The Palcom Device Web Bridge

Sandholm, Thomas; Magnusson, Boris; Johnsson, Björn A

2012

The Palcom Device Web Bridge

Thomas Sandholm
Boris Magnusson
Björn A Johnsson

Technical report, LU-CS-TR:2012-251
ISSN 1404-1200, Report 100, 2012

Lund University

LU-CS-TR:2012-251
ISSN 1404-1200
Report 100, December 2012

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

The Palcom Device Web Bridge

Thomas Sandholm, Boris Magnusson and Björn A Johnsson

Department of Computer Science, Lund University, Sweden
{thomass,boris.magnusson,bjornaj}@cs.lth.se

ABSTRACT

In this report we present the design of an application development toolkit for constructing web user interfaces to arbitrary device services. These device services may produce real-time flows of data that need to be analyzed and monitored. Additionally, they may allow control of physical equipment. For example, a medical pump may be controlled remotely to inject a precise dose of some pain-relief pharmaceutical in a cancer patient, based on monitored indications of pain. We focus our discussion around three main components of the toolkit, first, an event bus for efficiently communicating device-generated notifications and data; second, a web widget platform, and third a firewall-traversal solution for real-time, peer-to-peer communication in constrained organizational networks, such as a hospital site.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Web-based interaction*

1. INTRODUCTION

With the recent proliferation of Internet-connected personal as well as organizational devices, many integration opportunities as well as challenges surface. The first and most obvious challenge is how to make all devices use a common protocol to simplify orchestration and workflow configuration. Our previous work on the Palcom infrastructure [7] to a large extent tackle these issues by providing a common service platform on top of a uniform UDP protocol for discovering and communicating with device services. Palcom also provides an assembly tool and a programming language that allows developers to compose new services from pre-existing ones. Furthermore, the platform has the ability to tunnel communication between device services across networks in a secure and efficient way. The main difference from traditional software development in the area of device control is the bottom-up development paradigm where the devices, their services and, in many cases also a full workflow, exist and the main issue remaining is how to present these capabilities to end users.

The work described here focuses on providing building blocks for developers to easily compose end-user interface that interact with Palcom device services. Since both devices and people who need access to these devices are distributed in nature, accessibility and availability are key requirements. The ubiquity of web browsers on both old and new consumer devices as well as the rapid evolution of advanced new standards, such as HTML5, lead us to a design around web protocols and web-based user interfaces.

The report is organized as follows. First (Section 2), we discuss a web event bus for communicating asynchronous device events to web clients. Second (Section 3), we present a web widget platform designed to communicate with device services. Third (Section 4), we give an overview a novel solution implemented to tunnel real-time web data, such as audio, video across firewalls. Finally, we conclude the report with lessons learned and how our solution relates to other known efforts.

This work was done as part of a project to provide IT-based support for home-care treatment of cancer patients [1].

[1]http://itacih.cs.lth.se/itACiH/itACiH.html

2. WEB EVENT BUS

This section presents the design of a bridge from the Palcom service platform to browser clients, capable of both communicating asynchronous events generated by device services and controlling potential actuators on devices. Fundamentally, this meant building a Web server communicating with standard web protocols in one end (HTTP/HTTPS) and the Palcom UDP protocol in the other end. The bridge serves as the foundation for our integration work and higher-level features such as the web widget framework and the firewall traversal service described in Section 3 and Section 4 respectively.

The first challenge was to overcome the inherent poll nature of HTTP/AJAX based user interfaces prominent on the WWW today. Instead of polling a server for events we want the server to push events directly to all the browser clients interested in some event channel. Using notification technologies such as RSS/Atom is not appropriate due to the real-time nature of the events. Imagine delivering a patient heart-rate via RSS. Simply using polling AJAX calls could work but would not be very efficient and the time to receive an update would be tied to whatever poll frequency was configured. Configuring an appropriate interval not only relies on the application semantics but also the network and the client and server load, and thus is non-trivial.

HTML5 [4] comes to the rescue with the introduction of standards such as Server-Sent Events (SSE) and WebSockets. The main issue with these standards is that they are too immature to provide any real cross-platform portability which is the reason we turned to the web in the first place. Legacy browsers as well as legacy devices are important for us to support in our work. A collection of HTML4 hacks known under the umbrella of *COMET* [3], has been used to overcome this problem in legacy applications. We build our solution around the technique of long polling, as it is the most widely supported technique across all browsers. It only relies on the standard XMLHttpRequest (XHR) API known as AJAX [1].

The basic idea with long polling is to keep connections to the Web server alive until an event comes in from the server. Then the server writes the response and closes the connection and the client immediately turns around and issues a new request. If there has not been any event for a specific period of time the client would time out and create a new connection to the server as before. The main issue to be solved is to avoid too many reconnections while making sure that inactive clients don't consume server resources. In applications that don't accept any message loss it is also important to maintain a buffer on the server to capture events that arrive during the time when the client is reconnecting. The main attraction of this approach is that it is simple, works trivially even in legacy browsers with minimal code, and it is easy to configure to work efficiently in many network settings. A number of open-source long polling frameworks exist, such as Sock.JS [2] and Socket.IO [3], Tunguska [4]. They rely on immature backends and complex protocol negotiations between older and newer standards as well as COMET hacks and fallbacks such as flash. Our requirement was to build an easily embeddable solution on the server based on Java, as the Palcom protocol has its most mature implementation in Java for portability reasons. We hence decided to build our own long-polling solution on top of the Java Netty toolkit. Apart from the simple yet powerful architecture there are also a number of novel protocol features in our solution, described below.

These added features were designed to make the client simple, yet cross-platform compatible, and include event ordering, event batching, client controlled history retrieval, payload based server side filtering and client subscription reuse. Furthermore we define a payload format in JSON to make it easy to parse self-contained event data and meta data and a simple header to accept or reject messages before parsing the payload. This more structured payload also allows us to do more structured event filtering.

2.1 Basic protocol

The basic protocol is based on three different timestamps that are sent in a header and a payload that may be parsed separately. Many different payload formats may be plugged into the basic protocol but to get all the benefits of batching, ordering and filtering described below we also use a well-defined structured payload, described in the next section. There are three HTTP GET requests defined in the basic protocol, to join groups (aka channels), to poll client queues, and to leave groups. To be able to receive any events when polling a client queue the client id in the request needs to have been subscribed to groups that produce events. To subscribe to one or many new groups the folowing HTTP GET requests should be issued

```
HTTP GET join?id=CLIENT_ID&groups=GROUP_IDS
```

where CLIENT_ID is the id of a client browser that may be a generated guid that is stored in a cookie in the browser, and GROUP_IDS is a comma separated list of group ids. Similarly to unsubscribe to a channel the following HTTP GET request is issued.

```
HTTP GET leave?id=CLIENT_ID&groups=GROUP_IDS
```

To poll for events the following request is issued:

```
HTTP GET poll?id=CLIENT_ID&timestamp=TIMESTAMP&sequence=SEQUENCE
```

where CLIENT_ID is the id used in the join and leave calls, TIMESTAMP is the time in milliseconds of the last event seen (that should not be replayed) and SEQEUENCE is a sequence number used to order events occurring at the same millisecond. All timestamps are generated on the server before events are queued. The response of the poll call looks as follows:

[2]https://github.com/sockjs/sockjs-client
[3]http://socket.io
[4]http://www.sitepen.com/blog/2010/07/19/real-time-comet-applications-on-node-with-tunguska/

```
PROTOCOL_MARKER LAST_EVENT_TIEMSTAMP LAST_EVENT_SEQUENCE SERVER_TIMESTAMP
PAYLOAD
```

The first line of the HTTP Response payload is a header that contains a PROTOCOL_MARKER used to identify the header information and potentially version it, a LAST_EVENT_TIMESTAMP and LAST_EVENT_SEQUENCE that indicate the timestamp and sequence number of the last event returned in the payload respectively. The LAST_EVENT_TIMESTAMP is 0 in case no new events were found, i.e. the poll timed out. The SERVER_TIMESTAMP may be used by the client to reliably determine if the events returned are too old to be of interest by simply doing a diff between the LAST_EVENT_TIMESTAMP and SERVER_TIMESTAMP which would indicate how many milliseconds ago the most recent event happened. All this information is availabel so the client could reason about whether it makes sense to parse the payload. The last event timestamp and sequence values should also be used when sending the next poll message to ensure that no messages are missed without affecting any other polling clients. The server_timestamp may also be used to determine how old individual events are in the potential batch of events returned in the payload.

2.2 Payload format

The payload is a JSON object with the following format:

```
{"events":[{"timestamp":EVENT_TIMESTAMP,"message":MESSAGE}],"timestamp":LAST_TIMESTAMP}
```

The events array is an ordered list of events from oldest to newest adherring to the timestamp specification in the request. If the tiemstamp in the header is not 0 the events array contains at least one element. The outer timestamp is the most recent timestamp of all the event tiemstamps in the array. The message is a string that may in turn be a json formatted message. We also define a message format with some meta data as follows:

```
{"deviceID":DID, "serviceId":SID,"instanceId":IID,
 "command":COMMAND,"elements":[{"type":TYPE,"name":NAME,"data":DATA}]}
```

where DID is the device id of the device generating the event, SID is the service id on the device that generated the event and IID is the instance id of that service. COMMAND is the command or event name, elements is a list of parameters of the command. Each parameter has a TYPE which is typically a mime type and a NAME used for payload filtering as well as the raw DATA. This data may in turn be JSON but it is then application specific.

2.3 Event ordering

As we have seen from above the protocol allows for simple client-side reasoning about which events to process based on timestamps and the events are strictly ordered to make replays safe. To achieve this the server serializes the addition of all the events and timestamps them with a millisecond timestamp. If the timestamp of an incoming event is the same as an already timetamped event in the same client queue it will bump up the sequence number but use the same timestamp. The first event with a given timestamp always has sequence number 0.

2.4 Event batching

As could be inferred from the payload format a single poll request may return more than one event. This is crucial for scalability reasons and it significantly speeds up bootstrapping where a client may have missed many events when starting up. Also during very high load the server buffer may fill up fast when the client reconnects, and since the client needs to reconnect for each poll it could lead to thrashing when the client is just reconnecting and getting further and further behind in consuming the latest events. The server ensures not only that the returned events adhere to the timstamps in the poll request but that the events in the batch returned are ordered in the time they occurred which simplifies replay.

2.5 Client-controlled history

The timestamp parameter of the poll request described above may be 0 in which case all events not previously returned to that CLIENT_ID as known by the server will be retrieved. The timestamp parameter may also be a negative number in which case the absolute value of the timestamp is taken to mean the maximum number of events to be returned from the top (the most recent) of the queue. 0 and negative values of the timestamp parameter in the poll call may be used to bootstrap clients, e.g. after a browser page refresh. Using 0 continuously is a bit dangerous since it would for instance mean that multiple tabs in the browser may not all get the events if they share the same CLIENT_ID. So after the initial bootstrap it is recommended that the latest timestamp of the events returned is passed into the next poll call. A negative number may be used to replay a few messages in a stream to avoid caching them on the client but still displaying them in something like a positioning trace or historical graph to the end user. If a client has been off-line for some time it may also be useful to restrict the maximum number of events returned which otherwise could overload the client both when timestamp 0 or a last seen timestamp are used.

2.6 Payload-based server-side filtering

As alluded to above in the payload section the well defined message structure allows us to reason about which payloads we are interested in on the client side. We make use of this in a filter that may be attached to join request. By default when you join a group you receive all events on that group (assuming they are not only sent to a single client in the backend device which is also possible). To restrict the events further on the server to save on bandwidth a filet may ba appended to the group name in the join call. The format of the filter is:

```
|PARAM_1|FILTER_1||PARAM_2|FILTER_2....||PARAM_n|FILTER_n
```

The first "|" separates the filter specification from the group id. Then each filter is separated by a "||" and each filter comprises a parameter name and a filter value separated by a "|". All filters must match for the event to be sent to the client. So the "||" may be interpreted as a logical AND. If you want OR behavior for filters then the same group id with a different filter must be specified. The PARAM_n names must match the NAME of the message element NAME in the message payload described above in order to be evaluated. The evaluation is currently a direct string comparison (but could just as easily be a regexp) on one of the parameters, e.g. a record id.

2.7 Client channel subscription reuse

A browser client may create many subscriptions on a single id, create many subscriptions on a set of different ids, or share subscriptions by sharing ids with other browser clients. The typical case is that a unique id is generated for each browser instance, so that different tabs or page refreshes may read events from the same channel and share subscriptions. The pages and tabs may still read from different positions in the channel queues by making use of the timestamp parameter as long as it is not fully controlled by the server (e.g. timestamp is always 0).

3. WEB WIDGET PLATFORM

Building on the long polling model that we just described, we now present the components we designed to make it easy to build complete web application user interfaces. There are three parts to our design. First we build some core APIs to connect backend device service notifications to DOM UI elements, then we provide an application Grid container to layout content in pre-defined panes and to provide generic panes, last we provide a widget plugin model and provide a gallery of high-level generic widgets. These three parts are also supported by a number of generic backend services to allow for richer and more customizable interactions. We describe some of them below and we dedicated a separate section to the service that powers the web conferencing widget as it is more involved due to the firewall traversal capabilities.

3.1 Connecting device services to DOM elements

A device contains services. Both the device and its services are addressable using a UDP based messaging middleware called Palcom. A message in Palcom is called a command and each command has a list of parameters, which in turn are all typed and named. The command structure looks the same both for messages sent to the device services and for events produced by the device sensors. Sending a command through the web application and the Web server to a device is straightforward. We simply need to specify the device, and service ids and then send over the parameters via standard HTTP. The message is then translated by the WebServer into a Palcom UDP message and sent to the correct device and service. As a simplification we let each Web server have a default device to allow easy discovery of available services and to be able to connect to it without specifying a device id. Fur outgoing commands produced by the device services we leverage our long polling framework. Browser clients can subscribe to events from a service by specifying the group name of the long polling channel as a concatenation of service id and event. In our design this is achieved by specifying the id of a dom element and the group name together with a callback function, and some options to pass context from subscriber to event receiver. A number of standard callback functions are provided to populate e.g. the value attribute of textarea, input text, button etc automatically if an incoming event is received. Checkboxes, Radiobuttons, and Image tags may also be set automatically with standard callbacks. Below are some examples of callback functions for standard DOM elements.

```
Palcom.ValueSetter = function(ui,data,options) {
ui.val(decodeURIComponent(data[0].data));
}
Palcom.ValueChecker = function(ui,data,options) {
ui.attr('checked',(options.equals === data[0].data));
}
Palcom.ValueSelector = function(ui,data,options) {
ui.attr('selected',true);
}
Palcom.ImageSetter = function(ui,data,options) {
   ui.attr('src',data[0].data);
}
```

Here are some examples of how to use these callbacks

```
Palcom.connect(BUTTON_ID,SERVICE,ID,COMMAND,Palcom.ValueSetter);
Palcom.connect(TEXT_ID,SERVICE,ID,COMMAND,Palcom.ValueSetter);
Palcom.connect(RADIO_ID,SERVICE,ID,COMMAND,Palcom.ValueChecker,{equals:'true'});
Palcom.connect(CHECKBOX_ID,SERVICE,ID,COMMAND,Palcom.ValueChecker,{equals:'true'});
Palcom.connect(SELECT_OPTION,SERVICE,ID,COMMAND,Palcom.ValueSelector);
Palcom.connect(IMAGE_ID,SERVICE,ID,COMMAND,Palcom.ImageSetter);
```

The first ID in the call is the html tag element id to connect, and the second id is part of the service identifier. to send a command to a service on a device you simply issue the following call:

```
Palcom.sendCommand(SERVICE,ID,COMMAND,PARAMS,POST_DATA,CALLBACK,OPTIONS);
```

SERVICE, ID and COMMAND are specified as above. PARAMS is an array of parameters, and POST_DATA allows you to optionally send a parameter as a post message if it is too long to encode in a url. CALLBACK is a callback function that will be called when the command has been sent. OPTIONS is a dictionary of optional parameters, such as an external device to send the command too.

3.2 Application Grid

Now with the basic connection established between UI components and device services we wanted to provide some tools that allows complete applications to be designed more rapidly. The most foundational support to that effect is the application grid. The application grid allows the application developer to position elements in predefined sections of a Web page. Some sections are prepopulated with standard content and sections have predefined dependencies and interaction patterns. Apart from styling sections such as logo and application title and a toolbar to toggle to and from fullscreen there are four important sections of all applications written with out toolkit.

- **Navbar.** This in an area that displays record elements, such as patients or products, i.e. items you select for further exploration and interaction. It is implemented as an accordion widget.
- **Menu.** This is a functional menu that is application specific. The idea is that each function operates on the currently selected Navbar element. Each of these functions get a separate tab in a tabbed menu to display content. Each tab is also associated with a device service. When a new navbar item is selected the tab pane updates. When a new tab is selected the same navbar item is used for the new service associate with the selected tab.
- **Alert Bar.** This is an application wide notification area. All the alerts sent out will be broadcast to everyone currently displaying content from the application. There are three section, Alarm, Waring and Info. Each section has a queue of events and timestamps are displayed in a real-time updating human friendly format, e.g. two days ago, or two weeks ago. Anyone using the app can submit messages of any of the three alert types directly in the alert bar. A backend service persists the messages and potentially does some filtering to make sure everyone see the same lists regardless of browser restarts.

These areas can be put in a west,north,center,east,south layout where the center pane (the tab content areas) size adapts to the browser window size. Other application wide features include selecting a UI theme and a Web Font type font family for the application. This configuration as well as the Navbar and Menu content are provided by a UI service that may be extended to provide more dynamic content then the default service that simply reads the content from configuration files in a JSON format. An example of the layout for a typical application can be seen in Figure 1. This feature was implemented using the JQuery Layout plugin [5].

3.3 Widgets

To complete the picture and render a fully functioning web interface we also need to provide support for rendering the services mapped to the various menu panes. Developers may write their own HTML/JS/CSS code together with our core API to interact with services but we wanted to provide some higher level tools both for debugging and for styling and composition of standard widgets. Widgets may be configured with the Menu configuration. E.g. a standard widget is mapped to a menuitem which in turn is mapped to a service. The service will then need to support the protocol of the widget in order to function properly. We thus define some standard protocols, with payloads in JSON, where services are responsible for implementing some commands and then return the data in a standard format. Some of these services may be general enough to just be plugged in whereas others require you to plug in your own data stream to become interesting. Nevertheless, these standard widget protocols allow us to develop more complete, higher-level, and more reusable widgets than the typical widgets seen in JavaScript libraries such as JQuery UI [6] and YUI [7]. All the widgets are data driven, meaning that they can be reused without changing any of the UI components for a large variety of data sets. For instance, in the medical domain there are many datasets that are historical streams of patient device measuring data. All of these data sets could be rendered through the same widgets that allows for easy discovery, navigation and graphing of time series. Another widget, the table widget allows for display, search, navigation, selection, and styling of any tabular data including html, text, numbers, images, wiki content. There is also a form widget, making it easy to hook up input elements to a series of sendCommand calls as described above when a user clicks a submit button. Finally a debug widget gives allows the entire service widget to be rendered automatically including the ability to send events to available device services and receive events from them. A number of media related widgets are also provided but they are described separately as part of the firewall services in Section 4.

A gallery of widgets suported is shown in Figure 2. The Table and History widgets are used to present and navigate tabular data and data time series respectively. The Debug widget can be automatically generated from any service available in the Web server without configuration to debug events and controls. As a meta widget we provide the ability to configure nested menu bars in the Tabs widget. All the widgets may be arbitrarily nested inside menu panes.

Application developers may also write and plug in reusable widgets by implementing a couple of callbacks and registering themselves with the core library. As follows:

[5] http://layout.jquery-dev.net/
[6] http://jqueryui.com
[7] http://yuilibrary.com

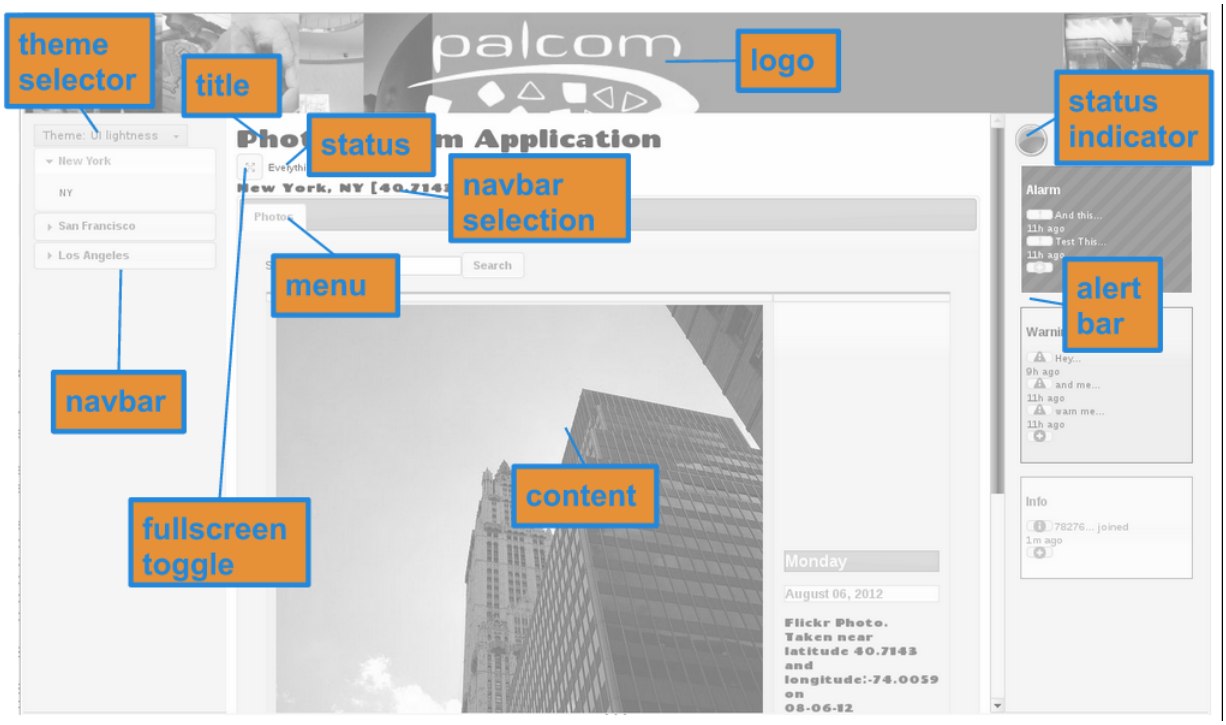

Figure 1: Web application layout.

```
var ExampleWidget;
if (!ExampleWidget) {
    ExampleWidget = {};
}
(function () {
ExampleWidget.setupWidget = function(service,instance,ui,value,options) {
    // display widget in ui here (DOM element ID where ui should be rendered)
    // and call service with core API (Palcom.connect,Palcom.sendCommand)
};
ExampleWidget.destroyWidget = function() {
    // clean up widget state here
};
}());
(function () {
   Palcom.registerWidget(EXAMPLE,ExampleWidget);
}());
```

EXAMPLE is the name of the widget as used in the menubar configuration to specify that a service should be rendered using this widget.

3.4 Widget Data Payload

Both the general purpose history and table widgets use the same standard JSON payloads to communicate graph point and table cell data respectively. This payload has the following format:

```
{
 "headers":[{"name":ROW_HEADER_TEXT","id":"header"},
            { "name":COL_1_HEADER_TEXT,"id":COL_1_HEADER_ID},
            { "name":COL_2_HEADER_TEXT,"id":COL_2_HEADER_ID},
            ...
            { "name":COL_c_HEADER_TEXT,"id":COL_c_HEADER_ID}],
 "rows":  [[ROW_1_HEADER_TEXT,VAL_1_1,VAL_1_2,...,VAL_1_c],
           [ROW_2_HEADER_TEXT,VAL_2_1,VAL_2_2,...,VAL_2_c],
           ...
           [ROW_r_HEADER_TEXT,VAL_r_1,VAL_r_2,...,VAL_r_c]],
 "rowids": [ROW_ID_1,ROW_ID_2,...,ROW_ID_r],
```

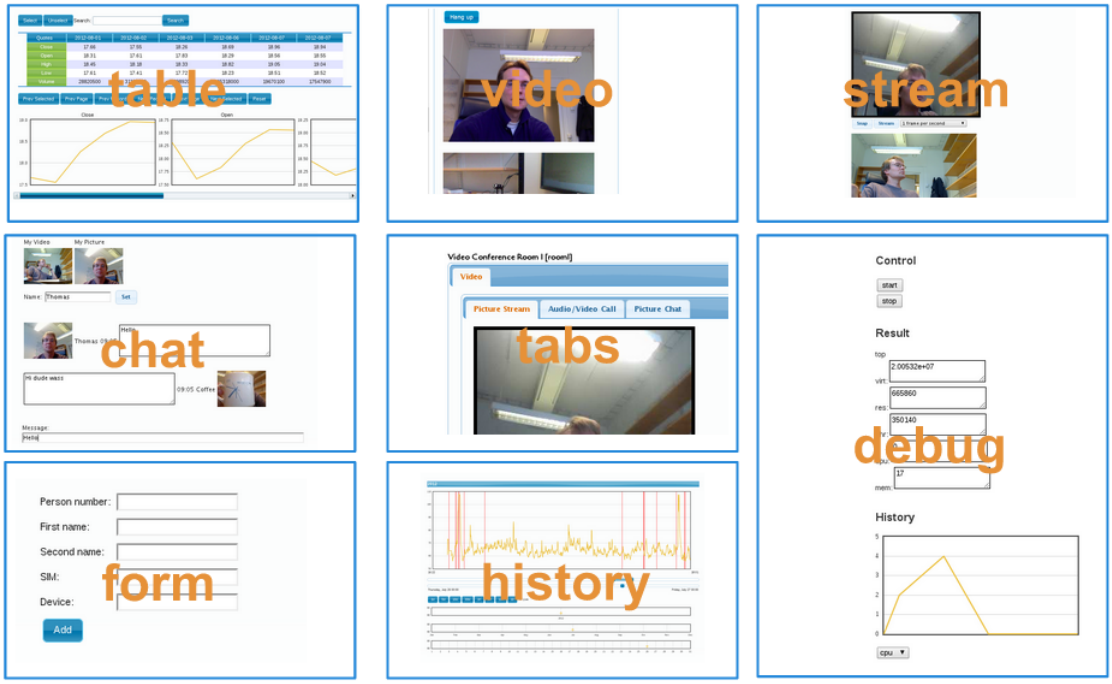

Figure 2: Gallery of supported widgets.

```
 "selected": [[SELECTED_ROW_ID_1,SELECTED_COL_ID_1],[SELECTED_ROW_ID_2,...,SELECTED_ROW_ID_s]]
}
```

This is an example with r rows, c columns and s selected cells. The SELECTED_ROW_ID must match an id in rowids and SELECTED_COL_ID must match an element in the headers list. The number of elements in each row array must be the same as the number of elements in the headers array. The first item in the headers array and the first element in the row array are by default used to denote table header labels. The header ids are used for pagination and search, e.g. give me 5 items after HEADER_ID x is a valid protocol request. Give me all items between HEADER_ID X and HEADER_ID Y is another example. All the requests are also associated with a record id, typically selected from the navbar items.

3.5 Widget Data API

Depending on which APIs a service exposes, different UI features will be available. A widget typically requires a minimal set of APIs to be implemented in order to at least initialize correctly.

```
// Gets an index of years, months, and days with data including
// average values for record id record
GetIndex(record) -> Index

// Gets all data records between epoch dates from and to
// Data follows the format described in the previous section
GetHistory(record,from,to) -> Data

// Gets period number of data records starting at position id
// period may be negative in which case records before position
// id will be retrieved. If period starts with "+" it is an
// inclusive search otherwise the record at position id
// will not be included in the Data returned.
GetData(record,id,period) -> Data

// Searches through the data like the GetData function
// but only periods with items marked as selected will
// be returned
GetSelectedData(record,id,period) -> Data

// Select a cell in a record table at a column id and
```

```
// row id
Select(record,colid,rowid)


// Unselects a cell in a record at a column id and
// row id
UnSelect(record,colid,rowid)
```

3.6 Generic Application Services

To power the UI interactions in a general way without having the application developer worry about features common across many applications we developed a number if general purpose services.

- **UIService.** Loads config of menu and navbar for applications, and sends status and alarms.
- **PingService.** Allows the generic application code to detect what level of connectivity there is to the backend services.
- **AlarmService.** This service as already mentioned coordinates alert broadcasting of alarms, warnings, and info across clients using the same application in real time. Messages are persisted and queued/dequeue/dequeued in a list of constant size.

All of these services may be customized by the application developer as long as the protocol is maintained.

4. WEBRTC FIREWALL TRAVERSAL

WebRTC [2] is an emerging standard from W3C allowing live peer-to-peer media streams to be shared between browsers without requiring an intermediary server. It relies on two HTML5 specifications, getUserMedia to get browser access to the web camera and microphone on the device and PeerConnection to share these media streams between browser clients in a web conference. The PeerConnection API relies on an offer and answer protocol referred to as JSEP which in turn relies on SDP (offer and answer payload schema) and ICE (peer IP and port discovery). ICE in turn relies on STUN [6] (NAT traversal) and TURN [5] (NAT relay). STUN simply tries to expose browsers behind NATs with the NAT IP whereas TURN relays messages through a third-part server typically on the public Internet. In TURN ports are allocated by the relay server and the communication is "turned" to the direction from the outside Internet to the peer behind the NAT. Thus it serves as a ephemeral public access point for the duration of the media session. It is not obvious how to set up or configure these TURN servers through complex corporate networks, in particular firewall traversal becomes problematic since ports are allocated on demand and there is one port that needs to be burned through the firewall for each active peer. The current WebRTC implementations don't make use of TURN out-of-the-box for these reasons. Unfortunately most corporate environments have very restrictive firewall policies and would not allow one port to be opened per peer. This is the case for our hospital setting, and thus we developed a new firewall traversal protocol that is designed to incur as little overhead as possible, and at the same time be easy to deploy. If it is more complicated to deploy than installing a Skype client we have lost part of the purpose of web video conferencing. Our solution only requires a single port to be opened corporate network wide for all the media session both coming in and going out. It does require the deployment of a gateway separately from the Web server and web browsers. But only one gateway is needed per local network that all browsers on that network can reuse. It is also very easy to deploy and could potentially be deployed automatically, e.g. in a corporate cloud since it only relies on a JVM to be installed. For home users behind firewall on their own DSL/Cable Wifi network it could become more of a burden to install a gateway and you could lose part of the benefit of the no-deployment, no-plugin web conferencing envisioned with WebRTC. For this scenario we allow the clients to use remote gateways, but then we are back to the issue of many ports being open to the outside world. Most such home networks only have NATs though. In the case of a very restrictive home firewall we could provide on-demand remote gateway provisioning where one IP is allocated in the public cloud on a short lived VM per peer and session on a well-known port. This is however future work, as it would require some billing structure too to buy or rent virtual machines from a cloud provider.

Now to our solution in more detail. It comprises three components, a tunnel, an rtc gateway service and a central rtc service (hereafter simply referred to as the rtc service). The rtc gateway services are typically deployed within local networks behind firewalls and the rtc service is deployed on the public Internet. The tunnel is responsible for multiplexing the traffic that comes into it through a single well known port to another tunnel peer which would sit on a different network such as the public Internet. One could imagine tunneling directly between the peer networks but that makes the configurations more volatile and it may also be a security issue to give access directly into services running in a remote local network (such as a hospital or corporate network).

To simplify deployment we by default deploy the rtc service in the default device hooked up to our long polling Web server so that only local gateway configurations need to be set to establish a session. If the rtc service detects that both peers use the same local gateway no gateway redirection will be perfomed and the standard WebRTC protocol will be used. However if the peers have two different local gateways the following setup logic is applied:

1. The webRTC runtime in browser A generates an SDP (Session Description Protocol) offer through JSEP (Javascript Session Establishment Protocol) containing among other things an IP and port where it wishes to receive remote media traffic (RTP and RTCP packets).

2. We now intercept the offer in Javascript locally before it is broadcast to potential peers. The signalling plane of WebRTC is not defined in the spec but could treat the JSEP protocol as a black box, i.e. no knowlage of payload is necessary, you just need to pass the data from the WebRTC runtime in browser A to browser B by some means. This makes it easy for us to attach additional information to the signalling payload without the knowledge of the respective webRTC runtimes while still being fully compliant to the JSEP handshake protocol. The information we add is simply the identity of the local rtc gateway.

3. The enhanced offer is now passed through the rtc service and broadcast to everyone listening on the channel where browser A sent the offer.

4. When browser B receives an offer payload we intercept the SDP data before it reaches the WebRTC runtime and rewrites it using the browser B rtc gateway. This is done as follows. The host and port of the offer is extracted together with the remote (browser A in this case) rtc gateway. A remote allocate call is then issued to the rtc service containinng, browser A port,host,gateway and browser B gateway. This remote allocate command is channeled through the tunnel that was set up on the browser B network to the browser B rtc gateway. The browser B gateway then allocates a port for the incoming session on a lease basis and passes back its own IP together with the allocated port. The idea is that all traffic on that port will be forwarded through the central service to the correct remote rtc gateway service where it will be sent out to the IP and port that were in the original SDP offer that the WebRTC runtime in browser A generated. Now browser B will get the new IP and port back pointing the the local gateway and the SDP will be rewritten with this info as the remote peer data channel.

5. The browser B webRTC runtime will now generate an SDP anwser in accorance with JSEP that will follow the exact same procedure as the offer with first attachment of local gateway and then a remote allocate call now on the browser B side of the network before it reaches the browser B runtime.

The general architecture of the firewall setup process is depicted in Figure 3.

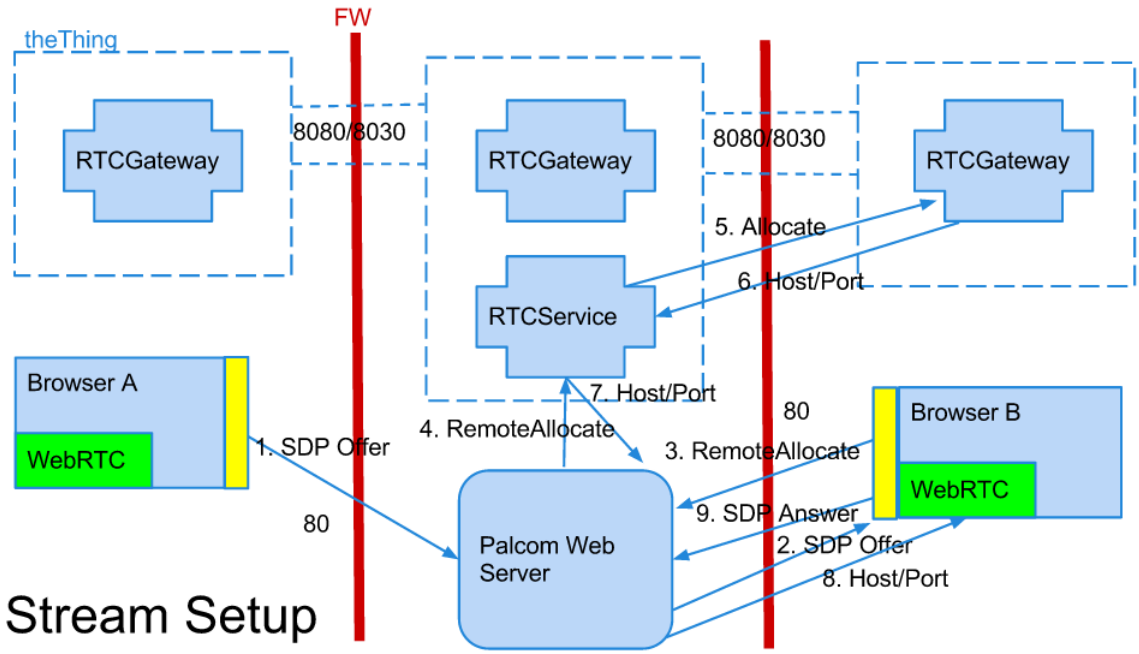

Figure 3: Setting up a firewalled webRTC session throught the Palcom tunnel.

After the session has been established following the above procedure the relay is ready to transfer data packets from browser A to the local rtc gateway through the tunnel to the central server through the second tunnel to the rtc gateway in the browser B network and vice versa for data packets from browser B to browser A. There is only one missing piece, the webRTC runtime does not send data to a remote IP and port socket before it has been authenticated with a series of STUN messages. These STUN messages are also used continously as a keep-alive mechanism and as soon as they don't receive a reply the data traffic stops. So our rtc gateways also have to implement the STUN protocol used by the webRTC runtime. The good news is that all information needed to generate both STUN responses and STUN requests required by the authentication and keep-alive handshakes is already available in the RTC gateways. The bad news is that there is a chicken and egg problem where browser A does not send out data traffic until it has sent STUN requests where it got valid replies and also received STUN requests from the remote browser. At the point where it both got back STUN responses and received compatible STUN requests data

traffic must be generated immediately for the channel not to be rejected. Hence we need not only reply with the correct STUN responses to incoming STUN requests and generate compatible STUN requests at the correct time in the handshake but we need to also be aware of when the remote browser is actually ready to send data packets. This apparent chicken and egg scenario is resolved by the fact that the browsers generate data packets a bit in advance before the full authentication has happened as a test. However if we don't manage to establissh the channel quickly enough the attempt will be dropped and the data traffic will stop, and once it stops on one end it stops on both ends (both from browser A and browser B). This complicated dance has a relatively simple solution. Our gateway replies to STUN messages properly for a period of time and also generates STUN resquest so that the local browser will start sending test data packets. After some point it stops replying to STUN messages to allow the remote browser to react. Now when the same test data packets start showing up from the remote browser the local gateway starts replying to STUN messages and generate STUN requests again to authenticate the channel and be compliant to the keep-alive protocol. The nice thing about this solution is that the very high frequence STUN messages never have to travel beyond the local network and never have to be sent through the tunnels to the other peer. One word of caution is that the STUN heartbeats may be used to detect the latency of the connection which of course won't be accurate in our case since the STUN pings only happen between the browser and the local rtc gateway. For this reason we also allow a delay to be injected in the STUN replies to simulate a remote peer over a slower connection. In practise this is however rarely needed. We are currently only aware of one limitation of this protocol. The handshake does not seem to work over ssh port forwarding setups. It does work reliably over both wired, WiFi and 3G networks, though. And most importantly the audio and video packets stay in sync thanks to the webRTC use of RTP packets where both audio and video frames are synced at the source. We should also mention that webRTC uses a single port for RTP and RTCP instead of the usual odd and even port numbers to simplify NAT traversal and this also helps us in the firewall traversal setup. Otherwise the same setup would have to be duplicated for each of the two ports. Finally the WebRTC RTP payloads are all encrypted so one only needs to make sure that the SDP singalling plane is encrypted where the credentials are included to secure the protocol. This means that if someone else receives a data packet. e.g. by sniffing our tunnel or rtc gateway traffic they cannot do anything with it. We also have the option of encrypting our tunnel traffic, but in this case it is overkill. To encrypt the signalling traffic we simply rely on HTTPS which is supported by our long polling implementation trivially since it only relies on the browser XHR (XMLHttpRequest aka AJAX) API.

Figure 4 shows how RT(C)P UDP traffic flows through the WebRTC browser runtimes and the Palcom tunnel during a audio/video conference session.

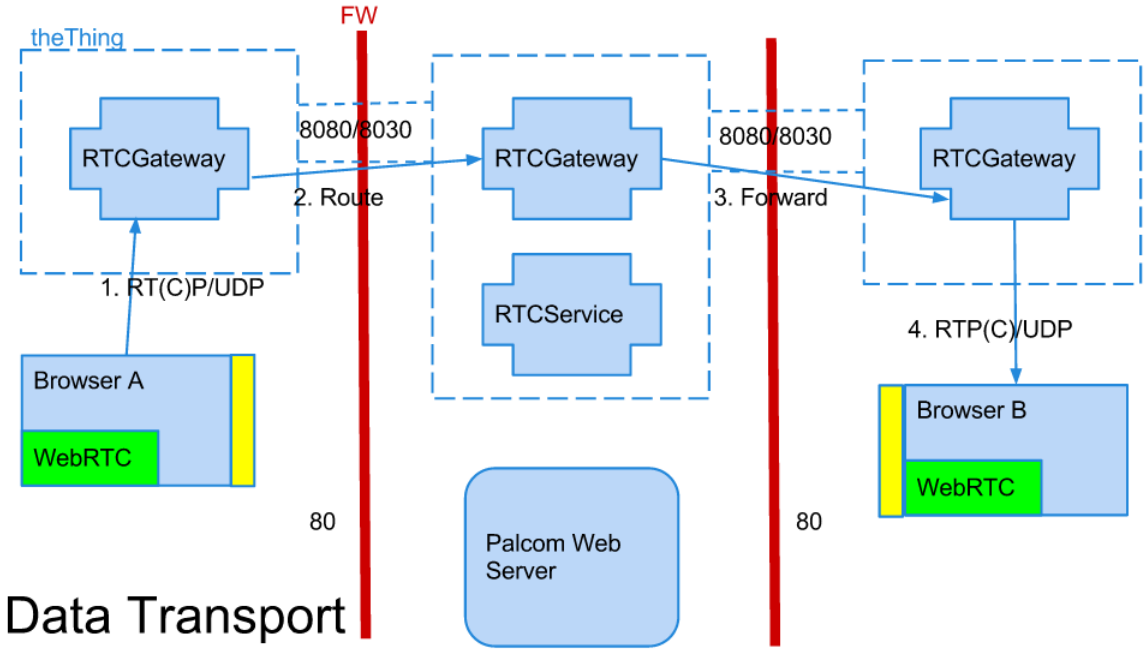

Figure 4: Data traffic throught the Palcom Tunnel and WebRTC browsers.

5. CONCLUSIONS

We have described three parts of the Palcom Web bridge toolkit that makes it easy to build rich web clients to communicate with Palcom device services such as medical pumps. The main contributions of this work comprise:

- a long polling solution with server side filtering and history features,

- a web widget tool-box for real-time data stream monitoring and device control, and
- a firewall traversal service to tunnel WebRTC audio and video data traffic across networks protected by NATs and firewalls.

The main challenges we addressed were how to communicate efficiently between devices and web front-ends, how to build rich web clients for monitoring and controlling devices in real-time, and how to facilitate real-time audio and video conferencing through secure networks such as a hospital firewall.

To our surprise the browser runtime was very efficient in processing and rendering large amounts of streamed data. 2000 datapoints rendered in our timeseries widget every 2 seconds showed some of the opportunities of this solution. If more data needs to be streamed in real-time simple data reduction techniques such as averaging may be applied.

The second positive surprise was the audio/video data stream tunneling performance. Despite a fairly complex route through networks and tunnels, the WebRTC protocol was very resilient to changing rates and the impact on the user experience was negligible. We were even able to run the traffic through a 3G cellular network with acceptable performance. WebRTC does however require a lot of CPU power to run smoothly. 5-6 years old Windows XP PCs that we still could run on given our cross-platform design were a bit too slow to give a good user experience. On the other hand, modern laptops such as MacBook Air and MacBook Air Pro provided an excellent audio and video experience.

Future work includes integrating our work in the hospital field study and providing custom solutions for smaller form factors such as smartphones and tablets.

Acknowledgments

We would like to thank John Sturk, Lars Nilsson and Karl Kullberg for their help with testing and developing some of the infrastructure our work relies on.

6. REFERENCES

[1] J. Aubourg, J. Song, and H. Steen. Xmlhttprequest. Editor's draft, W3C, Dec. 2012. http://www.w3.org/TR/XMLHttpRequest/.

[2] A. Bergkvist, D. Burnett, C. Jennings, and A. Arayanan. Webrtc 1.0: Real-time communication between browsers. Working draft, W3C, Aug. 2012. http://www.w3.org/TR/webrtc/.

[3] D. Crane and P. McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, Berkely, CA, USA, 2008.

[4] I. Hickson. Html 5.1 nightly. Editor's draft, W3C, Dec. 2012. http://www.w3.org/html/wg/drafts/html/master/Overview.html.

[5] R. Mahy, P. Matthews, and J. Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). Internet rfc 5766, IETF, Apr. 2010. http://tools.ietf.org/html/rfc5766.

[6] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for nat (stun). Internet rfc 5389, IETF, Oct. 2008. http://tools.ietf.org/html/rfc5389.

[7] D. Svensson Fors, B. Magnusson, S. Gestegård Robertz, G. Hedin, and E. Nilsson-Nyman. Ad-hoc composition of pervasive services in the palcom architecture. In *Proceedings of the 2009 international conference on Pervasive services*, ICPS '09, pages 83–92, New York, NY, USA, 2009. ACM.