frontiers in Robotics and AI

CODE
published: 18 December 2017
doi: 10.3389/frobt.2017.00067

Connecting YARP to the Web with Yarp.js

*Carlo Ciliberto**

University College London, London, United Kingdom

We present yarp.js, a JavaScript framework enabling robotics networks to interface and interact with external devices by exploiting modern Web communication protocols. By connecting a YARP server module with a browser client on any external device, yarp.js allows to access on board sensors using standard Web APIs and stream the acquired data through the yarp.js network without the need for any installation. Communication between YARP modules and yarp.js clients is bi-directional, opening also the possibility for robotics applications to exploit the capabilities of modern browsers to process external data, such as speech synthesis, 3D data visualization, or video streaming to name a few. Yarp.js requires only a browser installed on the client device, allowing for fast and easy deployment of novel applications. The code and sample applications to get started with the proposed framework are available for the community at the yarp.js GitHub repository.

Keywords: yarp, robotics, iCub, web, websocket, Internet of things

OPEN ACCESS

Edited by:
Ugo Pattacini,
Fondazione Istituto Italiano di Technologia, Italy

Reviewed by:
Tobias Fischer,
Imperial College London, United Kingdom
Alessandro Roncone,
Yale University, United States
Lars Schillingmann,
Bielefeld University, Germany

***Correspondence:**
Carlo Ciliberto
c.ciliberto@ucl.ac.uk

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *19 August 2017*
Accepted: *22 November 2017*
Published: *18 December 2017*

Citation:
Ciliberto C (2017) Connecting YARP to the Web with Yarp.js. Front. Robot. AI 4:67. doi: 10.3389/frobt.2017.00067

1. INTRODUCTION

Smartphones, tablets, and wearable devices have drastically changed human communication and are nowadays a key component of everyday life, enabling humans to connect with each other and other devices in real time, forming a dense network of complex and frequent interactions. In this revolution, the Internet and Web technologies in general are playing the key role of a "lingua franca," establishing novel standards for modern communication protocols adopted by most platforms and operating systems. Indeed, as information technologies advance, we are steadily moving toward an "Internet of Things (IoT)" (Xia et al., 2012), where everyday object will be able to offer an interface for digital communication with humans and other devices.

In this scenario, robotic agents designed to operate in human environments will undoubtedly need to be well-versed in these new practices to seamlessly integrate within the IoT network. Towards this goal, in this paper we present *yarp.js*, a novel framework developed with the goal of connecting the YARP network with external devices using modern Internet protocols. YARP (Metta et al., 2006) is to date one of the most efficient and flexible robotics middlewares, adopted by many robotics laboratories worldwide and used as main communication tool for robotic platforms, such as the humanoid iCub (Metta et al., 2008) and R1 (Parmiggiani et al., 2017). In this sense, yarp.js provides a platform-independent approach to establish a two-way communication between YARP modules (e.g., the robot itself or other machines on the YARP network) and external systems whose only requirement is the ability to run an Internet browser.

Yarp.js decouples a server side, which must run on the YARP network, from a client side, which simply needs to be capable of tcp/ip communication with the server. The server side is built over a Node.js (Tilkov and Vinoski, 2010) abstraction layer wrapping the main YARP functionalities (e.g., opening/connecting ports, creating bottles or images, and writing/reading them via ports). Two

main benefits arise from this choice of server-side language: 1) the possibility to write YARP modules in Node.js and therefore, leverage the wide range of packages made available by the related community via the well-established Node Package Manager (NPM),[1] and 2) the event-based philosophy of Node.js offers a different perspective for programming the robot cognitive skills, possibly allowing for novel and more reactive behaviors. Yarp.js server is supported on OSX 10.11.6+ and Ubuntu 16.04+.

The client side of yarp.js consists of a pure JavaScript library and runs on both Google Chrome[2] and Firefox[3] browsers. Communication is performed across WebSockets, which allow for real-time exchange of data between the device on which the client is running and the server. Yarp.js endows both client and server with same functionalities, allowing also clients on external device to open and write/read on a YARP port. This is particularly useful to connect an external sensor, such as a smartphone microphone, inertial sensors, camera, etc., to the YARP network and allowing other modules to access its measurements. In this sense, yarp.js allows to effortlessly extend YARP functionalities to non-YARP devices by simply serving the required JavaScript library so that there is no need for custom installation, essentially making yarp.js automatically platform-independent on any browser-enabled device.

Yarp.js v1.0.0[4] is available for the community as a GitHub repository.[5] We have provided a number of examples for new users to get started with the proposed framework.

2. BACKGROUND AND MOTIVATIONS

We introduce the necessary background and motivations to understand the main contributions of yarp.js.

2.1. YARP

Yet Another Robot Platform (YARP) (Metta et al., 2006) is a framework developed to handle the low-level communication processes between different sensors, processors, and actuators in robotics applications. The main goal of YARP is to provide researchers and developers with a unifying cross-platform layer of communication in order to foster the diffusion and reproducibility of novel results in robotics. **Figure 1** (left half) reports a pictorial representation of a YARP network, where a number of computational nodes (gray circles) communicate with each other by leveraging on the abstraction layer offered by YARP (blue lines). In a spirit similar to YARP, several robotics frameworks have been proposed in the recent literature, such as Player (Gerkey et al., 2003), ROS (Quigley et al., 2009), OROCOS (Bruyninckx, 2001), MIRO (Utz et al., 2002), and LCM (Huang et al., 2010) to name a few. We refer to Fitzpatrick et al. (2014) for a discussion on the topic.

Unarguably, the most successful example of YARP application is the iCub (Metta et al., 2008), a humanoid robot adopted by more than 30 laboratories worldwide: Exploiting the flexibility of YARP functionalities, computational models developed by a number of different research groups to perform diverse tasks ranging from torque control (Fumagalli et al., 2010, 2012; Del Prete et al., 2012) to grasping (Gori et al., 2014), balancing (Pucci et al., 2016), visual attention (Ruesch et al., 2008), visual or haptic object recognition (Ciliberto et al., 2013; Higy et al., 2016), supervised learning (Gijsberts and Metta, 2011), can be combined on the same platform, enabling the robot with advanced cognitive capabilities such as in Ivaldi et al. (2013); Fischer and Demiris (2016); Morse and Cangelosi (2017).

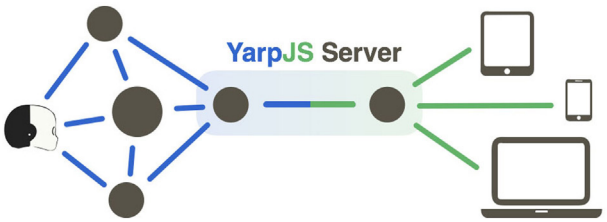

FIGURE 1 | Example of YARP network connected with non-YARP-capable devices on the Web via yarp.js.

2.2. Robots, Modern Web APIs, and Node.js

With the diffusion of lightweight portable devices, such as smartphones and tables, in recent years it has become a necessity for web applications to efficiently access and process information acquired from diverse sensors, such as microphones, embedded cameras, or inertial sensors. To this end, most modern browser has designed a wide range of APIs that allow accessing such resources across most devices, platforms, and operating systems. This has significantly fostered the deployment and diffusion of many novel applications capable of running natively in the browser, such as image object recognition,[6] GPS mapping and route planning,[7] speech-based assistants,[8] videoconferencing,[9] navigation in virtual reality environments[10] to name a few.

Making these capabilities available to a robot is clearly appealing and indeed the potential benefits of such interaction have been thoroughly investigated in the literature (Taylor and Wright, 1995; Hu et al., 2012; Kamei et al., 2012; Kehoe et al., 2015). However, robotics application typically requires real-time performance and deploying the necessary communication infrastructure to satisfy such requirements can be difficult or not possible due to compatibility issues. On the contrary, Web APIs are already designed to take care of the low-level communication with embedded sensors as well as the transmission of data across

[1] https://www.npmjs.com.
[2] https://www.google.com/chrome.
[3] https://www.mozilla.org.
[4] Yarp.js DOI: https://doi.org/10.5281/zenodo.1007786.
[5] https://github.com/robotology/yarp.js.
[6] https://www.clarifai.com/.
[7] maps.google.com.
[8] https://sdkcarlos.github.io/sites/artyom.html.
[9] https://appr.tc/.
[10] https://playcanv.as/p/sAsiDvtC/.

a network (i.e., the Internet). In this sense, the yarp.js framework proposed in this work acts as an intermediate layer allowing YARP and a browser to communicate, essentially "assimilating" non-YARP capable devices within the robot's network.

The above motivations are shared with recent work (Osentoski et al., 2011; Toris et al., 2015), where a JavaScript framework was developed to allow portable devices to communicate with the Robot Operating System (ROS) using Websockets and JavaScript. In this sense, the client side of yarp.js can be interpreted as the equivalent of the ros.js framework for the YARP environment, and one interesting byproduct of this work is the possibility to create applications that naturally bridge YARP and ROS frameworks by leveraging the two corresponding JavaScript libraries.

A second relevant byproduct of our work is the extension of standard YARP C++ routines to Node.js. This could be beneficial in developing robotics applications. Indeed, Node.js (and more generally JavaScript) is based on a system of callbacks that are activated when the corresponding registered event occurs (Tilkov and Vinoski, 2010). While this approach can be equivalently implemented in more traditional languages used in robotics (indeed its core is based on a C++ engine), Node.js encourages a programming style that is asynchronous by design and in this sense could be helpful in speeding-up the development of high-level applications in robotics without the need for ad-hoc careful synchronization between multiple modules and threads. As a practical example, consider the ActionsRenderingEngine (ARE)[11]: this iCub module manages a number of possible behaviors for the robot, combining both visual cues and motor actions and requires several threads (e.g., a vision thread, a motor thread, a visuo-motor thread, etc.) to be carefully synchronized in order to avoid low-level errors (e.g., concurrent memory access). This module would be significantly easier to develop (and read/debug), if written in an event-based language where the low-level details related to asynchrony are taken care of by design.

In the rest of this paper, we describe yarp.js and present a number of sample applications highlighting the potential benefits of the proposed framework in robotics.

3. SYSTEM OVERVIEW

Yarp.js is conceptually organized in two separate components: a server side, equipped with YARP communication capabilities and a client side, which is able to transmit and receive data from other nodes on the YARP network by exploiting the server side as a proxy. **Figure 1** reports a pictorial representation of a yarp.js network, where messages from non-YARP equipped devices (e.g., smartphones, tablets, etc.) are first sent via WebSockets (green lines) to the yarp.js server and then propagated through the YARP network (blue lines). The communication with YARP and WebSockets is bi-directional, allowing to transmit data from the network to the client.

The two-level structure of yarp.js is imposed by the nature of web technologies. Indeed, while on one hand browsers offer flexible cross-platform solutions to the deployment of novel applications, they also need to cope with extremely critical security issues (e.g., handling of passwords or sensitive data over the Internet). As a consequence, code running in the browser is allowed very limited interaction with the rest of the machine hosting it, let alone other machines on the same local network. In this sense, the server side of yarp.js can be interpreted as a standard YARP module that is also able to communicate with the browser, effectively acting as the missing link between the client and the YARP network.

As a final note, we care to point out that YARP is already equipped with basic HTTP communication functionalities[12] via Representation State Transfer (REST) (Fielding, 2000). However, RESTful interoperability is not suited for real-time two-way communication between server and client; one of the main motivations that led to the design of the WebSocket standard (Lubbers and Greco, 2010).

3.1. Server Side: YARP in Node.js

The server side of yarp.js is written in Node.js (Tilkov and Vinoski, 2010) and comprises two layers: first, a low-level library of C++ addons for Node.js [13] that allows to access and use YARP objects and functionalities from the Node.js environment. Second, a set of Node.js APIs offering easier management of the YARP addons (e.g., opening and connections of ports) as well as communication with client browsers. Below, we discuss these two layers in detail.

3.1.1. First Layer: Node.js Addons for YARP (Language C++ → Node.js)

This layer exposes the APIs to create the following YARP objects as Node.js objects: Bottle, Image, Sound, BufferedPort, RPCPort, and Network. It is written in C++ using the *Native Abstraction for Node.js* (NAN)[14] library and provides a set of Node.js wrappers for the corresponding YARP objects. As an example, below we report the minimal Node.js code to open a YARP port and write a Bottle on it using yarp.js.

```
var yarp = require('<yarp.js-folder>/build/Release/
Yarp JS');
                                          //get yarp.js
var yarp_net = new yarp.Network();
                              //get the YARP network


var port = new yarp.BufferedPortBottle();
                                    //create a port
port.open('/yarpjs/example');
                        //open it on the YARP network


var bottle = port.prepare();
                        //prepare the Bottle to write
bottle.fromString('hello yarp.js!');
                                   //fill the Bottle
port.write();
                          //write it over the network
```

[11] http://wiki.icub.org/brain/group__actionsRenderingEngine.html.

[12] http://www.yarp.it/yarp_http.html.

[13] https://nodejs.org/api/addons.html.

[14] https://github.com/nodejs/nan.

Note that these addons can be used as a standalone package to develop YARP modules in Node.js. This is extremely advantageous that it allows to effortlessly import Node.js packages from NPM to YARP applications. As a matter of fact, the second layer of yarp.js leverages a number of NPM packages to manage the communication between YARP and the browsers.

Callbacks. Callbacks can be provided dynamically to YARP objects. Below, we report the minimal code for reading from a port and printing the content of the received message on the terminal.

```
port.onRead(function(yarp_object){
   console.log('Message received: '+yarp_object.
   toString());
});
```

Extending yarp.js. By leveraging on the NAN abstraction layer, it is possible to easily extend yarp.js addons with new functionalities or create new ones wrapping other YARP objects. However, one aspect of this process deserves particular care, namely the conceptual separation between the threaded nature of YARP applications and the event-based philosophy of Node.js. To this end, we provide the `C++` class `YarpJS_Callback`, which stems a separate Node.js worker thread from the main one and runs the prescribed callback function when the required event occurs. This allows to dynamically provide callback functions to YARP objects as discussed above.

3.1.2. Second Layer: Yarp.js Server Manager (Language Node.js)

The second layer is a JavaScript module wrapping the yarp.js addons provided and offering (opinionated) management functionalities: 1) a *Port Manager* handling operations on the YARP network, such as opening/closing/connection of ports and 2) a *Browser Communicator* in charge of the communication with the client via WebSocket. In particular, this latter component interprets messages from the browser as either messages to be propagated to the network or as YARP commands that cannot be executed directly from the browser (e.g., opening a port).

Port Manager. This component exposes a set of functions meant to simplify the management of the YARP network from the Node.js module. Specifically, it allows to recover ports by name, connect two ports, and offer fallbacks in case of name conflicts (e.g., more clients trying to open the same port). It also manages to close all hanging objects when the Node.js module ends, cleaning memory and the YARP network. The code snippet below shows the difference in using the manager rather than the rawNode.js addons.

```
var yarp = require('<yarp.js-folder>/yarp.js');
                                          //get yarp.js
//no need to call YARP network

var port = new yarp.Port('bottle');
                                    //create a port
port.open('/yarpjs/example');
                     //open it on the YARP network

var bottle = port.prepare();
                     //prepare the Bottle to write
bottle.fromString('hello yarp.js!');
                                    //fill the Bottle
port.write();
                          //write it over the network
//alternatively, port.write('hello yarp.js!'); would do
the same
```

Browser Communicator. The browser communication component is based on the Socket.io package, which is designed to create webservers with robust WebSockets functionalities. To initialize the yarp.js manager it is sufficient to provide a Socket.io object to the `Browsercommunicator` method. All the communication with client browsers is then automatically handled. The following code makes use of the standard HTTP[15] and Express[16] packages to provide a minimal example on how to create a webserver offering yarp.js functionalities and listening on a port for incoming connections.

```
var http = require('http').Server(require
('express')());
                              //create the web server
var io = require('socket.io')(http);
                       //create the Socket.io object
http.listen(3000);
                //Run the server on locahlhost:3000

var yarp = require('<path to yarp.js>');
                         //get the yarp.js addons layer
yarp.browserCommunicator(io);
                    //Initialize the yarp.js manager
```

Once the yarp.js manager is initialized with Socket.io, all messages coming from the client side of yarp.js are automatically captured and processed by it. In Section 3.2, we list the main functionalities offered by using this intermediate layer.

This component is in charge of communicating to the Port Manager in which YARP ports are to be opened instead of the browser clients. In particular, whenever such a port reads a message in input, the Browser Communicator recovers it and pushes to the corresponding clients via WebSockets. This piping of the message is meant to create the "illusion" of having the browsers directly reading from the port. This is extremely helpful to develop code for the client side of yarp.js, however, it is important to keep in mind that for computationally intensive applications the Browser Communicator could become a bottleneck through which all messages from YARP to the clients need to flow. Clearly, this issue could be mitigated by having more than one yarp.js server module running on the network.

3.2. Client Side: YARP in the Browser (Language JavaScript)

The client side of yarp.js is a lightweight JavaScript library that leverages the browser implementation of Socket.io to communicate with the server side described in Section 3.1. The only requirement in this sense is for the browser to have WebSocket functionalities. Yarp.js can be initialized using the following code,

[15]https://nodejs.org/api/all.html.
[16]https://expressjs.com.

which here is assumed to be placed in the HTML page served to the browser:

```
<script src = "/socket.io/socket.io.js"></script>
<script src = "/yarp.js"></script>
<script>
   yarp.init(io());
   yarp.onInit(function(){
     //yarp.jscode
   });
</script>
```

The yarp.js manager on the client side offers the same APIs of the Port Manager on the server side. Specifically, it exposes a `Network` object that can be used to create new connections among ports on the YARP network and also a `Port` object that can be used to create new buffered ports and open them. As explained before, these operations cannot be performed directly by the client but are rather executed on the server side of yarp.js after receiving the corresponding message via WebSocket. Below, we report a code sample showing how to open a port and write/read messages which are automatically sent to the YARP network. All JavaScript code is to be assumed to be run within the onInit.

```
let port = new Port(port_type);
                    //port_type default: 'bottle'
port.open(port_name);
      //if the port does not exist the server open
                                          ones.

port.write([1,2,3]);
            //write a bottle containing 3 integers

port.onRead(function(yarp_object){
   console.log(yarp_object.toString());
});
```

The functionalities of yarp.js in the browser allow to easily develop and deploy YARP applications on the hosting device as we describe in the following.

4. APPLICATIONS

On the yarp.js repository we provide a number of sample applications to get new users started with the proposed framework. They are organized in a single bundle[17] that can be run by executing the code

```
$>node examples/examples.js
```

on the machine, where the server side of yarp.js is installed. Then, from any other device on the same local network, the example bundle can be accessed by navigating with Firefox on Google Chrome browser on http://<ip.of.yarpjs.machineer:3000.

Figure 2 shows how examples are rendered to the user.

4.1. Reading and Transmitting Inertial Data

This application shows how sensors on external devices (e.g., where YARP is not installed) can be accessed from the YARP network. We make use of the Web API[18] to read from the inertial sensor of a smartphone and stream it through a port.

```
window.addEventListener("deviceorientation",
function(event){
   port_orientation_out.write([event.alpha,event.
   beta,event.gamma]);
}, true);
```

Another client can read the inertial data streamed through the network and visualize the corresponding 3D orientation of the device using WebGL functionalities (a topic addressed in more detail in Section 4.4). **Figure 2B** shows a snapshot of this application.

4.2. Speech Recognition and Synthesis

This application uses the Web Speech API[19] for speech recognition and synthesis. To simplify the access to the Web Speech API yarp.js provide a synthetizer

```
yarpSpeakPort.onRead(function(msg){
   yarp.Synthetizer.speak(msg);
});
```

which allows to speak aloud text, read from a YARP port and Recognizer module

```
yarp.Recognizer.enableAutorestart();
    \\starts the speech recognition module
yarp.Recognizer.addEventListener('yarp speech finished',
function (e) {
   yarpSpeechRecPort.write(e.detail[0].transcript);
}, false);
```

which recognizes human speech from the embedded microphone and emits the event "`yarp speech finished`" as soon as the Web Speech API consider the audio signal to have terminated.

4.3. Stream Video (a "yarpview" in the Browser)

YARP images can be read from a port on the yarp.js client and visualized in the browser. Ideally, the WebRTC protocol (Johnston and Burnett, 2012) should be adopted for the transmission of large amounts of data over UDP. Unfortunately, to this date a standard solution for server-to-browser WebRTC communication does not exist. To reduce the burden on the server/client communication, we compress the images in either PNG or JPEG before sending them over WebSockets.

Images can be then visualized in a `<canvas>` HTML element using the following code.

```
let canvas = document.getElementByTag('canvas');
let img = new Image();
```

[17] https://github.com/robotology/yarp.js/tree/master/examples.

[18] https://developer.mozilla.org/en-US/docs/Web/API/Window/ondeviceorientation.

[19] https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html.

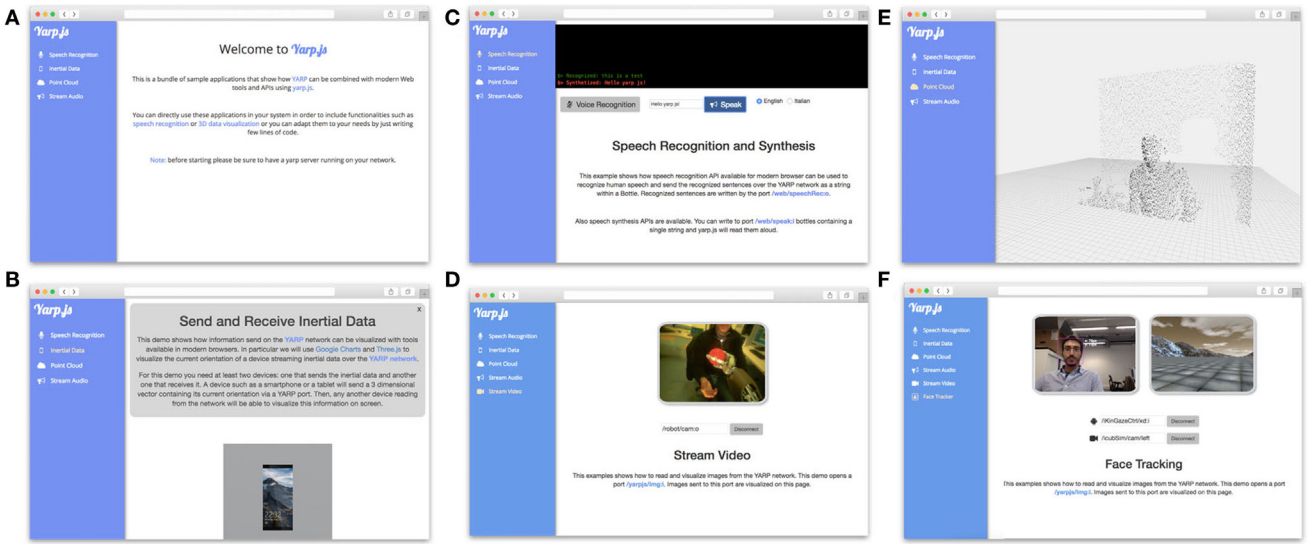

FIGURE 2 | The bundle of yarp.js application examples available on the project repository. **(A)** Landing page of the examples bundle. **(B)** Reading and transmitting inertial data (Section 4.1). **(C)** Speech Recognition and Synthesis (Section 4.2). **(D)** Visualizing Yarp Images in the Browser (Section 4.3). **(E)** 3D Visualization of YARP data (Section 4.4). **(F)** Face tracking for robot teleoperation 4.5. All depicted individuals provided their consent for the publication of this image.

```
port_video_in.onRead(function(yarp_img){
    img.src = yarp.getImageSrc(yarp_img.compres-
    sion_type,yarp_img.buffer);
    canvas.getContext('2d').drawImage(img,0,0);
});
```

Figure 2D shows the yarp.js acting as a `yarpviewer`[20] in the browser.

4.4. 3D Visualization of YARP Data

WebGL[21] is a standard Web API providing 3D graphics functionalities on the browser. Exploiting the three.js-WebGL library[22], we built a simple application to visualize point clouds read from YARP ports received as Bottles of one or more 3D array which are interpreted as 3D cordinates and rendered in a navigable virtual scene (**Figure 2E**).

Note that allowing the browser to directly interact with the graphic card of the hosting machine opens a wide range of possibilities. Indeed, recently there has been interest in developing applications to run Deep Learning models directly in the browser.[23]

4.5. Teleoperation with Face Tracking

We conclude by proposing a teleoperation application, where a face tracker running in the browser is used to actively control the head of the iCub robot. We used the Tracker.js[24] library to capture images from the device camera, detect the face of a user, and obtain the (u,v) position of the corresponding rectangle in the image. Then, the position was translated to a 3D point

$$(x, y, z) = (-1, u/w - 0.5, v/h - 0.3)$$

which is sent to the `/xd:i` port of the `iKinGazeCtrl`[25] (Roncone et al., 2016) to control the gaze of the robot to point toward it. See the following code.

```
let tracker = new tracking.Object Tracker
('face');
tracker.on('track', function(event){
    let rect = event.data[0];
    let u = rect.x + rect.width/2;
    let v = rect.y + rect.height/2;

gazePort.write([-1, (u/w - 0.5), (v/h - 0.3)]);
});
```

where w and h, respectively denote the height and width of the device camera. **Figure 2F** shows an example of this application, where images streamed from the robot camera are send back to the browser are described in Section 4.3.

5. CONCLUSION

We have presented yarp.js, a JavaScript framework to enable YARP-based robotics systems with modern Web APIs functionalities. Yarp.js allows modules running on the YARP network to access sensors information on devices that are not equipped with the YARP communication layer by exploiting WebSocket communication. By leveraging on Web technologies, applications

[20] http://www.yarp.it/yarpview.html.
[21] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.
[22] https://threejs.org.
[23] http://cs.stanford.edu/people/karpathy/convnetjs, https://github.com/transcranial/keras-js, https://pair-code.github.io/deeplearnjs/, https://tenso.rs.
[24] https://trackingjs.com.
[25] http://wiki.icub.org/brain/group__iKinGazeCtrl.html.

based on yarp.js are easy to deploy and develop. We have presented a number of applications showing the benefit of the proposed approach.

Yarp.js is easy to extend and a main challenge in the future will be to enrich its capabilities with WebRTC functionalities, which would be the natural solution to the issues related to the transmission of large amounts of data between server and client. We will investigate this direction in future work.

AUTHOR CONTRIBUTIONS

The author confirms being the sole contributor of this work and approved it for publication.

FUNDING

This work was supported by EPSRC grant EP/P009069/1.

REFERENCES

Bruyninckx, H. (2001). "Open robot control software: the Orocos project," in *Proceedings 2001 IEEE International Conference on Robotics and Automation, ICRA 2001*, Vol. 3 (Seoul: IEEE), 2523–2528.

Ciliberto, C., Fanello, S. R., Santoro, M., Natale, L., Metta, G., and Rosasco, L. (2013). "On the impact of learning hierarchical representations for visual recognition in robotics," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Tokyo: IEEE), 3759–3764.

Del Prete, A., Nori, F., Metta, G., and Natale, L. (2012). "Control of contact forces: the role of tactile feedback for contact localization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (San Francisco: IEEE), 4048–4053.

Fielding, R. (2000). "Representational state transfer," in *Architectural Styles and the Design of Netowork-Based Software Architecture*, 76–85.

Fischer, T., and Demiris, Y. (2016). "Markerless perspective taking for humanoid robots in unconstrained environments," in *2016 IEEE International Conference on Robotics and Automation (ICRA)* (Stockholm: IEEE), 3309–3316.

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Software Eng. Robot.* 5, 42–49.

Fumagalli, M., Ivaldi, S., Randazzo, M., Natale, L., Metta, G., Sandini, G., et al. (2012). Force feedback exploiting tactile and proximal force/torque sensing. *Auton. Robots* 33, 381–398. doi:10.1007/s10514-012-9291-2

Fumagalli, M., Randazzo, M., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "Exploiting proximal f/t measurements for the icub active compliance," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 1870–1876.

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). "The player/stage project: tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, Coimbra, Vol. 1, 317–323.

Gijsberts, A., and Metta, G. (2011). "Incremental learning of robot dynamics using random features," in *2011 IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai: IEEE), 951–956.

Gori, I., Pattacini, U., Tikhanoff, V., and Metta, G. (2014). "Three-finger precision grasp on incomplete 3d point clouds," in *2014 IEEE International Conference on Robotics and Automation (ICRA)* (Hong Kong: IEEE), 5366–5373.

Higy, B., Ciliberto, C., Rosasco, L., and Natale, L. (2016). "Combining sensory modalities and exploratory procedures to improve haptic object recognition in robotics," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun: IEEE), 117–124.

Hu, G., Tay, W. P., and Wen, Y. (2012). Cloud robotics: architecture, challenges and applications. *IEEE Netw.* 26. doi:10.1109/MNET.2012.6201212

Huang, A. S., Olson, E., and Moore, D. C. (2010). "LCM: lightweight communications and marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 4057–4062.

Ivaldi, S., Lyubova, N., Droniou, A., Gerardeaux-Viret, D., Filliat, D., Padois, V., et al. (2013). "Learning to recognize objects through curiosity-driven manipulation with the icub humanoid robot," in *2013 IEEE Third Joint International Conference on Development and Learning and Epigenetic Robotics (ICDL)* (Osaka: IEEE), 1–8.

Johnston, A. B., and Burnett, D. C. (2012). *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC.

Kamei, K., Nishio, S., Hagita, N., and Sato, M. (2012). Cloud networked robotics. *IEEE Netw.* 26. doi:10.1109/MNET.2012.6201213

Kehoe, B., Patil, S., Abbeel, P., and Goldberg, K. (2015). A survey of research on cloud robotics and automation. *IEEE Trans. Autom. Sci. Eng.* 12, 398–409. doi:10.1109/TASE.2014.2376492

Lubbers, P., and Greco, F. (2010). HTML5 web sockets: a quantum leap in scalability for the web. *SOA World Mag.* 1.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 8. doi:10.5772/5761

Metta, G., Sandini, G., Vernon, D., Natale, L., and Nori, F. (2008). "The ICUB humanoid robot: an open platform for research in embodied cognition," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems* (Gaithersburg: ACM), 50–56.

Morse, A. F., and Cangelosi, A. (2017). Why are there developmental stages in language learning? a developmental robotics model of language development. *Cogn. Sci.* 41, 32–51. doi:10.1111/cogs.12390

Osentoski, S., Jay, G., Crick, C., Pitzer, B., DuHadway, C., and Jenkins, O. C. (2011). "Robots as web services: reproducible experimentation and application development using ROSJS," in *2011 IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai: IEEE), 6078–6083.

Parmiggiani, A., Fiorio, L., Scalzo, A., Sureshbabu, A. V., Randazzo, M., Maggiali, M., et al. (2017). "The design and validation of the r1 personal humanoid," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Vancouver: IEEE), 2591–2598.

Pucci, D., Romano, F., Traversaro, S., and Nori, F. (2016). "Highly dynamic balancing via force control," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun: IEEE), 141–141.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Vol. 3 (Kobe), 5.

Roncone, A., Pattacini, U., Metta, G., and Natale, L. (2016). "A cartesian 6-dof gaze controller for humanoid robots," in *Proceedings of Robotics: Science and Systems*, AnnArbor, MI. doi:10.15607/RSS.2016.XII.022

Ruesch, J., Lopes, M., Bernardino, A., Hornstein, J., Santos-Victor, J., and Pfeifer, R. (2008). "Multimodal saliency-based bottom-up attention a framework for the humanoid robot icub," in *2008 IEEE International Conference on Robotics and Automation, ICRA 2008* (Pasadena: IEEE), 962–967.

Taylor, A. L., and Wright, J. T. (1995). "A telerobot on the world wide web," in *In National Conference of the Australian Robot Association* (Melbourne: Citeseer).

Tilkov, S., and Vinoski, S. (2010). Node. js: using javascript to build high-performance network programs. *IEEE Internet Comput.* 14, 80–83. doi:10.1109/MIC.2010.145

Toris, R., Kammerl, J., Lu, D. V., Lee, J., Jenkins, O. C., Osentoski, S., et al. (2015). "Robot web tools: efficient messaging for cloud robotics," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Chicago: IEEE), 4530–4537.

Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G. (2002). Miro-middleware for mobile robot applications. *IEEE Trans. Robot. Autom.* 18, 493–497. doi:10.1109/TRA.2002.802930

Xia, F., Yang, L. T., Wang, L., and Vinel, A. (2012). Internet of things. *Int. J. Commun. Syst.* 25, 1101. doi:10.1002/dac.2417

Conflict of Interest Statement: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.