Framework for Rapid Prototyping of Distributed IoT Applications Powered by WebRTC

Jan Janak and Henning Schulzrinne
Department of Computer Science, Columbia University, New York, NY, USA
Email: {janakj,hgs}@cs.columbia.edu

***Abstract*—We argue that the future of Internet of Things (IoT) systems, especially when it comes to privacy and security, lies in distributed IoT applications. Distributed IoT applications implement a model we call "computation follows data". In this model, application modules are deployed directly on IoT devices that produce sensitive data. Developing such applications is, however, not easy. Based on our own experience, we identify the lack of a rapid prototyping and development environment as the biggest challenge in the development process. In this paper, we describe a framework that aims to help simplify the process. The framework provides a web-based user interface with interactive virtual IO ports and a runtime environment for IoT device emulation. We also describe a network architecture with support for WebRTC-based direct device-to-device connections. The network architecture allows experimentation with an entire network of IoT devices, both emulated and physical.**

***Index Terms*—WebRTC, Internet of Things**

I. Introduction

The idea of connecting everyday objects and devices to the Internet, also known as the Internet of Things (IoT), is certainly not new. Until recently, this technology area was mainly of interest to researchers, tinkerers, and technology enthusiasts. The prevalence of commodity embedded platforms, ubiquitous networking, and affordable cloud infrastructure has generated interest in IoT systems and applications from a much wider audience.

The current generation of IoT systems can be characterized as "data follows computation". An IoT application or service is typically deployed to cloud infrastructure. Individual devices transfer data to the application in the cloud where it is processed. This approach has, of course, its advantages and drawbacks. Developing IoT applications for standardized cloud infrastructure is much easier and faster than developing software for embedded devices. The developer can use high-level languages and convenience libraries without the need to worry about platform limitations. The drawbacks include bandwidth and latency limitations, loss of privacy and security when sensitive sensor data is transferred to the cloud, and a single point of failure.

There appears to be consensus in the community that a more distributed architectural style for IoT applications might be needed. Applications written in the distributed style can be characterized as "computation follows data". Such applications typically provide components that can be deployed close to the origin of the processed data, i.e., at a nearby gateway node or onto the sensor node itself. Notable advantages of this architectural style include improved privacy and security guarantees, as well as support for additional "closed-loop" application scenarios. Unfortunately, developing and deploying reliable distributed IoT applications is difficult.

While there have been efforts to move to a more distributed IoT application style, the effort is still in its infancy. Based on our own practical experience, we believe that the biggest obstacle that hinders wider adoption is the lack of a good framework for rapid experimentation and prototyping with distributed IoT applications without using real hardware. Various methods of emulation, simulation, and Computer Aided Design (CAD) are common in other engineering disciplines. Yet, in the IoT space, one is often forced to develop on real embedded hardware right from the start.

In this paper we present some of our efforts towards building a web-based application framework for rapid development of distributed IoT applications. We first present the most important design considerations and requirements. We then describe a network architecture and a web application framework for rapid prototyping of distributed IoT applications with local computation and direct device-to-device connections. We also describe one particular approach to emulate applications for real physical devices, such as the Arduino, with minimal changes.

The rest of the paper is organized as follows. Section II presents the most important requirements and design goals for the framework. In section III we describe the network architecture required by the web application framework. Section IV discusses the design of the web application. In section V we provide an overview of the implementation details. Section VI discusses existing prior work and alternative approaches. Finally, we conclude in section VII.

II. Design Goals & Requirements

In this section we present some of the design goals for the framework. The list is not exhaustive, but includes the most important design goals and decisions.

a) Direct device-to-device communication: Where possible, the framework should make it possible for two devices (virtual or physical) to establish a direct communication session. The devices can then run any application-level specific protocol over the session. Direct device-to-device communication is important for various "closed loop" IoT scenarios where the state of device A directly influences the state of another device B. Direct communication also minimizes

latency and is an important building block for privacy sensitive IoT applications.

b) Prototyping without hardware: One of the biggest obstacles on the path to more distributed IoT applications is the difficulty of setting up and managing heterogeneous networks of devices. A distributed IoT application typically consists of a number of independent sequential processes to be deployed across nodes in a network. These processes then perform local computation on the node and setup communication sessions with other nodes. In a network of heterogeneous devices, each node may provide a different environment which further complicates the problem. We wanted to take this into consideration while designing the framework and make it possible to work with emulated (virtualized) devices. Even low-fidelity device virtualization can significantly simplify the development of an initial application prototype.

c) First-class support for browser devices: The increasing popularity and sophistication of web-based applications and services pushes the envelope of the browser. Over time the web browser application has turned from a simple formatter of HTML pages into a fully featured application platform. Modern browsers support a variety of technologies including a first class programming language, a protocol stack for real-time communications, a variety of transports, and support for persistent data storage. On many mobile devices the web browser is de-facto the shell and the single most important application the user will ever use. For these reasons, we would like to make mobile browser-first devices first class citizens in a network of IoT devices. Mobile devices (e.g., tablets) could be used as versatile, programmable user interface components in an IoT system.

d) Component-based design: We decided to design the whole framework as extensible from the start. It should be possible for independent third party developers to provide extension packages with additional UI elements (widgets). The system framework should let user import such packages independently, without the need to wait for an approval from central authority.

III. Network Architecture

Consider a simple network of IoT devices as shown in fig. 1. That network includes four devices nodes labeled A through D. The devices are heterogeneous and each node represents a different type of device found in real-world systems. In addition to device nodes, the diagram also includes a minimum set of services necessary for the network to function. The figure shows two types of network sessions: the thin lines represent WebSocket signaling connections, the thick lines represent dynamically negotiated WebRTC data channels.

Each of the four device nodes represents a different device type. Device A represents mobile devices equipped with a JavaScript-enabled browser such as tablets or smartphones. In a real-world system such devices could be used to implement interactive control panels and dashboards with a web-based user interface. We assume the browser on the device supports the WebRTC protocol stack and supports the Data Channel

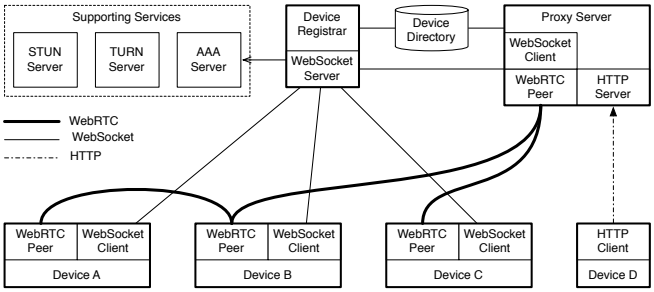

Figure 1. An example network architecture with heterogeneous IoT devices and supporting network services. The network uses WebRTC data channels to enable direct device-to-device communication where possible. Where not possible, device-to-device communication is relayed by a TURN server [1] or a HTTP-RTCWeb proxy. Devices A to C represent WebRTC enabled devices of various form factors, e.g., tablet with a browser, embedded device, or a server. Device D represents an IoT device without support for WebRTC protocols.

API. Device B represents a class of larger embedded IoT devices, typically Linux powered, capable of running a WebRTC compatible protocol stack. Pretty much any Linux-powered embedded device would fall into this category. The WebRTC protocol stack for such devices is available in form of native libraries. Device C represents a general purpose computational node, e.g., a server, that provides a virtualization environment for IoT devices. The node may be running a number of virtual IoT devices, using Qemu [2], or any other virtualization technique. Device D represents an IoT device that does not itself support WebRTC protocols, but may need to directly communicate with the other devices, e.g., an Arduino-based device.

The services STUN [3], TURN [1], and AAA are cloud-based services that that can be shared by any number of such networks. These are commodity services that can be provided by a third party, e.g., Google. The device registrar is a custom hosted service also running in the cloud. The purpose of the registrar is to keep track of all active device nodes and maintain an active signaling connection with each node. The Proxy Server box enables direct device-to-device communication between devices that support WebRTC peer signaling and devices that do not. The proxy server would typically be deployed close to the devices, e.g., on a gateway node close to device D.

The primary service this network architecture aims to provide is the possibility to setup a direct communication session between any two device nodes, in both directions. The communication session could be either UDP-like unreliable session or a TCP-like reliable session, depending on application requirements. The application may run an arbitrary higher-level protocol over the session, depending on the application and use-case. The session setup process is complicated by the presence of NATs and device nodes that do not provide full networking services to the application (browser JavaScript).

Without the presence of NATs and other middle boxes, WebRTC data channels, represented by thicker black lines in

the diagram, enable direct device-to-device communication. These direct connections are used for all application specific communication needs, including sensor state queries, state change subscription and notifications, and sensor data streaming. In the presence of devices behind NATs, a TURN [1] media relay server is used to relay some of the traffic.

All devices that participate in direct device-to-device connections maintain a persistent signaling channel to a device registrar server. The registrar server facilitates device discovery and exchange of signaling messages. It maintains a device directory database that maps unique device IDs to WebSocket signaling connections. The signaling channel is only used for device discovery and WebRTC signaling, i.e., exchanges of SDP offer-answer messages and ICE candidates [4]. The signaling connections are not directly used by or exposed to IoT applications, they are used purely for network maintenance.

A. Network Forming

A new device joins the network by establishing a WebSocket [5] connection to Device Registrar. The address of the registrar, along with an authentication token for the connection must be configured out of band. Once the connection is established, the client sends a JSON-formatted `register` message to the registrar. The register message includes a unique ID autogenerated by the device as well as an authentication token. The registrar verifies the token against an external AAA server and enables communication on the connection. The device keeps the connection open persistently, the registrar will forward messages from other nodes over the connection.

Once two nodes A and B have joined the same network, they may attempt to establish a WebRTC peer session. The session needs to be established once before any of the two nodes attempt to create a direct device-to-device connection. To establish a WebRTC peer connection to node B, node A generates an SDP offer and sends the offer over its signaling connection, addressed to the unique ID of node B. Node B generates an SDP answer and sends the answer back to the unique ID of node A. Finally, the two nodes exchange ICE candidate messages over the same signaling connection until the WebRTC peer session is established.

To establish a new direct device-to-device connection, the devices use the negotiated peer session and open a new WebRTC Data Channel connection. Multiple data channel connections may be opened simultaneously between two devices. The devices can run any higher-level protocol over the connection.

The connection setup is slightly more complicated for device D which does not contain the WebRTC protocol stack. Fig. 2 illustrates the process. Device D sends a HTTP CONNECT request [6] to the proxy node and specifies the unique ID of the target device (A) in the Request-URI. The proxy node negotiates and establishes a new data channel to the target node and sends a 200 response back to device D. Any traffic sent by D is then transparently forwarded by the proxy to device A and vice versa, until one of the nodes closes the connection.

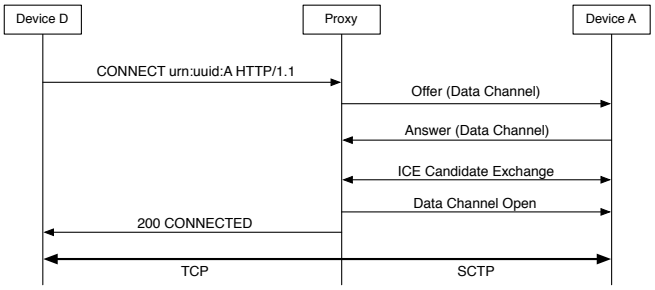

Figure 2. Constrained device D issues HTTP CONNECT request to the proxy node to establish a tunneled connection to device A The proxy negotiates and opens a new WebRTC Data Channel connection to device A. Once connected, the proxy sends a 200 back to D and splices the two connection. Any data sent over the connection by D will be received by A and vice versa.

IV. Web Application

In this section we describe a web application that gets installed on selected device nodes within the network architecture. In particular, referring to fig. 1, the application will be downloaded onto Device A (browser based). The device runtime component of the application can also be deployed on the server based Device C.

The main component of the framework is implemented in form of a web application implemented primarily in JavaScript. To download the application, the user visits a well-known URL of the framework and logs in. Upon logging in, the web application user interface will open in the user's browser.

The architecture of the web application is illustrated in fig. 3. The application consists of two main components: User Interface (UI) and Device Runtime Environment. For performance and security reasons, the two components are designed to run in separate JS contexts. Each JS context is represented by a WebWorker object running within a separate OS-level thread. Additional worker threads may be created at runtime for individual applications, depending on device configuration.

The user interacts with the application via the UI component. The UI provides a means to display a collection of widgets arranged into virtual control panels. An example panel with a larger collection of widgets can be seen in figure 5. Each widget on the panel is internally connected to one of the virtual ports managed by the device runtime environment. The widget either displays the state of the port (for output ports), or provides a means for the user to adjust the port's state (for input ports). The UI also provides an interactive drag-and-drop panel editor (not pictured). The editor can be used to create new panels or modify existing ones. To add a widget onto a panel, the user simply drags the widget onto the panel from a pre-configured widget library. Basic properties of the widget, e.g., colors, names, or value limits can also be configured in the editor.

Apart from UI panel widgets which provide interactivity, virtual ports can also be connected to other sources of data.

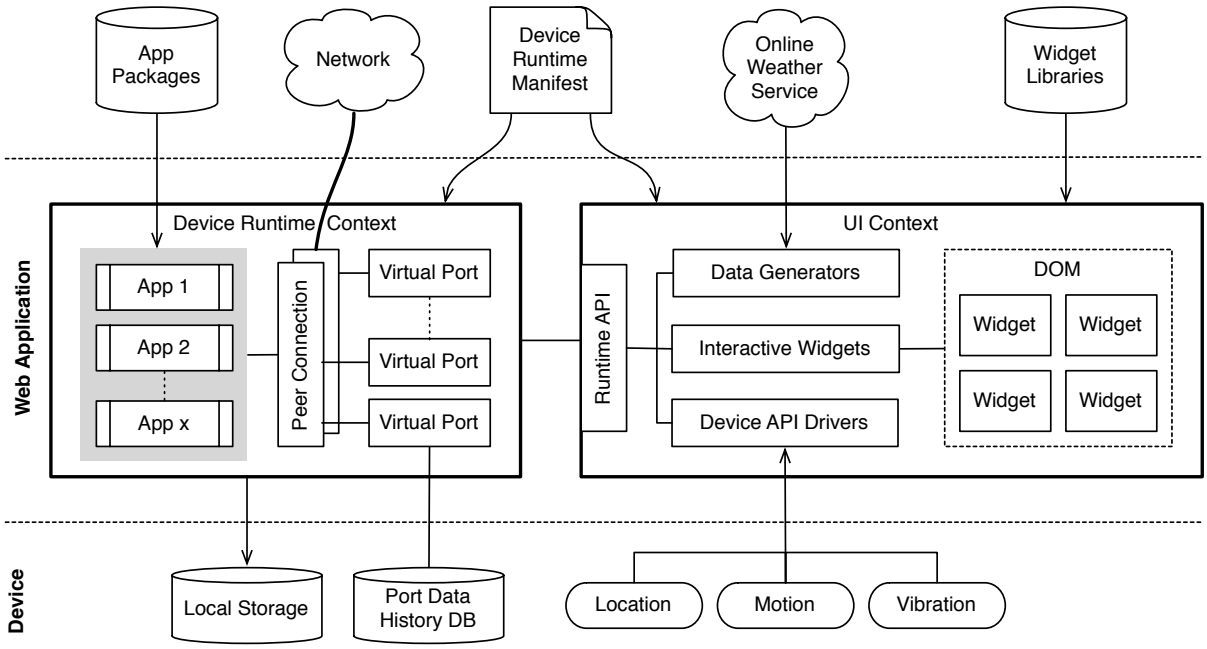

Figure 3. The architecture and main components of the web application. The application is activated when the user visits the URL of the web interface. After initialization the UI component downloads a Device Runtime Manifest and configures the runtime environment according to the manifest. One or more device runtime environments can be created, one for each emulated IoT device.

For example, the user might connect some of the virtual ports to real sensors provided by the JavaScript platform via the Device API [7]. Other virtual ports may receive state updates from online services. Consider a weather service that provides real time temperature updates for a given location. The user could implement a simple function to periodically download the most recent temperature reading and update a selected virtual port with the value.

The device runtime environment, running in a separate JavaScript context (thread), emulates an IoT device. The environment manages a collection of virtual ports which emulate IO ports found on physical hardware devices. Direct device-to-device connections to other devices are managed by the WebRTC protocol stack. The runtime environment may maintain any number of direct connections at a time. By default, the runtime environment provides a protocol with a HTTP-like semantics to query the current state and subscribe to changes of any of the virtual ports. As the state of virtual port changes over time, the runtime environment automatically stores the previous value in a local database. This information is then available to applications in form of a time series database for each virtual port.

Without any local applications installed, the only service that the device runtime environment provides is access to virtual ports. The user or developer can, however, install custom applications into the environment (depicted with gray background in the diagram). If multiple applications are installed at the same time, each application executes within its own Web Worker [8]. Applications are implemented in JavaScript and can access virtual ports and their history data. The primary purpose of local applications is to implement filtering and stateful triggering on the virtual port data. Local applications may also create new virtual ports to enable application chaining. Applications are installed into the runtime environment dynamically at runtime, from packages imported from external repositories such as Github.

A. Physical Device Emulation

The web application described in previous sections provides all the features necessary to model non-trivial IoT devices in the browser. Each such device can provide a set of virtual IO ports connected to interactive web widgets, or to external web services providing environmental data. Local computation over the port data can be implemented with JavaScript processes running in background threads. Access to the device's state and data can be provided via a HTTP-like API running on top of WebRTC data channels.

This model, although powerful, is not without its drawbacks: application developers need to familiarize themselves with custom JavaScript APIs, the application logic must be implemented in JavaScript, and the resulting code is not easily portable to real embedded devices. This is not a problem for applications that have been primarily designed for the browser environment, e.g., control panels with rich web-based UI, like the one shown in fig. 5.

Nevertheless, modern JavaScript engines are performant enough to make emulation of entire embedded IoT devices possible. In other words, it is possible to take an application written for an embedded device in a language other than JavaScript, compile it into JavaScript, and emulate the device

platform entirely in the browser. Device emulation, in addition to pure virtual devices, would let the user rapidly prototype and develop distributed IoT applications for heterogeneous networks of devices, all without the need to have access to real hardware. We discuss the emulation approach in the rest of this section and demonstrate the approach for the popular Arduino platform.

The Arduino board provides a simple 8-bit Atmel CPU with a small amount of memory [9]. Digital (GPIO), analog, and a USB port are provided for connecting peripherals (sensors and actuators). The board can be connected to the Internet using expansion boards for Wi-FI and Ethernet. Programs for the Arduino are written in a C++-like language which is compiled to C and assembler for the target CPU. Only single threaded applications are supported due to the limitations of the hardware and platform.

The simplicity of the Arduino platform makes it a suitable candidate for emulation in the browser. Consider a simple Arduino program written in C++. The program interacts with attached sensors and actuators, performs some minimal amount of processing on the data, and provides access to the state of device via an attached Ethernet adapter or USB [1]. Our JavaScript framework will make it possible to emulate such an application in the browser, requiring no or trivial changes to the source code. To run the Arduino application in the browser, we need to translate the Arduino C++ code into JavaScript. Figure 4 illustrates the compilation process.

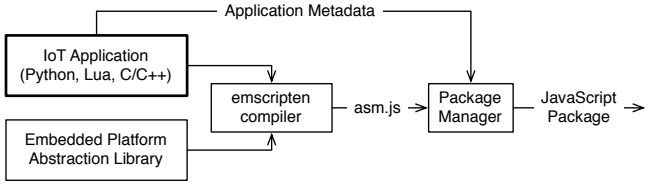

Figure 4. An embedded application written in C/C++ is combined with a platform library, e.g., for the Arduino. The code is compiled with the Emscripten compiler into a subset of JavaScript called asm.js. The result then packaged and uploaded to a server from which it can be imported into the browser JavaScript context.

To make the compilation possible, we first need to provide an Embedded Platform Abstraction Library. This is a shim library that translates access to the hardware resources of the Arduino platform to their emulated counterparts in JavaScript. Most importantly, the library transparently translates access to IO ports from Arduino hardware ports to virtual ports in JavaScript. Our framework will provide platform abstraction libraries for selected simple embedded platforms.

We first compile the Arduino application with the compiler provided by the Arduino IDE to generate standard C code. The generated C code, together with the platform abstraction library, is then compiled by Emscripten [10]. Emscripten uses LLVM to compile C/C++ into bytecode and then compiles the bytecode into a subset of JavaScript referred to as asm.js [11]. Asm.js is a low-level, performance optimized subset of JavaScript that can be directly interpreted by the browser. In the final step, the framework creates a self-contained package out of the generated code and metadata provided by the application and the platform library. The metadata includes platform configuration information, e.g., the type and number of virtual ports to create, and other information. The resulting package is compatible with the `npm` and `jspm` package managers and can be used in the browser or on a Node.JS server [12].

The generated package can be directly imported into the device runtime environment like any other application. That way, developers can use native JavaScript applications to implement local computation for devices that will always be running in the browser, e.g., tabled based control panels. Applications that will eventually be installed on real hardware can be prototyped in the browser and, eventually, transferred to real hardware. Browser based prototyping rapidly speeds up and simplifies the development process of IoT applications running on IoT devices themselves, as opposed in the cloud.

The Emscripten compiler comes with a large collection of libraries that re-implement the most common APIs in the browser. Included are (among others): filesystem emulation, graphics (SDL, OpenGL), and audio APIs. This opens a possibility of emulating larger, more sophisticated IoT devices that provide, e.g., a user interface or multi-media capability. We only mention this for completeness and to show that IoT device emulation in the browser might be a promising research direction. The ability to emulate heterogeneous networks of devices would open up the possibility of rapid IoT application development.

V. Prototype Implementation

In this section we briefly describe an initial prototype implementation of the web application framework.

The bulk of the framework is implemented in form of a single-page JavaScript application. The web application is implemented in the upcoming ES6 (Ecmascript 2015) dialect of JavaScript, compiled by Babel to a dialect that modern browsers and Node.js understand. The application relies heavily on the ES6 module API. We use the module API to provide the possibility to import external widget libraries, provided by third-parties, into the framework. The same module API is also used to download and install IoT application code into the device runtime environment. Our prototype uses the `jspm` package manager to locate and download IoT applications and widget libraries from public repositories such as Github.

The UI component of the web application requires a modern browser and currently works in recent versions of Google Chrome only. This component requires some of the more recent HTML5 APIs that, as of writing this document, do not seem to be provided in other browsers. The UI itself, including virtual port widgets, is implemented in form of reusable ReactJS components. We choose ReactJS because the framework makes it easy to design complex, highly interactive

[1]This kind of application is considered a "HelloWorld" application equivalent in many IoT projects

UIs without the need to deal with the complexities of DOM updates.

As of the time of writing this document, the framework does not include support for native applications described in section IV-A. Nevertheless, we have experimentally verified that it is indeed possible to compile a simple Arduino application into JavaScript and run it emulated in the browser. Support for compiled native applications together with an Arduino platform library will be included in a future version.

The Websocket Server and Device Registrar services have been implemented on top of Node.js, using the the `ws npm` package. Currently, the only purpose of the WebSocket signaling server is to facilitate an exchange of WebRTC offer-answer messages and to enable device discovery. The server authenticates incoming WebSocket connections from clients against an external authentication service. The authentication is based on HTTP cookies and OAuth2 bearer tokens submitted by the client in the initial HTTP upgrade request. The WebSocket server relays messages among the connected clients based on their registered addresses (randomly generated UUIDs). The registrar is a custom JavaScript application which keeps track of registered device addresses and translates those addresses to WebSocket connection identifiers. The information about device addresses and connection identifiers is stored in a REDIS memory database.

The HTTP Proxy Server is also implemented in JavaScript on top of Node.JS, but runs in a separate server instance. The proxy server includes a server-based WebRTC peer implementation based on the native openwebrtc library. The WebRTC peer is used to enable direct data channels connections with other WebRTC enabled devices. The HTTP proxy is implemented on top of the Express framework.

The HTTP-only Device D is a BeagleBone Black device running Linux with a custom application implemented in Python. This device does not support WebRTC signaling and is used to demonstrate the functionality of the WebRTC-HTTP proxy. The device keeps a subscription to some of the virtual ports in one of the browser devices.

In addition to the components described above, the system relies on a few external services provided by Google. These services include STUN [3] and TURN [1] servers, as well as the HTTP DNS Resolver API (for resolving domain names in JavaScript).

VI. Background And Related Work

The idea to use the WebRTC protocol stack [13] for communication in IoT systems is not new and it is not the only approach. Nevertheless, it is the only approach that is readily available in modern browsers which are a significant component of our architecture. An implementation for embedded devices is available in form of a native WebRTC library.

The web application uses the JavaScript Device API to gain access to real physical sensors available on the platform running the browser. The set of sensors provided via the API is limited. The authors of Maverick discuss a better approach in their paper [14].

Figure 5. An example of a control panel showing an assorted collection of port view widgets. Each widget is connected to a virtual port and either controls the port or displays its value. The application provides a drag and drop UI to create such panels. The panel is running in the Chrome browser on a mobile device (tablet).

The topic of device emulation and simulation has also received considerable attention from the research community. A variety of Arduino emulators exist on the market [15], [16]. The qemu emulator [2] is a general purpose platform emulator that can emulate a large collection of platforms and devices. Most of the existing emulators require the installation of custom software on user's computer, provide no support for networks of devices, and cannot be used with different devices at the same time. The framework that we envision in this paper addresses some of these shortcomings. Viptos is a graphical simulation environment for Cyber-Physical Systems [17].

Instead of doing emulation in JavaScript, an alternative approach might be to run an emulation environment on the server and access it remotely via an API. This is an approach taken by Mininet [18] in the context of network emulation.

Some ideas behind the web-based control panel and widgets have been inspired by the programming language Scratch [19].

The development and deployment of distributed applications has received some attention in the context of Wireless Sensor Networks (WSN) [20]. Nevertheless, the primary focus in WSN is on communication, rather than computation. We are not aware of any attempts to reuse these ideas in the IoT context.

A. WebRTC Protocol Stack

Our network architecture and the prototype web application relies heavily on the WebRTC protocol stack. The protocol stack has been primarily developed for direct browser-to-browser media connections, e.g., video calls from the browser. Our framework currently uses a subset of the stack only, namely, we use the Data Channel API to implement direct device-to-device communication.

WebRTC implements the data channel using the Stream Control Transmission Protocol (SCTP) [21]. We use the SCTP protocol in ordered (reliable) mode to emulate TCP connections. In WebRTC, the SCTP protocol communication is encrypted using DTLS [21] and encapsulated in UDP.

In order to setup the SCTP connection, the two browsers need to learn the IP address of the remote peer and negotiate session parameters. WebRTC uses the SDP offer-answer model [22] to negotiate parameters and ICE [4] and STUN [3] protocols to learn IP addresses. All communication during the negotiation phase between the two browsers takes place over a WebSocket [5] signaling connection.

VII. Conclusions and Future Work

In this paper, we have argued that the initial development and prototyping of distributed IoT applications is a complex task. This is due to missing tools and abstractions which forces developers to perform this initial prototyping on real hardware.

We have presented an architectural design and early prototype implementation of a web-based application framework that attempts to address some of these obstacles. The framework provides a web-based user interface where the user or developer can quickly create interactive widgets. The application also provides a runtime environment for device emulation. Finally, the proposed network architecture makes it possible to build networks of such nodes and integrate devices emulated in the browser with real physical devices.

We have also described an approach towards emulating other device platforms that are not based on JavaScript, namely the Arduino. We have experimentally verified on simple applications that the approach is indeed feasible. The current version of our application does not include this functionality. We intend to add support for this feature in a future version of the platform.

A. Acknowledgements

The authors would like to acknowledge Martin Wandtke and Claudia Armbruster, visiting students from UniBW Munich, for their contribution to the implementation of the initial framework prototype. We would also like to thank Tim Eschert, visiting student from RWTH Aachen for help with development and experimentation.

References

[1] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5766.txt

[2] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[3] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008, updated by RFC 7350. [Online]. Available: http://www.ietf.org/rfc/rfc5389.txt

[4] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010, updated by RFC 6336. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[5] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[6] R. Khare and S. Lawrence, "Upgrading to TLS Within HTTP/1.1," RFC 2817 (Proposed Standard), Internet Engineering Task Force, May 2000, updated by RFCs 7230, 7231. [Online]. Available: http://www.ietf.org/rfc/rfc2817.txt

[7] "Device apis working group," http://www.w3.org/2009/dap, Accessed: Apr. 2014.

[8] "The WebWorker API," https://html.spec.whatwg.org/multipage/workers.html, Accessed: Jul. 2016.

[9] "The Arduino," http://www.arduino.cc/, Accessed: Jan. 2013.

[10] "The Emscripten Project," http://emscripten.org/, Accessed: Jul. 2016.

[11] "The Asm.js Project," http://asmjs.org, Accessed: Jul. 2016.

[12] "Node JS," http://nodejs.org, Accessed: Apr. 2014.

[13] "WebRTC 1.0: Real-time communication between browsers," https://www.w3.org/TR/webrtc/, Accessed: Jul. 2016.

[14] D. W. Richardson and S. D. Gribble, "Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices," in *Proceedings of the 2011 USENIX Conference on Web Application Development (June 2011)*, vol. 11. USENIX Association, 2011.

[15] "Code:blocks," http://arduinodev.com/codeblocks/, Accessed: Jul. 2016.

[16] "Circuits.io," http://circuits.io, Accessed: Jul. 2016.

[17] E. Cheong and Y. Zhao, "Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks," in *SenSys*. New York, New York, USA: ACM Press, 2005, p. 302.

[18] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 253–264.

[19] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 1–15, 2010.

[20] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: A methodology for architecture-independent programming of networked sensor systems," in *Proceedings of the 2005 Workshop on End-to-end, Sense-and-respond Systems, Applications and Services*, ser. EESR '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 19–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1072530.1072535

[21] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6096, 6335, 7053. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[22] J. Rosenberg and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)," RFC 3264 (Proposed Standard), Internet Engineering Task Force, Jun. 2002, updated by RFC 6157. [Online]. Available: http://www.ietf.org/rfc/rfc3264.txt