Building a Ground M&C System with WebSocket – A New Way to Talk to an Antenna

Yi Wasser[1] and Dr. Armin Hauke[2]
German Space Operation Center, 82234 Weißling, Germany

Not until so long ago, a web-based application is basically not suitable for the concept of M&C, since a permanent, full-duplex link is essential for the real-time monitoring and commanding and the HTTP protocol is just not designed for such a purpose. However, the arrival of the WebSocket protocol opens a new door in front of us. This paper demonstrates how we could benefit from this newly emerged protocol to efficiently develop browser-based application as frontend for antenna monitoring and control system. Depending neither on the hardware, nor on the operating system being installed, such a frontend can run on any device or on any computer where a browser is available – assuming the browser supports the WebSocket protocol. This generality makes the task of monitoring and control mobile. Sitting at home, waiting for flight at airport, through a secured Internet connection, an operator can have real time information about the antenna anywhere anytime with his laptop, tablet or smartphone. If he is allowed, he can even command the antenna. The new technology and the widely spread mobile devices are changing our traditional way of monitoring and control. We shall take advantage of the new technology and be ready for the changes.

Nomenclature

M&C	=	A system to remotely monitor and control some given hardware equipment on ground
PDB	=	Parameter Database – a generic M&C framework developed at GSOC
WARP	=	Weilheim Antenna Remote Processing, Antenna M&C system used at Weilheim ground station
NEMO	=	Network Monitoring, M&C system used at Weilheim ground station
QT	=	A cross-platform framework for developing GUI application

I. Introduction

Developing cross-platform software is not a new topic, a number of companies and organizations have made great effort in this area during the last decades. In GSOC, one of our core software, a generic M&C framework named PDB – parameter database, is aimed to be platform independent at the very beginning of its birth. To a certain degree, we do manage to have it run on both Windows and Linux operating system. Things start getting difficult when complex user interface is involved. Having the performance in focus, we have chosen QT, a library written in C++, to develop PDB frontend. QT offers us a very good speed at graphics rendering and reaction time, and it runs smoothly on Linux. However, many fine details make the QT-based frontend difficult to run on Windows as smoothly as on Linux. Furthermore, the dependency on QT makes our life not easier. Each time when QT upgrades, we must on the one hand adapt PDB to the new changes, on the other hand to make sure that PDB still works on both Linux and Windows. Great effort and time are invested to keep the software QT compatible as well as platform independent instead of focusing on solving real problems.

The arrival of the WebSocket protocol starts to let us think an alternative way for writing cross-platform software, in our case especially the frontend segment. Many of the popular web browsers, such as Firefox, Chrome, and so on, support more than one operating system, and these browsers are talking more or less in the same language, namely HTML and JavaScript. Taking such an advantage of the browser, a web-based application is deep in its nature platform-independent. Considering the difficulty to keep our PDB running on both Windows and Linux, such a problem does no longer exist if the PDB frontend is web-based. Furthermore, a web-based application goes farther

[1] Software Engineer, Department "Communication and Groundstation", GSOC, DLR; yi.wasser@dlr.de
[2] Deputy Head of Department "Communication and Groundstation", GSOC, DLR; armin.hauke@dlr.de

on the way of independency – it is not only cross-platform, it is also cross-device in a certain way. With the wide-spread smartphone and tablets, software companies are working hard to keep their browsers mobile-aware. Nowadays we could hardly find a mobile device not having any browser installed. Together with the ever faster Internet infrastructure, it is interesting to observe how the new trend might change the way of the monitoring and control task which is traditionally restricted to control room with a user interface run on a physical computer.

Not in the least, with the rich features of HTML5 and CSS, today it is possible to script a professional-look GUI which is comparable to a classical GUI written in C++. Although it may still not be able to compete to the speed of a native GUI based on QT, it does have the advantage of being "no need for compilation, no need for installation". All these together make a browser-based frontend an attractive alternative.

II. Characteristics of WebSocket Protocol

Despite all the advantages we discussed above, a browser-based solution didn't draw much attention so far for a real-time system such as M&C. This is because HTTP protocol is based on a simple request/response model, server responses to request from client, but server cannot deliver any data to client without a request. The limitation of HTTP restricts applications which need two-way communications such as online game, instant message or real time system. The goal of the new WebSocket protocol is to try to solve the problem.

Sits direct over TCP, WebSocket is an application layer protocol which can be used by a web browser to communicate with a remote server. However, different to the classical HTTP protocol with request/response model, the WebSocket protocol provides a permanent connection between server and the browser and allows a two-way communications between them. Server and Browser can send data independently from the other anytime at will. The milestone has set free the potential power of the browser and it largely extents the spectrum of browser-based applications. It is not surprising that although being a young member of the TCP/IP protocol family, WebSocket protocol is rapidly gaining popularity during the last few years. At the time this paper is written, nearly all the main-stream browsers support WebSocket protocol.

A. WebSocket Protocol

The WebSocket protocol consists of two parts, the handshake and the data transfer[1]. Unlike many other protocols, these two parts have nothing in common. The handshake is a HTTP upgrade request in ASCII format whereas data transfer uses a binary frame.

- **Handshake**

  Being a TCP-based protocol, the WebSocket protocol is independent from HTTP protocol. The only relationship to HTTP protocol is the opening handshake. The client opens a connection by sending its handshake in a format of HTTP upgrade request. If the server accepts the request, it sends its handshake back to the client with HTTP status code "101". Any other status code indicates the handshake is not complete.

  The server must also confirm that it received the client's handshake by combining the information in client's "Sec-WebSocket-Key" field with the GUID, a SHA-1 hash of the combined information is stored to its "Sec-WebSocket-Accept" field and send back to the client.

  It is purposely designed that the WebSocket handshake takes the form of a HTTP request. In such a way, WebSocket protocol can continue use the HTTP port 80 and HTTPS port 443 to talk to the server, which means that the same port can be used to serve both HTTP client and WebSocket client. For many companies, these are almost the only ones that open. Moreover, WebSocket is a powerful protocol with growing extensions, the negotiation about how data should be transferred between the server and the client all takes place during the handshake. By using the well-defined HTTP upgrade request for this purpose, it avoids re-design the gear from scratch.

  After the handshake is completed, the data transfer can begin.

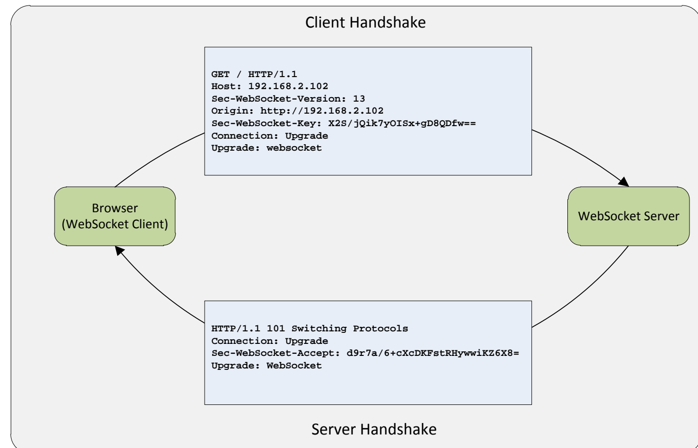

Figure 1: WebSocket Handshake

- **Data Transfer Frame**

  Frame is the smallest unit the WebSocket use for data transfer. The WebSocket frame has a variable header size between 2-14 bytes. The design is to provide a minimized framing to make the protocol message-oriented instead of stream-oriented (remember WebSocket protocol is on top of TCP, which is itself stream-oriented), and to distinguish between text message and binary message.

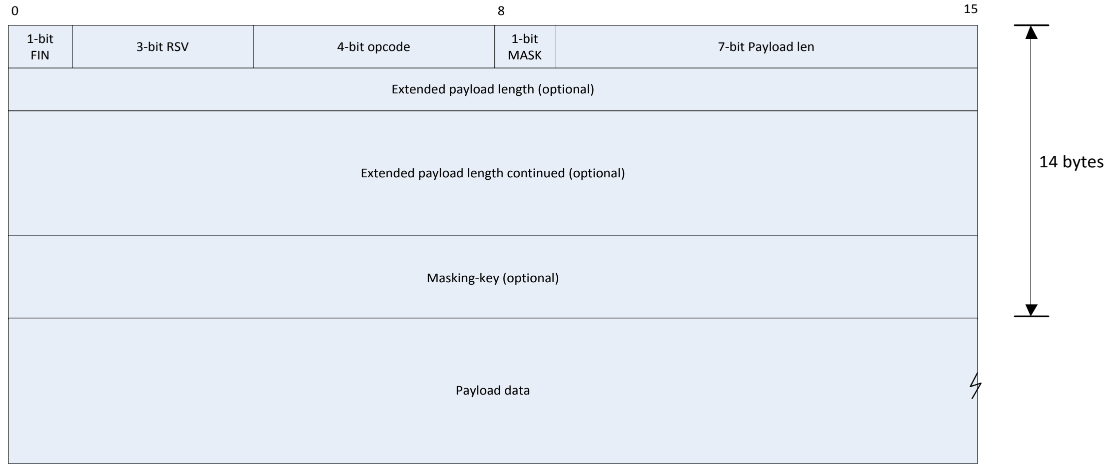

Figure 2: WebSocket Data Frame

FIN bit indicates if the frame is the final fragment of a message. RSV must be zero unless negotiated otherwise. Opcode is used to identify the message type:

- 0x0: it is a continuous frame
- 0x1: it is a text frame
- 0x2: it is a binary frame
- 0x8: it is a close frame
- 0x9: it is a ping frame
- 0xA: it is a pong frame

Mask bit indicates if the payload data is masked. It is always 0 if frame is sent from server to client, and it must be 1 if frame is sent from client to server. For M&C system, it can interpret as "monitoring data that come from server always has mask bit clear, commanding data from client (browser) has mask bit on, and its header contains the extra 4 bytes Masking-key.

7-bit payload length indicates the length of the payload data. If it has the value of

- 0-125: it is the payload length
- 126: the following two bytes (16bits integer) indicate the payload length
- 127: the following eight bytes (64bits integer) indicate the payload length

Masking-key only presents when the mask-bit is set to 1. All data sent from client to the server must have mask-bit on and the 32bits masking-key. The masking key is then used to mask the payload data, the payload length is not affected by the masking. The masking key shall be unpredictable, for each frame the client sent, a fresh masking key must be chosen from the allowed 32bit values. The main purpose of using masking key is to protect against attacks on intermediaries such as proxies.

B. Stand-alone

Except the opening handshake is a HTTP upgrade request, the WebSocket protocol is a complete TCP-based protocol on its own. It is not necessary that the server shall listen at port 80, and it is also not necessary that a client have to be a web browser. Although at the first sight it does not seem appealing since most of the WebSocket applications are browser-based, for real-time system, however, this property offers more flexibility for data processing.

Taken as an example, there are two kind of WebSocket clients in the M&C system we have built for project EDRS (we will discuss it in the next section), one kind is browser-based clients, the other is stand-alone clients. Once connected, server will deliver data to each of them. The browser-based clients consume the data and update its display accordingly. The stand-alone client, on the other hand, could do whatever necessary to the data, archiving, or

forwarding them further to other processing program. In our case, the stand-alone WebSocket client complements the browser by providing complex data processing which an ordinary browser cannot achieve.

C. Message-oriented

Data are transferred between WebSocket server and client as message. On the wire, a message can be split into one or more frames. A Frame can be further dissembled into more than one TCP segments (MTU is usually configured to 1500bytes on Ethernet interface). Unlike raw TCP connection, which is stream-oriented and the application must take care if data is complete and do the proper parse; the WebSocket protocol is message-oriented[4], all the dissemble and resemble complexity are hidden from application layer. For a large message, when it is dissembled during the transport, the browser will buffer the data until all frames are received, it will only raise an event to inform that data is arrived after the message is completely resembled to the one the server has sent. In another word, the receiver can always be sure that the message it received is exactly the one the sender has sent. It makes writing WebSocket application a lot easier than working directly on raw socket API.

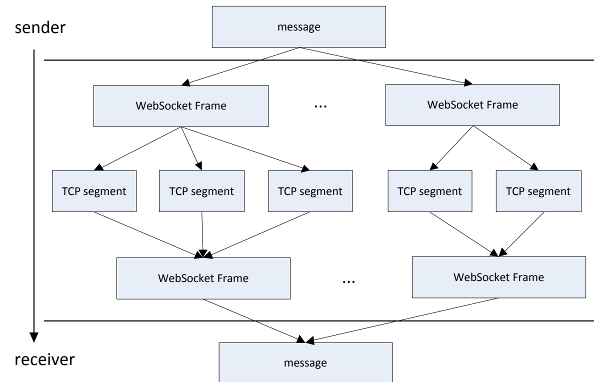

Figure 3: Message-Oriented

D. WebSocket Protocol Extension

As we discussed above, in order to keep the overhead small, the WebSocket data frame is designed to be very compact and the frame header can be as small as 2 bytes long. Consequently all the flexibility the protocol offers must therefore define in the opening handshake where the server and the client negotiate how they would like to transfer the data. The negotiation is often realized through the WebSocket extension mechanism and it proves to be a powerful way for adding new features to the protocol. One of the latest extensions is the data compression extension[2]. Figure 4 shows handshakes from two different browsers trying to connect to the same WebSocket server – a Firefox 24 running on Linux and a Firefox 43 running on Windows. Firefox 24 is an older browser which supports WebSocket protocol but not any of its extensions. Firefox 43, on the other hand, advertised that it supports the permessage-deflate extension in its opening handshake, and the server answered the advertisement by sending the same extension back in its handshake, confirming that it also supports the extension and the message will be sent in compression.

Figure 4: Data Compression Extension

Thus, the same server sent data to each of its client in different format, depending on the individual negotiation it made with each of the client at the opening handshake. In our test, the server is supposed to deliver text message continuously to the client when connected. Whereas Firefox 24 received its data in pure uncompressed ASCII format, Firefox 43 received the same data in compression. As we mentioned before, WebSocket is message-oriented, the protocol extension is transparent at the application level, so both browsers have the same update shown on their webpages, the application will not even notice whether data arrived in compression or not. What makes

them different is the consumption of bandwidth usage. With a continuous 40Mpbs data rate from the server, the network load showed ca. 40Mbps for the connection to Firefox24 as expected but less than 2Mbps for the connection to Firefox43, showing a data-rate savings about 95%. It proved that the per-message compression extension has significantly reduced the bandwidth consumption for message in text format.

III. Using WebSocket in Antenna M&C System

In the following section we will first give a brief overview of the existing M&C system used at Weilheim ground station, Germany. We then demonstrate how the system is extended using WebSocket. We conclude this section by showing a real life WebSocket application in project EDRS.

A. The Current M&C Environment in Weilheim

PDB is a generic framework developed at GSOC for the purpose of fulfilling monitoring and control task. A PDB instance is not a single-process application. It is a collection of programs including server, generator, consumer and processor, when necessary also logger and proxy. At run-time, monitoring information is first collected by the generators, the generators propagate it further to the server, the server in turn sends it to the consumer – the PDB GUI, and the processor is used to calculate derived parameter based on the raw information provided by the generators. PDB has a centralized architecture, where PDB server sits in the middle and the others connect themselves to the server via TCP links. All data flow must go through server, allowing no direct connection between any other two players.

At ground station Weilheim, there are two PDB-based applications, WARP and NEMO. WARP is the short name for Weilheim Antenna Remote Processing. It is a new generation antenna M&C system developed at GSOC in the recent years. Having the 11m KU band antenna run with WARP as the last, Weilheim ground station has completed the migration from the old Tigris M&C System to WARP. All eight Antenna located in Weilheim are now operated with this new M&C system. An in-depth discussion about WARP system is published in the previous paper[3].

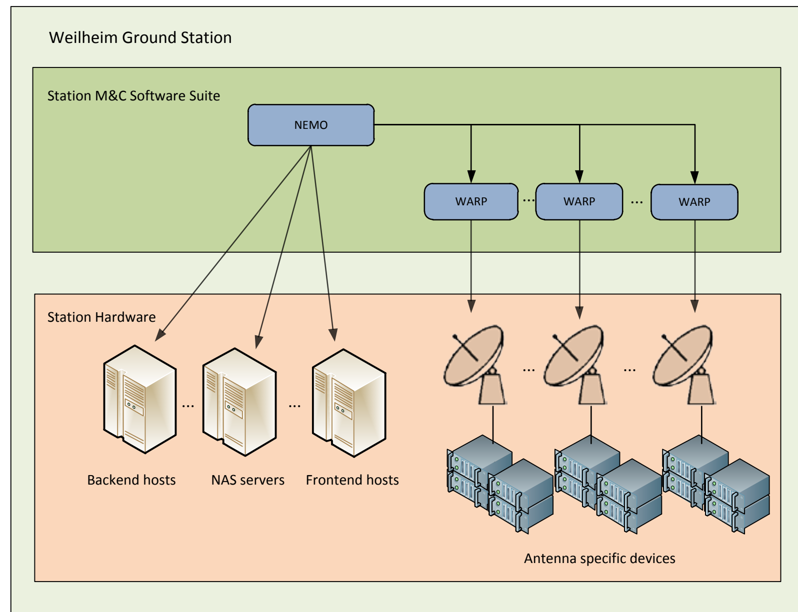

Figure 5: M&C System at Weilheim Ground station

NEMO stands for Network Monitoring. It is originally designed to monitor network device and services running on it. Due to its flexibility, it can be easily configured to perform a variety of monitoring tasks for different purposes. Together with WARP, they build up the complete M&C system currently used at Weilheim ground station.

Sharing the same PDB framework, both are being designed for the purpose of monitoring and control, WARP and NEMO differentiate themselves from the target system been monitored. WARP is targeting at antenna devices, NEMO, on the other hand, is being used to control the station hosts and the WARP system--- consider that WARP is such a complex system with around 30 - 50 processes, we need a way to watch and ensure that all the processes are running smoothly as well as the possibility to migrate the entire WARP system from one backend host to another backend host in the case of malfunction. NEMO is the one doing such a job.

B. PDB Extension with WebSocket

PDB-based applications such as WARP and NEMO are compound suite with many pieces of software where each piece has its dedicated role to play, e.g. some are generators, and some are consumers, and so on. Independent from their roles in the system, they all talk to the PDB server in the same language – the PDB protocol. At the beginning when we tried introducing WebSocket to the PDB system, our main concern is how to keep the existing

system intact at the same time to make it fit for the web browser. Being a distributing system where backend and frontend run at different hosts, the philosophy of PDB design is to separate the data processing from the data presentation as much as possible. Indeed, the current QT-based PDB frontend actually does nothing but simply display data, no further data processing takes place at PDB frontend. Taking advantage of the design, our intention is therefore to continue using the functionality the PDB backend provides and plugin a browser-based application as a new frontend.

In order to minimize the impact on the existing system, we decided that no changes shall be made at all within the PDB. Instead, a piece of new software is written. To make it dock to the current PDB system, this new software must understand both PDB protocol and PDB roles. In the end, a PDB proxy is used to link the PDB world to the new software.

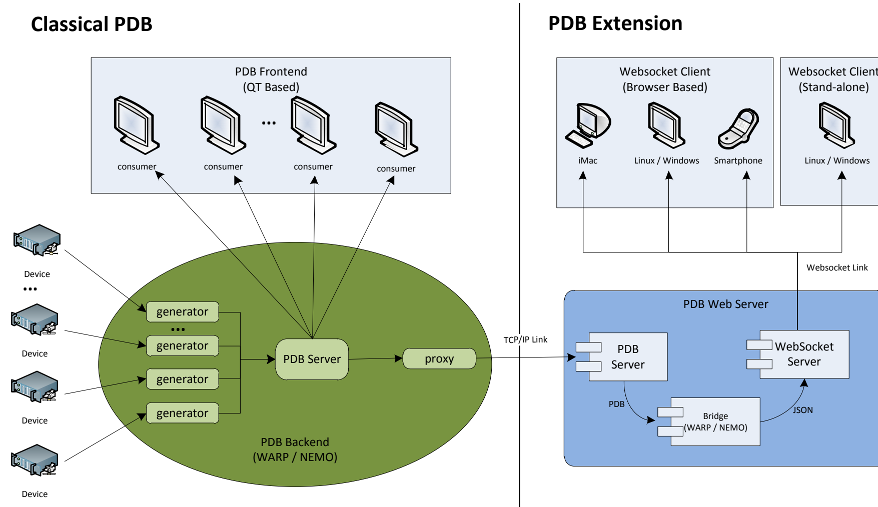

Figure 6: PDB Extension with WebSocket

A PDB proxy is originally designed to connect two PDB servers running on different sub-system, allowing one PDB server delivers all the information it possesses to the other PDB server at a higher level. In a complex domain with more than one sub-systems and each sub-system has its own PDB instance running, the PDB proxy provides a way to accumulate information from lower level PDB system and propagate them to a higher level PDB system. In our case, the proxy will forward all the information from the classical PDB server to the PDB web server – the new software. Commands received at the web server will go back to the PDB server through the same way in the opposite direction.

Our new PDB web server is a single piece of software with three components – a PDB server, a WebSocket server and a bridge in between. The PDB server component is 100% compatible to the classical PDB system, allowing the proxy connects to it via a standard TCP link and registers its role in a usual PDB way. Data come from the proxy are encoded in PDB protocol, a binary frame used for the communication within the PDB world. The PDB server component of our web server forwards the incoming PDB parameter to the bridge component, which in turn translates the PDB parameter into JSON format before it passes it further to the WebSocket server component. If there are any connected WebSocket clients, the WebSocket server will immediately deliver the JSON parameters to the clients without any delay.

JSON stands for JavaScript Object Notation. Similar to XML, it is a text format that can be used for data exchange and data storage. However, having less overhead compared to XML, JSON is lightweight and much simpler. Moreover, JSON is in pure JavaScript syntax, making it easier for machine to parse and understand, especially in the web world, where almost all browsers understand JavaScript. Recall that WebSocket protocol supports message in both text and binary format, we can of course pass the PDB parameters directly to the browser without translating them into JSON format, by doing so however we must expose the internal PDB protocol to the external side, which is not always suitable and desirable.

C. WebSocket application in EDRS Project

In 2015 a WebSocket based M&C frontend is developed at GSOC, Germany for the ESA project EDRS. EDRS is the short name for European Data Relay System. Via laser link, two relay satellites EDRS-A and EDRS-C in geostationary orbit shall transmit data from low-earth orbit satellites to the ground. With the relay satellites, EDRS will provide a data highway by freeing the low-earth orbit satellites from their limited visible time. EDRS-A was successfully launched early this year.

The ground segment of the EDRS project involves 4 KA band antennas locating at three different ground stations. Having the requirement for building a system with reliability as well as flexibility, the ground system is designed in such a way that

1. A central AMC (Antenna Management Center) shall be able to control all 4 antennas, in spite of their different locations. This shall be the nominal operational mode.

2. Each antenna shall also be accessible and operable with full functionality locally. This is to ensure that the antenna is still reachable in case failure occurred at central AMC.

To achieve the above goals, each station has its own dedicated WARP server(s), NEMO server and its data-storage unit to make a full-functional local M&C possible. In addition, station Weilheim is chosen as the central AMC, the other two NEMO instances at Harwell and Redu are connected to the central NEMO server at Weilheim via NEMO proxies. Monitoring information collected at local stations is sent to the central NEMO instance through these proxies.

EDRS is an ESA project with many participants sitting at different locations in Europe. Besides the central AMC in Weilheim, other participants also demand getting certain monitoring information from the ground stations. To ensure a successful operation as well as an efficient communication among different partners, therefore, in addition to the classical M&C system, GSOC agrees to provide a real-time monitoring service through a browser-based application. At each of the EDRS ground station, a PDB web server built with WebSocket technology will deliver monitoring information in real time. Wherever they are, all EDRS partners can now access the webpages either from within the EDRS network or through a secured connection from other network if is allowed. Figure 7 shows the main webpage provided by the web server at station Weilheim. Being the central AMC, the webpage is showing the information of all four KA band antennas from three ground stations. For security reason, however, it is not permitted to command the antenna from the web-based frontend, hence commanding through WebSocket link is completely disabled for project EDRS.

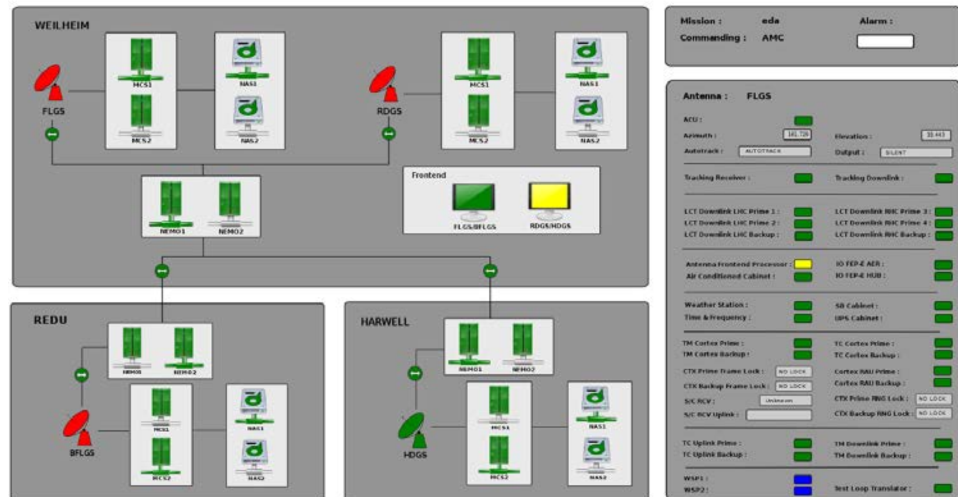

Figure 7: Browser-based M&C frontend in project EDRS

IV. Further Considerations of Browser-based Application

A. Cross-Browser Issue

No doubt it is a big advantage that a browser-based application is capable of running on any host regardless the installed operating system. However, the competition among different browser vendors has set barriers in the web world just the same. Each vendor speaks JavaScript in his own dialect and the same HTML and JavaScript code might show different results on different browsers. Deciding to have a browser-based PDB frontend as an alternative, inevitably we must also face the cross-browser issue. How we cope with it and to what extend must we go indeed depend on what we really want to achieve with our system.

To keep simple things simple, we try to avoid using any third-party plugins and restrict the frontend technology to HTML and JavaScript as long as they can meet our requirement. By doing so, we not only eliminate the client-side installation completely but also the dependency on third-party software. The frontend has the full freedom in its own hand and is able to run anywhere when a browser is available, which is otherwise not the case if the frontend depends on a certain version of JRE or a flash player, or so. Denying other web technology is for sure a pity, and it might not always be the correct decision. As we address earlier, it really depends on the purpose of the system. It is all possible that we might raise the restriction if the future requirement can otherwise not be achieved.

For most of the web developers, one of the main focuses is probably to have their webpage run on different browsers. The JavaScript interpreters from different browser vendors somehow make the goal difficult since the interpreters are not always speaking the same thing. Web developers nowadays often will adopt a middleware

library such as JQuery to overcome the difficulties. JQuery is a JavaScript library who supposes to understand the different dialects from the different browsers and hence make the cross-browser programming easier.

For our browser-based PDB frontend, however, the focus is different. Unlike the most web applications, the user domain of PDB frontend would always be very limited and project-specific, and there is never the intention to expose the PDB frontend to the Internet, which is freely accessible by people from all over the world. Instead, for a real-time M&C system, performance is more important than the generality on browsers. With a middleware library, we will win more browser support but compromise the performance because extra steps cost time. Although it is not always easy to find the balance between generality and performance, in our case we try to keep the browser-based PDB frontend free from any middleware software, in the meantime to give full support to at least two mainstream browsers such as Google Chrome and Mozilla Firefox.

B. Performance

One of the main performance concerns of the traditional web application is the response time of the browser. When user clicks on a link, a HTTP request is sent from the browser to the HTTP server, who in turn processes the request and send a response back to the browser. The big effort to improve the web application performance is hence to reduce this round trip time – the RTT time. WebSocket application such as PDB frontend, using a permanent link for data transfer, the general discussion on HTTP application do not always apply. Nevertheless, comparable to HTTP application, the response time of the PDB frontend depends mainly on two factors:

- The distance
- The processing time within the PDB system including server, generator, processor, etc.

The first factor relies on network infrastructure and the underlying bandwidth utility, and the second factor is PDB internal and it is sometime also device-dependent. Leaving the first factor aside, and let both QT-based PDB frontend and browse-based frontend physically located in the same network, the response time of the browser-based frontend is very much close to the QT-based frontend.

On the other hand, JavaScript is after all a scripting language, although it has been optimized to a great length in the recent years, it still cannot compete to the speed of a native language. With the increasing complexities of the GUI, webpage-toggling on the browser-based frontend becomes visibly slower than QT-based frontend, though the rendering speed of the JavaScript is still more than acceptable.

C. Security

Besides the regular connection using a “ws” URI scheme, WebSocket also provides a secured connection over TLS using a “wss” URI scheme. By default, port 443 is used for the secured connection. To achieve a better protection, however, only strong TLS algorithms shall be used by the WebSocket client, who does the negotiation with the server during the TLS handshake. General client-side authentication such as HTTP authentication or TLS authentication which applies to a generic HTTP server also applies to the WebSocket server.

WebSocket protocol proves the |Origin| field to prevent malicious JavaScript from running inside a trusted application. If the |Origin| field in the opening handshake is not what the server expects, it must reply with HTTP 403 Forbidden status code. However, this model only works if the WebSocket client is browser-based, stand-alone WebSocket client might still cheat the server with a faked |Origin| header field. Therefore, the server shall be in general skeptical about any input and perform additional security checks if necessary.

D. WebSocket vs. HTTP/2

In May 2015 IETF published the HTTP/2 standard[5], which includes many features that the WebSocket offers, e.g. binary frame, bidirectional communication through server-push – server can send several HTTP responses to a single HTTP request from the client, and so on. Naturally we will ask: would WebSocket be replaced by HTTP/2? Or would WebSocket and HTTP/2 exist parallel and complement each other? Honestly, HTTP/2 is still new and it is too early to see the future through.

One of the main goals of HTTP/2 is to improve the performance limitation of HTTP/1.1, but it is not going to change the HTTP semantics, all the HTTP methods, header fields remain the same. WebSocket protocol, on the other hand, is nearly a ‘raw’ protocol that is very close to TCP, as long as a WebSocket link is established, data flow over it is more natural and similar to the one based on a classical TCP-based application, there is no need to pretend to be a HTTP request or HTTP response, which would be true if we use HTTP/2 instead of WebSocket. For now therefore, WebSocket still seems more suitable for real-time system like M&C.

In fact, both protocols are pointing to the same direction, although we do not know how the future will be like, our system will nevertheless benefit from the latest development of the web technology.

V. Conclusion

It has been an exciting time since we started the adventure with WebSocket application two years ago. The changes what WebSocket protocol brings is almost revolutionary. Workaround solutions like long polling or HTTP streaming is no longer necessary, for the first time a web browser and server can communicate in two ways and transfer data independently from the other anytime they like. The WebSocket protocol has brought web browser new power and made it attractive to areas which are traditionally only suitable for classical standalone software. Together with the ever better IT infrastructure and the fast-growing mobile device industry, a change is taking place and it is exciting to see how it may influence the software development in space industry.

Our real-time monitoring and control system also benefits from this new protocol. At GSOC, we have extended our M&C system with a browser-based frontend. Based on the WebSocket technology, the browser-based frontend can provide real-time monitoring information from the station as well as accept commanding from operator. Such a frontend can run anywhere on any device where a browser is available. The monitoring and control task of a ground station is therefore no longer necessary restricted to the control room. It is thinkable useful for many situations, such as monitoring of an unmanned ground station or on-call service when an operator manager needs access to the station with his mobile phone or tablet.

In 2015 using the WebSocket protocol, we have developed a web-based M&C frontend for ESA project EDRS. Being a European cooperation project, EDRS involves participants from different countries where many participants demand to have certain monitoring information from the ground stations. With the browser-based solution, we successfully deliver a selected set of monitoring data which is accessible by all partners from either within the EDRS network or through a secured connection from other network. It again shows the power of a browser-based solution which is extremely simple and flexible. No software installation is necessary and what the partner needs to access the monitoring data is simply a browser.

The application of WebSocket protocol is fast growing, the work to bring classical standalone software web-aware has just begun, and we are still on the way searching for the best-suited rendering techniques for browser-based user GUI. To what direction this way leads us is unknown, but for sure it is exciting and hopefully promising.

Appendix

Acronym List

ESA European Space Agency
GSOC German Space Operations Center
GUI Graphical User Interface
TCP Transmission Control Protocol
M&C Monitoring- and Control-System
NEMO Network-Monitoring
WARP Weilheim Antenna Remote Processing, Weilheim's new M&C-system
JSON JavaScript Object Notation
Mbps Megabits per second
EDRS European Data Relay System
AMC Antenna Management Center
QT A cross-platform framework for developing GUI application
PDB Parameter Database – a generic M&C framework developed at GSOC
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
Tigris The predecessor of WARP system used in Weilheim
TLS Transport Layer Security
JRE Java Runtime Environment
CSS Cascading Style Sheets
IETF Internet Engineering Task Force
URI Uniform Resource Identifier
API Application Programming Interface

Acknowledgments

The Authors would like to thank all members of the software group at GSOC's department "Communication and Ground Station", who all contributed significantly to WARP and NEMO: Michael Beer, Rossella Falcone, Sylvain Gully, Thomas Ohmüller and Stefan Veit. Furthermore we wish to thank all the colleagues at GSOC from the

various departments, and the operation teams at Weilheim ground station. Very special thanks to Udo Häring, the founder of our PDB framework, whose experience has been invaluable to us. Finally we want to express our thanks to the respectful colleagues from SES TechCom S.A. and Airbus Defense and Space for the inspiring discussions and seamless cooperation, together we have brought success to the EDRS project.

References

[1]I. Fette, Google, Inc. and A. Melnikov, Isode Ltd., "The WebSocket Protocol", RFC 6455, December 2011.

[2]T. Yoshino, Google, Inc. “Compression Extensions for WebSocket”, RFC 7692, December 2015.

[3]A. Hauke and E. Barkasz, *“Multi-Mission Support with WARP”*, presented at SpaceOps 2012.

[4]Ilya Grigorik, “*High Performance Browser Networking”*, O'Reilly Media, 2013, Chap. 17.

[5]M.Belshe, BitGo and R. Peon, Google, Inc. and M. Thomson, Ed. Mozilla, “Hypertext Transfer Protocol Version 2 (HTTP/2)”, RFC 7540, May 2015