning devices or an unresponsive user interface. Lastly, there was a concern regarding the escalating browser RAM usage. This was particularly troublesome, given that Taranta is a JavaScript application, hence client-side, and could significantly slow down the user's computer or even cause it to crash.

Addressing this problem promptly was imperative since these often severe issues were eroding user confidence in using Taranta. It is important to note that this type of issue arose only in dashboards with a high number of running widgets over an extended period. For simpler dashboards executed for a limited time, the problem was negligible.

The developers of Taranta were already aware of the performance limitations stemming from an architecture inadequate for supporting a high number of widgets and updates. However, priority was given to developing new features rather than resolving this issue. Taranta users had been using the software for several years without encountering significant problems and appreciated the newly added features. The community's reporting of this issue is a clear indication that users are increasingly adopting the software for more complex scenarios, not just as a simple prototyping and testing tool. This underscores the software's good quality, the utility of the introduced features, and a growing confidence in using Taranta.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP044

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

## PROBLEM ANALYSIS

In order to analyze the issue, a tool called React DevTool [4] was employed. React DevTool is a suite of tools that facilitates debugging of React applications, providing a range of useful instruments such as the Profiler. This tool allows for a component-by-component analysis of the incoming data flow and monitoring the ability to process and render the data.

Through analysis sessions, it was revealed that when a dashboard is run, triggering data transmission from TangoGQL, all components within the dashboard, including headers, etc., are triggered by each piece of data arriving at Taranta from TangoGQL. However, not every trigger necessarily results in an actual new component render, as the data is often not involved in the component. It is thus inferred that upon data arrival, Taranta forcibly refreshes all components within the dashboard sequentially. Therefore, the subsequent data must await a total refresh before being processed. Consequently, a few seconds after running the dashboard, all components, as evident in the image below, are unable to process the incoming data and thus get queued. As the volume of data and time increase, the queue of data to be processed grows, causing progressively greater delays, eventually leading to a halt in the dashboard update. The increase in the volume of data to be processed corresponds to an increase in the space required for their storage, hence resulting in the reported increase in RAM usage by users.

All these issues can be attributed to a known architectural problem. Let us consider the dashboard view, where each time data arrives from TangoGQL, it is handled by the class RunCanvas.tsx [5]. This class sequentially updates all components linked to the dashboard using attributeEmitter with the received data, even if these components are not involved with the received data.

Previously, no significant issues were encountered since users used Taranta solely for testing and debugging purposes, utilizing a limited number of widgets for a limited duration. As Taranta evolved into a more powerful visualization tool, users began creating increasingly complex dashboards and running them for longer periods based on use cases. For instance, these dashboards would run for the entire duration of an experiment, which could last for hours. The increase in the number of widgets in a single dashboard led to a rise in the number of associated React Components. The addition of connected devices increased the volume of data arriving at a dashboard. The longer execution times led to an increase in the message queue. All of these factors, exacerbated by users' need to open multiple sessions simultaneously, resulted in dashboard crashes and an overall system slowdown, as elaborated in the preceding section.

In reality, this architectural problem had been identified earlier during benchmark testing of Taranta in its initial versions, back when it was still known as Webjive. The design rationale behind this architectural choice stemmed from the initial intent of the project, which was to create dashboards with few devices and widgets, to be used for a limited period. However, as the project evolved and the range of use cases expanded, this architectural solution proved to be limiting. Consequently, a comprehensive software refactor was deemed necessary.

## ARCHITECTURAL SOLUTION

To address the architectural issue, our focus centered on the relationship between components and data. The primary challenge was that each component was triggered every time new data arrived, regardless of whether the data was relevant to the component. This was due to the component management being delegated to specific classes that responded to incoming data. Consequently, we designed an architecture where each component is connected to the data and is triggered only when data of relevance arrives. Otherwise, the component remains on standby, avoiding unnecessary renders. Additionally, we aimed to decouple individual components from the data structure. Previously, components were directly connected and dependent on Tango APIs developed for Taranta to establish connections with TangoGQL. However, in this new approach, we opted to utilize an internal store within the web application to connect React Components. This ensured that components were not directly tied to the data source, thereby improving architectural modifiability, extensibility, and testability. To complete the system's architecture, we introduced a middleware responsible for subscribing and unsubscribing components to data and populating the internal store.

## IMPLEMENTATION

The proposed architectural solution is based on the React technology stack, incorporating Redux [6] to fully leverage the capabilities of this popular framework. React operates on the principle of 'reacting' to changes in data or, more precisely, the component's state, rendering only the involved component thanks to the Virtual DOM, a core concept introduced by React.

Each component can maintain its state as an object, and when this state changes, React automatically re-renders the component, reflecting those changes in the user interface. The state is managed internally within a component and represents the data that may change over time based on various events or user interactions. Components can re-render to update the UI based on the current state.

Redux extends the concept of local component state provided by React. It is a state management library that allows for the management of application state in a more predictable and centralized manner. The Redux Store was employed as the application's store.

The Redux store is a central, plain JavaScript object that encapsulates the entire state tree of a Redux application. It serves as the nucleus of a Redux application, functioning as a single source of truth for the application's state. The store contains the application state and provides a mechanism for dispatching actions to update the state based on those actions.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP044

The management of data population, subscription, and unsubscription was delegated to the Redux middleware. In Redux, middleware is a means to extend the functionality of the store's dispatch method. It acts as a third-party extension point between dispatching an action and the moment it reaches the reducer. Middleware enables the execution of additional code and logic before an action is handled by the reducer, facilitating the implementation of various advanced capabilities. Middleware functions in Redux have access to the action, the current state, and the ability to dispatch new actions. This makes middleware a potent tool for adding features such as logging, asynchronous behavior, or modifying actions before they reach the reducers.

The communication with TangoGQL is naturally handled by the middleware, specifically named websocketmiddleware, which is responsible for subscribing and unsubscribing to the data requested by components and populating the store. Figure 1 is an architectural diagram.

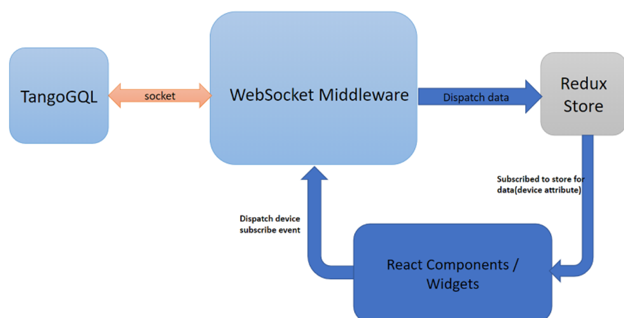


Figure 1: Websocket block diagram.

The communication mechanism between components and the store was designed following best practices in the React/Redux stack.

Taranta operates by establishing a socket connection at the commencement of the runCanvas, initiated by pressing the Start button on dashboards, or upon the selection of a device in the overview. This socket remains active until either the Edit button is pressed on the dashboards, or the user exits the overview or closes Taranta within the browser.

Taranta has two points of data injection: one for dashboards and another for the overview.

These injection points are defined in the following files:

- taranta/src/dashboard/components/RunCanvas/emmiter.ts (for dashboards)
- taranta/src/jive/state/api/tango/index.js (for the overview)

In both these files, a function called socketUrl(tangoDB: string) defines the URL for socket communication.

For communication between Taranta and TangoGQL, the data type is defined by Graphene-Python, a library used to construct GraphQL APIs in Python.

In the creation of the WebSocket middleware, a socket object is instantiated to establish a connection with TangoGQL. Several listeners are attached to this socket object, including:

- **open**: This listener allows for the execution of tasks or setting flags to indicate successful socket opening.
- **message**: This listener is activated each time data or a message is received from TangoGQL via the socket. Upon receiving a message, an action is dispatched to store this message in the store. The subscribed components then react (re-render) based on this updated data in the store.
- **disconnected**: This listener is invoked when the socket connection is terminated.

Additionally, the WebSocket middleware continuously listens to Redux events (UNSUBSCRIBE, SUBSCRIBE, etc.).

- **SUBSCRIBE EVENT**: This event is triggered when the socket is open, but no data is received from TangoGQL. To receive data (in this case, AttributeDisplay), a subscribe event needs to be raised for a specific device attribute. For example, the Attribute Display component can raise a SUBSCRIBE event with the device and the attribute in the payload. The WebSocket middleware listens to this subscribe event and requests data (in this case, the device attribute value) from TangoGQL via the socket.
- **UNSUBSCRIBE EVENT**: This event facilitates unsubscribing from data coming from TangoGQL.
- Custom listeners can also be implemented as needed.

The store is an object that maintains the state of the entire application in object format. Every incoming message or data from TangoGQL is stored within the store. React components are subscribed to the store object. Any change in the store prompts the respective subscribed component to re-render. Figure 2 is a Websocket architecture.

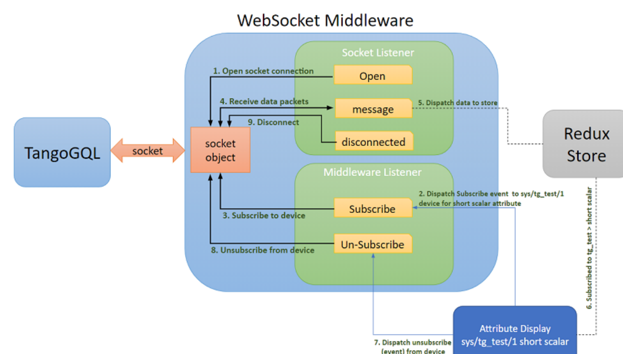


Figure 2: Websocket architecture.

## *Implementation Roadmap*

The initial phase of development commenced with Taranta version 1.3.12 and was divided into two stages. The first involved creating the middleware and the store, along with



Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP044

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

refactoring the Device view. The Device portion was chosen due to its relative simplicity, and releasing this version as a beta test allowed for valuable feedback to address any potential bugs or malfunctions resulting from such a significant refactor.

The released version was 2.0.0 [7], involving 125 commits and modifications to 101 files. Upon the release of this version, the community was encouraged to test and utilize it, aiming to identify and rectify errors while implementing necessary improvements.

Following the release of version 2.0.0, a new branch was created for a complete refactor of the dashboard view. Given the complexity of this refactor, both versions, 2.0.0 (for bug fixes) and the version under development, were maintained. In addition to the dashboard refactor, a comprehensive update of libraries to their latest versions and an upgrade to React 17 were performed. Upon completion of development, various improvements and bug fixes from the 2.0.0 version of the main branch were also incorporated. The final released version, inclusive of all updates, was version 2.4.0 [8], involving 226 commits and 342 files.

## BENCHMARK TEST

The results of the conducted benchmarks are then reported. To carry out the tests, a dedicated Tango device was developed, comprising 100 attributes, each characterized by:

- The attribute name follows the pattern Int_RO_xxx, where xxx is a sequential number ranging from 001 to 100.
- The attribute value is a random integer between 0 and 100, followed by the character '@' and the timestamp indicating when the number was generated.
- The attribute updates every second.
- An attribute updates with a one-tenth-of-a-second delay from the subsequent attribute.

The tests were conducted using this type of machine. Other machine types with varying operating systems, browsers, and hardware produced similar results.

Machine Specifications:

- MacBook Pro
- 2.6 GHz Intel Core i7 6-core
- AMD Radeon Pro 5300M 4 GB
- 16 GB 2667 MHz DDR4
- MacOS 13.4.1 (c) (22F770820d)
- Google Chrome 117.0.5938.149 (Official Build) (x86_64)

Two tests were created using the Performance tool of Chrome DevTool version 117. The tested versions of Taranta were 1.3.12, the version without the refactor, and 4.2.1, the version with the complete refactor and some bug fixes introduced after the release of 4.2.0. Both tests ran for 10 seconds, capturing data once the data flow towards Taranta commenced. The subscription and unsubscription phases were not recorded.

The first test focused on the Device view. Specifically, the Attributes tab was tested, wherein data from the 100 attributes, which update every second, is rendered.

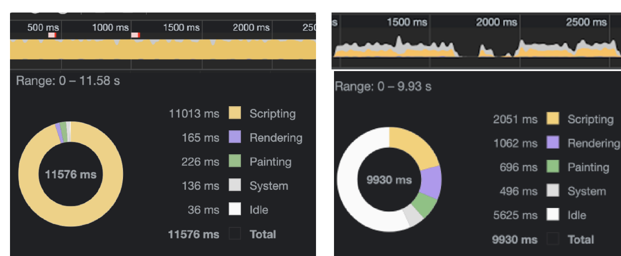


Figure 3: Device view benchmark results.

In Fig. 3, we observe the results of version 1.3.12 on the left, indicating that the majority of the time was dedicated to scripting. Only 36 milliseconds, out of slightly over 10 seconds in total, were spent idle. In contrast, utilizing version 4.2.1 on the right, we note a significantly reduced scripting time, with over half the duration spent in the idle state. Additionally, it is evident that the data flow in version 1.3.12 is saturated, while in version 4.2.1, it fluctuates. This suggests that in the former case, the components struggle to process the data in time, while in the latter, they manage to do so. This also justifies the rendering times, which are more than 10 times higher in the second test. This is not due to a slowdown, but because version 4.2.1, by processing more data, renders them quickly, thus achieving a higher rendering rate. Consequently, in the first version, the displayed data was not up to date, as initially complained by users.

An additional test was performed by creating a dashboard designed to concentrate data arrivals within specific intervals and observe how the system responded to these data impulses. The same dashboard was used to test both systems. The tests were repeated for two dashboards, each containing 300 and 600 widgets, respectively. This was to ascertain whether the performance scaled in a linear or exponential manner.

In Fig. 4, the left graph displays the benchmarks for version 1.3.12, while the right graph presents those for 4.2.1. The same test was repeated for 300 widgets, depicted in the upper section, and for 600 widgets, shown in the lower section.

It is evident that for 300 widgets, version 1.3.12 still manages to render the data, albeit occasionally reaching saturation. However, for 600 widgets, it is consistently at saturation and cannot render the data. It is noteworthy that while the scripting times reach saturation, the rendering times remain nearly the same, indicating that the components struggle

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP044

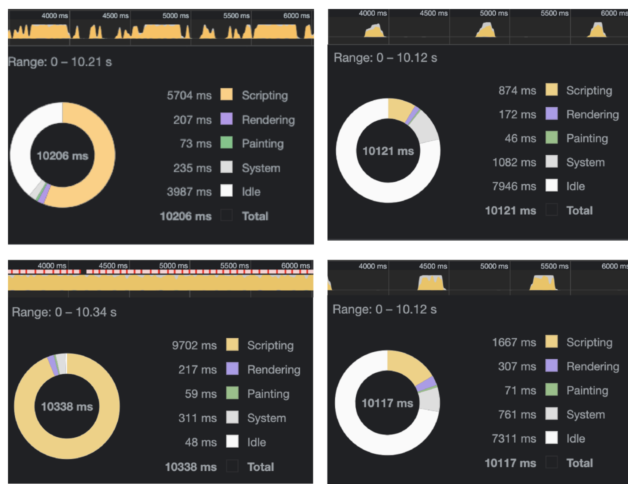


Figure 4: Dashboard view benchmark results.

to render the new data. Conversely, in version 4.2.1, the scripting times increase linearly with the number of widgets, and the rendering time is consistent. This suggests that the new version scales effectively even with more complex dashboards.

Below are tables presenting the data obtained from various tests.

Table 1: Scripting Time Benchmark Results

| Test | Taranta v. 1.3.12 | Taranta v. 2.4.1 |
|---|---|---|
| Device view | 11 013 ms | 2051 ms |
| Dash. 300 widg. | 5704 ms | 874 ms |
| Dash. 600 widg. | 9702 ms | 1667 ms |

Table 2: Idle Time Benchmark Results

| Test | Taranta v. 1.3.12 | Taranta v. 2.4.1 |
|---|---|---|
| Device view | 36 ms | 5625 ms |
| Dash. 300 widg. | 3987 ms | 7946 ms |
| Dash. 600 widg. | 48 ms | 7311 ms |

## CONCLUSION

The article has shed light on how, starting from the issues identified by users, it was possible to redesign and implement a version of Taranta that addressed the described architectural problems. In the dedicated testing section, we observed how specific performance tests conducted to measure the system's performance demonstrated a significant improvement in the software. Furthermore, during the system's refactoring, we updated libraries and dependencies to address security risks and vulnerabilities present in the old version. As anticipated in the section on the architectural solution, besides performance improvement, other architectural aspects such as testability and extensibility were enhanced.

Testability improved as we decoupled the data source, TangoGQL, from individual components connected through the store. This enabled the creation of dedicated tests for individual components by mocking the store and performing specific tests for the store and data connection through middleware testing. The result of this redesign was a significant increase in code coverage. Additionally, by decoupling individual components from the data source, the system's extensibility increased. Taranta is no longer tightly coupled with TangoGQL; we can now rewrite a middleware like WebsocketMiddleware that handles device subscription and store management. It will be possible to connect Taranta to other data sources, whether connected with Tango or not.

However, despite the theoretical simplicity of this last factor, there is no clear documentation on writing a new middleware, nor are there other case studies. One potential improvement could involve expanding the documentation and creating an alternative prototype to TangoGQL to demonstrate the system's extensibility.

Lastly, a mechanism to inform the user of any slowdown in data updates or a potential block has not yet been implemented. Despite the recent changes resulting in a faster system and no critical issues observed in updates even with complex dashboards, the user is still unaware of the delay between data sending and actual rendering.

## REFERENCES

[1] Taranta, `https://taranta.readthedocs.io/en/latest/`

[2] M. Canzari *et al.*, "How Taranta provides tools to build user interfaces for TANGO devices in the SKA integration environment without writing a line of code", *Proc. SPIE: Software and Cyberinfrastructure for Astronomy VII*, vol. 12189, Aug. 2022. doi:10.1117/12.2630141

[3] Y. Li *et al.*, "Taranta Project - Update and Current Status", presented at ICALEPCS'23, Cape Town, South Africa, Oct. 2023, paper FR2BCO03, this conference.

[4] React DevTool, `https://github.com/facebook/react/tree/main/packages/react-devtools`

[5] `https://gitlab.com/tango-controls/web/taranta/-/blob/1.3.12/src/dashboard/components/RunCanvas/RunCanvas.tsx`

[6] Redux, `https://redux.js.org/`

[7] `https://gitlab.com/tango-controls/web/taranta/-/merge_requests/362`

[8] `https://gitlab.com/tango-controls/web/taranta/-/merge_requests/452`

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI