ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

# canone3: A NEW SERVICE AND DEVELOPMENT FRAMEWORK FOR THE WEB AND PLATFORM INDEPENDENT APPLICATIONS*

G. Strangolino, L. Zambon, Elettra, Trieste, Italy


## *Abstract*

On the wake of former web interfaces developed at ELETTRA [1] as well as in other institutes, the service and development framework for the web and platform independent applications named PUMA (Platform for Universal Mobile application) has been substantially enhanced and rewritten, with the additional objectives of high availability, scalability, load balancing, responsiveness and customization. Thorough analysis of Websocket limits led to an SSE (Server-Sent Events) based server technology relying on channels (Nchan over NGINX) to deliver the events to the clients. The development of the latter is supported by JQuery, Bootstrap, D3js, SVG (Scalable Vector Graphics) and QT and helps build interfaces ranging from mobile to dashboard. Ultimate developments led to successful load balancing and failover actions, owing to the joint cooperation of a dedicated service supervisor and the NGINX upstream module.


## DESIGN RATIONALE

The system consists of a cluster of servers, thereafter synonymously named services, and two client side development environments. One is based on web technologies on browsers. The second is a C++ client library to build native Qt applications. The main objectives of the service design are reliability, security, scalability and accessibility. To satisfy them, a set of state of the art technologies and software serve as the groundwork of the system.

## RELIABILITY

The design rationale identifies the principles of a reliable service as follows. The system shall work:

- from any place and platform;
- at any time;
- regardless the number of clients
- included when part of the system is unavailable
- included when the network performance is suboptimal or even subject to charges.

The first requirement ruled out the *WebSocket* technology after an accurate analysis of its assets and liabilities.

### *WebSocket*

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. [2]

WebSockets are widespread and efficient when handling huge amount of messages from both ends, where duplex communication is continuously involved: Massive Multiplayer Online (MMO) and messaging applications.

The list of liabilities is nevertheless long in our situation:

- WebSockets can be potentially blocked by proxies;
- CORS (Cross-Origin Resource Sharing) [3] related concerns;
- no multiplexing over HTTP/2 (implementing it on both ends is complicated);
- no load balancing;
- susceptible to DoS;
- problems already taken care of in HTTP must be solved ad hoc;
- operational overhead in developing, testing and scaling is increased.

Some proxy servers are transparent and work fine with WebSockets; others will prevent them from working correctly, causing the connection to fail. In some cases, additional proxy server configuration is required.

Load balancing is very complicated. When servers are under pressure and new connections need to be created and old ones closed, the actions that must be taken can trigger a massive chain of refreshes and new data requests, additionally overloading the system. It's not possible to move socket connections to a different server to relieve one under high load. They must be closed and reopened. It turns out that WebSockets need to be maintained both on the server and on the client.

Multiplexing is usually handled by front end HTTP proxies that cannot be handled by TCP proxies which are needed for the WebSockets. Connecting to the sockets and flooding servers with data is a possible eventuality.

Concerning the last weakness in the list, we observe that mobile devices would maintain a WebSocket open by keeping the antenna and the connection to the cellular network active. Battery life would be reduced, heating increased, and, where applicable, extra costs for data usage applied.

### *SSE*

Server-Sent Events (SSE) is a server push technology enabling a client to receive automatic updates from a server via an HTTP connection, and describes how servers can initiate data transmission towards clients once an initial connection has been established. They are commonly used to send message updates or continuous data streams to a browser client and designed to enhance native, cross-browser streaming through a JavaScript API called EventSource, through which a client requests a

* inspiration by Alessio Igor Bogani, Elettra, Trieste, Italy

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

particular URL in order to receive an event stream. The Server-Sent Events EventSource API is standardized as part of HTML5 by the W3C [4].

Server-Sent Events technology analysis underlined some disadvantages; most of them do not appertain to our environment:

- SSE is unfit for duplex communications;
- binary data cannot be sent;
- the number of connections on a web browser is limited;
- mono-directional by nature, additional approaches are required for duplex synchronous operations.

The third limitation in the record must be addressed in web browser based applications, while it does not affect clients of different nature.

Nevertheless, in our context the opportunities offered by SSE outweight the weaknesses:

- SSE is based on HTTP, posing no issues with proxies;
- multiplexing is implemented in HTTP/2
- messages bear an *id*: the server is aware if the client misses one;
- data exchange requires a smaller number of connections;
- on failure, the *EventSource* reconnects;
- the connection stream is based on events, it is read only and goes from the server to the client;
- arbitrary events can be sent.

PUMA framework combines SSE with a channel based event streaming (*Nchan* [5]), thus enforcing the connection economy mentioned in the fourth item in the preceding enumeration. The hitherto introduced benefits offered by an SSE technology over WebSockets cover a fundamental aspect of the reliability essential. In particular, the a*ny place, any time and platform* principles are backed up by SSE. In combination with channels, the system gains scalability in reference to an arbitrarily high number of clients using the service at the same time.

Figure 1 is a representation of the client – server architecture based on Nginx, Nchan and the PUMA service.

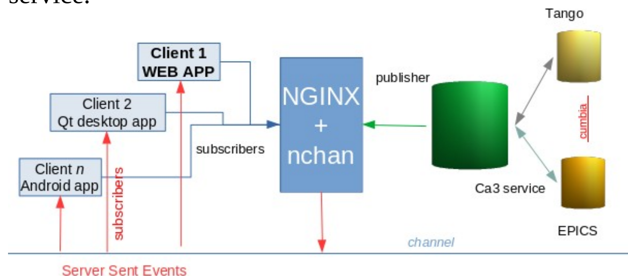


Figure 1: Nginx, Nchan and PUMA service use channels.

The next ingredient in our reliability recipe contributes to maintain the service available to clients even when parts of the system are unavailable. *Nginx* [6] comes into play as a balancer to distribute the load across several instances of the PUMA server and at the same time, through a PUMA *supervisor* service, to manage failover. Moreover, when Nginx and Nchan are combined with *Redis cluster* [7] (Remote Dictionary Server), channels attain high availability and failover capabilities [8].

## SECURITY

The goal is to realize a service and its context safe enough as to avoid tunnelling the traffic across a VPN. The latter implies additional software and configuration, hindering usability and reducing battery life, especially on mobile devices. At the moment of writing, the network architecture essential to achieve a proper level of security has still to be laid out. The observations made in the preceding section favour the choice of Nginx and HTTP over WebSockets as far as protection from DDoS (Distributed Denial of Service) is concerned. Nginx can be adjusted to limit the worker processes and connections, as well as the requests rate over time. Additionally, the number of connections that can be opened by a single client IP address can be curbed to protect the downstream PUMA services.

## SCALABILITY

A good infrastructure shall be designed with scalability in mind. Reliability and security will be inherently reinforced. Channels to which numerous clients tune in to receive messages are an obvious representative of scalability. Nginx combined with Nchan and Redis offer horizontal scalability to the PUMA service. Figure 2 shows this principle, while Fig. 3 illustrates the initial deployment of the PUMA service architecture at Elettra.

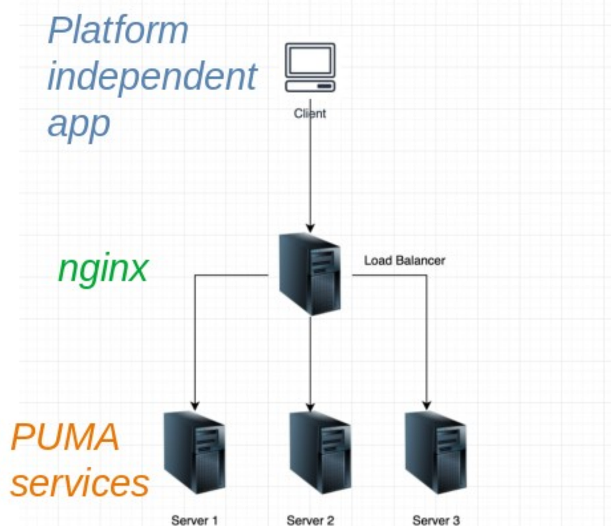


Figure 2: Horizontal scalability with Nginx.

## ACCESSIBILITY

The interaction with the service is performed through an open API relying on JSON (JavaScript Object Notation) both on the request and the reply sides. A simplified URL API for requests shall be included in a future release. The API serves the web, the mobile and the desktop applications.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

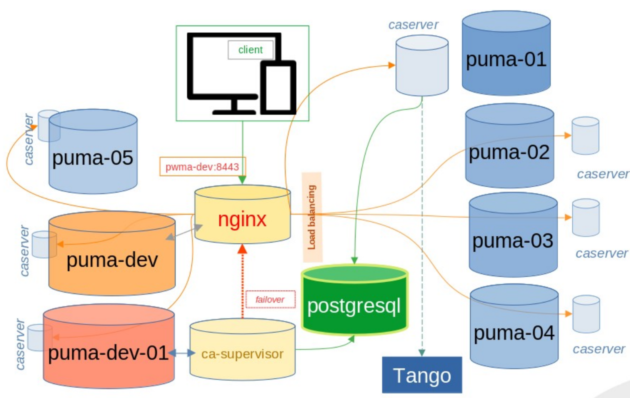


Figure 3: Horizontal scalability as deployed at Elettra.

Reliability, security, scalability and a generic API are the main principles of the PUMA service design rationale. In the following sections, the single components will be described in more detail.

## THE SERVICE

The *caserver* is the name given to the PUMA service mentioned several times in the preceding sections. Since PUMA is the evolution of an earlier project (dating to the year 2006) named Canone, *caserver* is a nickname referring to the Canone origins. As represented in Fig. 1 and Fig. 3, the client applications post their HTTPS requests to the Nginx web server. The latter forwards them to one or more *caserver* instances, working as a load balancer. The service is an object oriented, modular and plugin expandable C++ application relying on the *cumbia* and *cumbia-tango* [9] libraries to access the Tango control system in use at the Elettra synchrotron radiation facility. Once a request is received by the service, up to two actions take place:

- a synchronous reply is sent to the client immediately and normally carries either the result of the required operation enriched by additional information dependent on both the request and the underlying engine (e.g. from the Tango database) or an error message;
- if the operation is a subscription to the value of a source of data over time, updates are published on the channel the client subscribed to within the same request.

Clients, *caserver* and Nchan operate in a so called *pubsub* arrangement. The service publishes messages to channels using HTTP POST requests. Clients tune in to a channel to receive data through Server-Sent events.

The main type of requests that can be made to the server, herein named *methods*, are *read*, *conf*, *write* and *subscribe*. All of them are synchronous: an HTTP response is immediately sent back to the client. The subscribe method has the additional effect of data updates being delivered over a *pubsub* channel. A well behaved client shall pack together multiple requests into a single, more articulated, JSON string.

### Engines and Modules

The service code is organised into modules. The Tango control system in use at Elettra is accessed by a dedicated module. This can be replaced (for example by an EPICS engine) and the *caserver* adapts to another control system.

A module has an interface to process messages received from clients and through *factories* several implementations of a module can be installed. Thus we can let the *reader module* use both the Tango and the EPICS specific *reader* implementation. A module is registered on the server and several ones can be installed with a given priority. As a consequence, a message from a client can be processed in sequence until one (or no) module satisfies the request.

New modules can be written and added to the service, although the preferred extension strategy is through the plugin system.

### Cumbia

*Cumbia activities* use the standard C++ threads. Inspired by the Android *AsyncTask* interface [10] and paired with *cumbia timers* and *event loops* permit a multi threaded design of the service. Activities manage the transmission of data over the channels, the main socket server and plugins life cycles, the authorization process for write operations and so on. The *cumbia-tango* module is an abstraction layer facilitating the access to the namesake control system. Further implementation details are available on the project github page [11].

### Plugins

The service features can be extended by plugins. Plugins are dynamically loaded from a specific folder and can be disabled as simply as deleting the corresponding object from the file system. The *caserver* provides *hooks* to which the plugins can register in order to receive specific pieces of information. For example, hooks related to readings notify plugins upon new data, subscribe and unsubscribe operations. Likewise, the read module can switch from an active to an inactive state, a message can be delivered synchronously on a socket or entrusted to a channel. All these events can be monitored by plugins. The service *supervisor* [12], mentioned in the section dedicated to reliability, queries a PostgreSQL database to retrieve data about the health of the *caserver* instances in execution. Such records are written by the *ca-db-plugin* [13], hooked to the service socket receiver state change and the periodic *heartbeat* events. Another plugin [14] offers *introspection* capabilities straight through HTTP representing in JSON (JavaScript Object Notation) format diverse operating conditions of the server at a given moment (number of activities, threads, readings, and so on).

## THE SUPERVISOR

In the extent of the reliability of the PUMA service, and specifically for the accomplishment of the failover automation, the supervisor operates in conjunction with the *ca-db-plugin* discussed in the previous section. The

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

data recorded by the latter into a PostgreSQL database is analysed at regular intervals to determine whether every single *caserver* process is actually operating. Each instance is expected to register some kind of information periodically. If a unit fails, the supervisor undertakes recovery operations, redistributing the load formerly in charge of the missing service across the other processes online.

The architecture is designed so that if a policy of automatic and instantaneous restart of a broken down *caserver* is adopted, the restarted process regains control of its previous data *sources*[1]. The restart action must be undertaken promptly so that the supervisor is unaware of the failure.

The PostgreSQL database shall be centralised so that every instance of *caserver* can save its state regardless the host where it is executed.

## Qt CLIENTS AND LIBRARY

Alongside platform independent web interfaces, Qt [15] native applications written in C++ can be developed on top of a library named *cumbia-http*. They are expected to work on all platforms supported by the Qt framework itself. More generally, any application written on top of *cumbia* / Qt is able to run transparently either connecting to the native control system engine[2] or to the PUMA service. The application does not need to be recompiled in order to select the available or desired engine at startup.

### *Cumbia-http*

*Cumbia-http* is a *cumbia* module offered alongside engine specific ones such as *cumbia-tango* and *cumbia-epics*. Whilst the last two connect to the respective control systems natively and require complete access to the field and the software installation on the device as dependency, the HTTP module, connecting to the PUMA service, can be used by Qt applications from anywhere.

### *Cumbia Multi Engine Applications*

A *cumbia* Qt application is unaware of the module used to read from *sources* and write to *targets*. The same software can run in the control room, where the native control system is thoroughly accessible, and at office or at home on a computer laptop, wherefrom the control system is unreachable and the native libraries are not installed. A *cumbia* multi engine application loads the desired module at runtime and offers the same user experience and native performance both from the control room and from home. From the developer point of view, a control panel can be designed, built and tested with utmost efficiency from anywhere using the *caserver* and then deployed natively on the field. Developers and end users alike treasured *cumbia* multi engine while working from home during the COVID-19 pandemic.

[1] Cumbia and QTango adopt the term “source” to name the quantity data comes from, for example, a Tango attribute. Recovery pertains to the context of readers, thus every reader has a “source” of data.
[2] For example, Tango or EPICS native installations.

## WEB INTERFACES

Web interfaces are available on any device connected to the internet all over the world without any custom installation. Conversely, native applications benefit from having full knowledge of hardware. However, as soon as the quality and speed of web interfaces is close to native applications’, they are preferable because of their portability. An agnostic approach will implement all options letting the user choose the results rather than the technology. The same agnostic approach concerns the kind of devices supported; displays can be in a range about from 5 to 50 inches and can be touch or not; our goal is supporting as much devices as possible.

Mobile devices are usually strictly personal; this increases the opportunities of customization: apart from the style, the user can use predefined screens or create new ones arranging several items.

There is a particular screen called “starter”. Its purpose is to launch all other screens. Users can customize the starter screen either by a graphical tool or editing a JSON text.

Often the request of maintainability over years is in contrast with the choice of the most performing technologies available at the moment. So a compromise is necessary and the approach can be more aggressive or more conservative.

The choices made within the PUMA web interface development may be seen as conservative. We use JQuery [16] which enhances and simplifies JavaScript and has been quite stable over the past ten years. Bootstrap [17] is very helpful in being adaptive.

We had been using React for two years and decided to abandon it because in our particular use case the balance between the disadvantages (time consumed to keep the development environment updated) versus the advantages (modularity, state machine etc) was not satisfactory.

### *Adaptive Design*

Adaptiveness is mainly provided by Bootstrap (Fig. 4) and by flexbox [18], which is a standard of CSS3 (Cascading Style Sheet) [19].

Within the PUMA framework a web designer is available to produce device interfaces or simple dashboards, all responsive because based on flexbox. Screens are saved in JSON format in a PostgreSQL database. Any saved screen can be used as a component in a new screen. JSON screens are interpreted consistently also by a previous app [1].

Advanced users can import in PUMA any HTML (HyperText Markup Language) and SVG [20] file. We consider "advanced users" those with a good knowledge of HTML, JavaScript/JQuery and CSS, also the rest of this section is addressed mainly to advanced users. Other readers, if interested in such details, can find plenty of explanations on the web. We suggest MDN (Mozilla Developer Network).

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

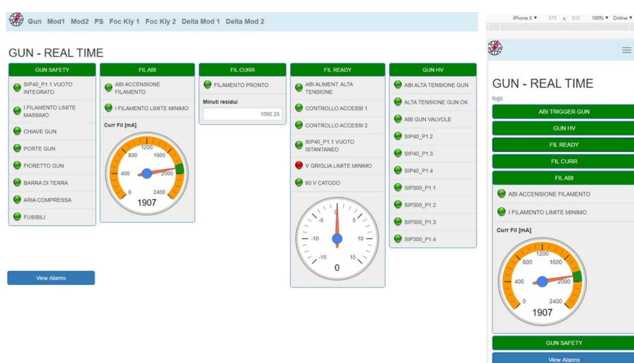


Figure 4: Desktop and mobile version of the same screen.

## *Vector Graphics*

SVG is used to produce 2D graphics, in particular large machine synoptics. In this way a very good flexibility in interaction with the user can be obtained; two different JavaScript libraries allowing zooming in and out by using the mouse wheel or by pinching are used. While zooming or moving around, some contents are added, removed or changed. In particular some variables are subscribed and unsubscribed on pan or zoom change so that only the visible ones (and a few side buffers) are continuously updated (Fig. 5). The JavaScript function which subscribes and unsubscribes variables, is called at a maximum rate of 2 times per second.

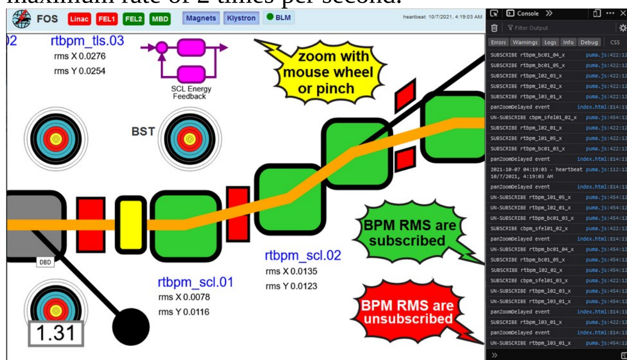


Figure 5: Some variables are subscribed and unsubscribed depending on the pan and zoom events.

In our working prototype this feature is completely transparent to the user, induced to believe that all variables are always updated without any loss in fluidity when moving the point of view. A demo is available on YouTube at the following URL: https://www.youtube.com/watch?v=z7FUDB7w2aw .

SVG is optimal for managing very complex 2D graphics and update any detail independently, but two main limitations subsist: it doesn't support 3D graphics and is not the best choice for displaying large quantities of data updated very quickly. WebGL [21] provides a very efficient solution to both problems.

PUMA provides an online HTML editor with a side preview which is constantly updated by the *keyup* event from the editor itself (Fig. 6).

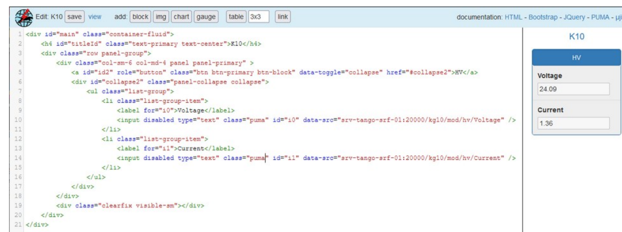

Figure 6: Web text editor with preview.

An authorized user can insert any HTML source which may include JavaScript, CSS and SVG. The connection to the control system can be implemented in any HTML or SVG tag by inserting a class "puma", a unique id and a custom data attribute [22] named "data-src". Only these steps are those necessary to connect to PUMA and there is no need to include any JavaScript. There is another optional attribute called "data-onupdate". Its value is the name of a JavaScript function that is triggered when new data is ready preventing the default update action. The "data-onupdate" JavaScript function is called with 3 parameters: the new value, the id of the calling tag (the same function can be used for more than one PUMA tag) and the source timestamp.

A one to one correspondence between tag and data-src is considered normal, though for exceptional cases this correspondence can be superseded. For example an array of booleans may be displayed as a table with green and red icons in the first column and labels in the second; such a table can be implemented using a hidden tag to associate a data-src to a data-onupdate function switching the colours of all icons. If a tag has to be updated according to the value of two or more variables, all variables should be associated to a different tag (hidden if not directly displayed) and to the same data-onupdate which provides a data synchronization mechanism and ultimately updates the multi-dependant tag.

The same screen can be reused in different contexts by inserting parameters into the data-src value.

SVG tags behave essentially in the same way as HTML tags; nevertheless there are a few differences to be taken cautiously in consideration, for example the JQuery [16] expressions .is(":hidden") and .is(":visible") work fine on HTML and SVG tags on Firefox, but they don't work on SVG tags on Chromium; instead the almost equivalent JQuery expression .css('display')=='none' works on SVG on all the most popular browsers (Fig 7).

It isn't very difficult to produce a thousand SVG elements image with a text editor, although a few graphical editors are available, for example Inkscape [23]. They tend to produce a lot of extra tags, that can be reduced by external tools which simplify the files.

In PUMA there are simplified versions of generic tools to browse all available control system variables, start and stop servers, monitor trends of a single variable. We also experimented with launching both web screens and desktop native applications from a browser. The second operation is explicitly forbidden by JavaScript. We overcome this limitation with a tiny bash script which executes the browser with an id as parameter, and waits in long polling mode [24] for a command to launch. Long

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-FRAR02

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

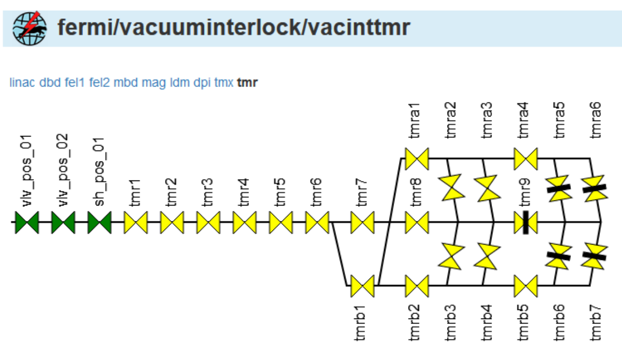


Figure 7: SVG on mobile.

polling is a cURL call to a particular page which receives an answer (resolves) only when there is a new event (a panel to be launched). Long polling is a technology much less advanced than WebSocket and SSE, but in this particular case it is still effective.

## TEST PHASE

The PUMA service, the web interfaces and the *cumbia* HTTP module have been used constantly and efficiently during the whole COVID-19 pandemic. A dedicated phase addressing stress and failover tests is currently underway. Web and Qt clients have been written to investigate and diagnose limits and points of failure. Examination results shall be stored into a database so that several parameters, such as synchronous reply time, speed and resource usage under pressure, can be monitored in the long run and improved across version updates. Initial experiments prove that a failing service process is detected by the supervisor and its load is redistributed amongst the online instances of the cluster.

In order to evaluate the overall performance of PUMA two web based tests have been developed: the first one subscribes and unsubscribes a configurable number of variables per second, the second one simulates panning and zooming of a real screen. As a result we reached about 300000 subscriptions per hour for several hours.

## CONCLUSIONS

The PUMA framework proved to be essential to develop and run graphical user interfaces from home during the COVID-19 pandemic, the other sole alternatives being either remote desktop solutions or X forwarding through SSH (Secure Shell). The first imply an amount of data compression degrading the graphics level of detail along with a moderately slow interaction, worsening to extremely sluggish in the second case. Actually, the authors' existing environment imposes two SSH tunnels over a VPN (Virtual Private Network).

The present day provides plentiful means in terms of computing speed and network bandwidth. Yet, they need to be economized. The data exchanged between the PUMA services and their clients involves only information relevant to the user: configuration and values from the control systems. In this sense, PUMA has an "ecological" approach: every bit is essential to knowledge, not even one carries redundant data and the user experience is always immediate because the resources of the device are operated natively. In case of portable devices, a remarkable battery saving shall be expected. Furthermore, due to the publisher – subscriber pattern over channels, the deployment of the PUMA framework in the control room on behalf of the native control system would definitely relieve the pressure on the devices and their servers. One need only think that umpteen readings of the same variable by PUMA clients translate into a single one to the control system.

The PUMA architecture has been planned with security, scalability and fail-safety in mind. On one side, the design contributes to administer the access and relieve the pressure on the control system. From the clients perspective, they are conferred scalable fast data exchange and lean on a failover redundant structure.

## REFERENCES

[1] L. Zambon, A. I. Bogani, S. Cleva, E. Coghetto, F. Lauro, "Web and multi-platform mobile app at Elettra", in *Proc.* 16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain, Oct. 2017, paper TUSH103, pp. 984-988. doi: 10.18429/JACoW-ICALEPCS2017-TUSH103

[2] https://en.wikipedia.org/wiki/WebSocket

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors

[4] https://en.wikipedia.org/wiki/Server-sent_events

[5] https://www.nchan.io

[6] https://www.nginx.com

[7] https://redis.io

[8] https://nchan.io/#high-availability

[9] https://github.com/ELETTRA-SincrotroneTrieste/cumbia-libs

[10] https://developer.android.com/reference/android/os/AsyncTask

[11] https://gitlab.elettra.eu/puma/server/canone3

[12] https://gitlab.elettra.eu/puma/server/ca-supervisor

[13] https://gitlab.elettra.eu/puma/server/ca3-db-plugin

[14] https://gitlab.elettra.eu/puma/server/ca-introspection-plugin

[15] https://www.qt.io/

[16] https://jquery.com/

[17] https://getbootstrap.com/

[18] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox

[19] https://developer.mozilla.org/en-US/docs/Web/CSS

[20] https://developer.mozilla.org/en-US/docs/Web/SVG

[21] https://www.khronos.org/webgl/

[22] https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/data-*

[23] https://inkscape.org/

[24] https://en.wikipedia.org/wiki/Push_technology/