# Web(-like) Protocols for the Internet of Things

Dr. Emmanuel Frécon
ICE - Interactive Collaborative Environments Laboratory
SICS Swedish ICT
emmanuel@sics.se

**Abstract**

This paper describes the motivation for and implementation of two Web-oriented protocols for the Internet of Things. While building upon the ubiquity and flexibility of the Web, WebSockets and STOMP minimise headers to the benefit of payload, thus adapting to the scarce resources available in IPv6 mesh networks. The websocket library provides a high-level interface to open client connections and further exchange data, along with the building blocks necessary to upgrade incoming connections to a WebSocket in servers. The STOMP library offers a near-complete implementation of the latest STOMP specification, together with a simple broker for integration in existing applications.

## 1. Introduction

The Internet of Things promises a near future where domestic and work environments, but also cities and factories, are augmented with sensors and actuators that all are Internet entities. The deployment of IPv6 together with mesh networks is key to this evolution by enabling each sensor or actuator to be accessed from any Internet enabled application or user, thus from almost anywhere. Given the widespread deployment of Web technologies, extending their reach to sensors and actuators would facilitate their integration in the fabric of our lives.

Being “everywhere” imposes a number of technological and economical constraints on sensor networks, leading to few radio, storage and computing resources available at each sensor. In particular, keeping protocol-relevant state to ensure the reliability of TCP imposes challenges to mesh networks since transmitting a packet reliably will require the resources of several sensors in the mesh. In addition, radio constraints impose small packet sizes, leaving little space to application data and protocol headers.

This paper describes the implementation of two Web(-like) protocols: WebSockets [1] and STOMP [2], targeting their integration in applications based on IPv6 mesh networks [3]. While being based on TCP, both protocols exhibit features making them suitable for low-level communication within a whole application deployment: from tiny sensors to users, via cloud services. WebSockets is merely a way to upgrade a regular Web connection to a lightweight and symmetrical communication channel, making possible data flows in both directions between the server and client. Within a WebSocket connection, individual packets are led by small binary headers. STOMP is also a bidirectional protocol, but with textual headers following rules similar to regular HTTP, and kept to a minimum.

## 2. Motivation

Most currently available sensor networks technologies use solutions for the "last mile" that are unlike the remaining of the Internet. Z-Wave[1], ZigBee[2], BTLE (bluetooth low energy)[3], EnOcean[4] are a few examples of these standardised radio and network technologies that have reached penetration in consumer and industrial applications. Being different from the transport protocols over which the Internet is built requires that information flowing from and to the sensors and actuators needs to be translated within bridges connected to both incompatible "worlds". This transition adds complexity to application development and deployment while introducing possible points of failure. There exists a number of middleware platforms [4] [5] [6] that attempt to hide the variety of sensor and actuator technologies. However, using these also introduces some complexity or at least programming and architectural models that are not always suitable for the final application at hand [7].

The motivation for this work is to experiment with ways to run the IP-family of protocols all the way from the clients to the sensors, via the server architecture. Both of the protocols which implementations are described in this paper are directly related to HTTP, the application protocol that forms a large part of the Internet today. WebSockets upgrade a regular web connection to a duplex, binary communication channel, while STOMP is a duplex mainly textual protocol sharing a lot of similarities with HTTP. Implementing both of these protocols has been used to understand the low-level technicalities and details involved. In itself, this implementation provides an assessment as to whether WebSockets and STOMP are suitable for an implementation on top of the constrained resources that are common place at the edges of sensor networks.

Finally, both protocols share a number of similarities with stream protocols in the sense that they sustain a long-lived duplex connection between the initial client and the server. While maintaining the state of this connection poses requirements on the underlying radio infrastructure, and especially on intermediate nodes in mesh networks, it also avoids the bursts involves in (re)creating the connections between sensors whenever information has to

[1] Information about Z-Wave can be obtained from the Z-Wave Alliance home page at http://www.z-wavealliance.com/ or from wikipedia at http://en.wikipedia.org/wiki/Z-Wave.
[2] Specifications for the ZigBee standards are available from the ZigBee Alliance home page at http://www.zigbee.org/, more information is available at http://en.wikipedia.org/wiki/ZigBee.
[3] BTLE is part of the specifications for Bluetooth 4.0 available at https://www.bluetooth.org/en-us/specification/adopted-specifications, more information is also available at http://en.wikipedia.org/wiki/Bluetooth_low_energy
[4] The full title of the EnOcean standard is: ISO/IEC 14543-3-10 Information technology -- Home Electronic Systems (HES) -- Part 3-10: Wireless Short-Packet (WSP) protocol optimized for energy harvesting -- Architecture and lower layer protocols, more information available at the EnOcean home page at http://www.enocean-alliance.org/en/home/.

reach the edges while ensuring a higher level of interactivity.

## 3. Background and Related Work

As described in the previous section, there exist a number of communication protocols targeting or suitable for the Internet of Things. IEEE 802.15.4 based protocols such as ZigBee and 6LoWPAN, Z-Wave, EnOcean, ANT[5], BTLE or even the IEEE 802.11 family of protocols (wifi) are all wireless protocols, operating either in the sub-gigahertz frequency ranges or the crowded 2.4GHz radio spectrum. Given the heterogeneity of the offering, the IEEE is pushing IPv6 and 6LoWPAN, a standard that would allow running IP, the Internet Protocol at all levels.

In this context, CoAP [8] is a proposed standard over UDP that specifically addresses the requirements of the constrained nodes carrying sensors and actuators in many deployments. These nodes usually run specialised operating systems such as Contiki [9] or TinyOS [10]. CoAP focuses on a binary HTTP-inspired open protocol for the implementation of RESTful environments, providing features such as discovery of resources and subscription for a resource (resulting in push notifications). In a similar fashion, MQTT-S is a cutdown version of the MQTT[6] (Message Queuing Telemetry Transport) protocol geared towards lossy radio protocols. MQTT is a protocol proposing a unified publish-subscribe approach for M2M communication. MQTT is a lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or reliable networks.

Apart from MQTT, there exist a number of more generic protocols implementing publish-subscribe patterns, STOMP and AMQP [11] (Advanced Message Queuing Protocol) being two well-known examples. Generic broker services such as RabbitMQ or ActiveMQ use these protocols to ease and scale the deployment of modern cloud services. Some of them make use of WebSockets to make it possible to use queuing protocols such as STOMP or AMQP directly from the web browsers. The combination of all these protocols allows for a unified approach to application development and deployment: from tiny sensors and actuators to clients, via scalable and elastic cloud services.

WebSockets will  upgrade a regular HTTP connection, as initiated from the client, to a duplex binary long-lived connection. As such, WebSockets will typically present an initial textual header of “subsequent” size following the rules of the HTTP specification, together with the handshaking specified in RFC 6455. Consequently, initiating a WebSocket connection will typically represent a burden on an existing sensor network, while later packets will allow applications to use most of the network packet size for their data. As WebSocket technically upgrade an existing web connection, they are typically suitable for passing through firewalls whenever needed. In addition, the specification requires a heartbeat facility that will keep the connection alive through, for example, network equipment implementing NAT solutions. The work described in this paper is partly based on a server-side implementation of the

[5] ANT is a wireless protocol mostly targeting the sports and outdoor activity domain. More information is available at http://www.thisisant.com/.
[6] MQTT has been proposed as an OASIS standard, the specifications are openly published and made available at the following web site: http://mqtt.org/.

WebSocket protocol that was made available as part of the Tcl web server wibble[7].

STOMP stands for the Simple (or Streaming) Text Oriented Messaging Protocol. The protocol provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers. STOMP is loosely modeled after the HTTP protocol: messages will typically start with an uppercase command (like `GET` in HTTP), followed by a number of textual headers in a format similar to HTTP and possibly followed by a body. Connections are client initiated, but messages can flow in both directions. STOMP, similarly to WebSocket specifying heartbeating mechanisms to keep connections opened at all time. However, heartbeating is handshaked at connection time, allowing for all types of situations, including applications where no heartbeating is required. The core of STOMP specifies publish-subscribe mechanisms where clients subscribe to topics (compare to a path in HTTP) and will be notified whenever messages have been sent to that topic. STOMP specifies a number of delivery mechanisms in order to implement different degrees of reliability. In the context of sensor networks, the shortness of commands and headers leaves enough room to application data. Together with implementations in, for example, JavaScript this opens for the use of STOMP at all levels of an architecture, including at the edges formed by the browser or the sensors. tStomp [12] is a SNIT-based client library partially implementing v 1.1 of the specification in Tcl. tStomp provides a basic level implementation, leaving aside some features such as authentication, virtual hosting, receipts, message acknowledgements or heart-beating.

## 4. WebSockets

The `websocket` library is a pure Tcl implementation of the WebSocket specification covering the needs of both clients and servers. The library offers a high-level interface to receive and send data as specified in RFC 6455, relieving callers from all necessary protocol framing and reassembly. It implements the ping facility specified by the standard, together with levers to control it. Pings are server-driven and ensure the liveness of the connection across home (NAT) networks. The library has a number of introspection facilities to inquire about the current state of the connection, but also to receive notifications of incoming pings, if necessary. Finally, the library contains a number of helper procedures to facilitate the upgrading handshaking in existing web servers.

Central to the library is a procedure that will “take over” a regular socket and treat it as a WebSocket, thus performing all necessary protocol framing, packetisation and reassembly in servers and clients. The procedure also takes a handler, a command that will be called back each time a (possibly reassembled) packet from the remote end is ready for delivery at the original caller. Another procedure allows callers to send data to the remote end in transparent ways. The procedures can be used from both servers and clients, even though the protocol is not entirely symmetric. A great deal of the implementation of these procedures originates from the initial wibble implementation.

[7] Original wibble code is available on the wiki at http://wiki.tcl.tk/26556.

Even though these procedures form the core of the library, they will seldom be used as clients and servers can benefit from a second layer of API at a higher level. Typically, clients will open a connection to a remote server by providing a WebSocket URL (`ws:` or `wss:` schemes, `wss:` being, as `https:`, encrypted) and the handler described above. The opening procedure is a wrapper around the latest `http::geturl` implementations: it arranges to keep the socket created within the `http` library opened for reuse, but confiscates it from its (internal) map of known sockets for its own use. Therefore, it will only work with the versions of the `http` library that have support for version 1.1 of the HTTP specification.

The following example arranges to connect to the echo server, send a simple text message and to finally close the connection and exit on reception of the message that it had sent to the echo server. The code also exhibits some of the introspection facilities that are built in the `websocket` library.

```
package require websocket

# Maximise loglevel to get detailed information on the console.
::websocket::loglevel debug

# Handler procedure registered as a receiver of all information
# from the server and internal state changes of the library.
proc handler { sock type msg } {
    switch -glob -nocase -- $type {
      co* {
          # Once connected, arrange to show off introspection
          # facilities and to send a simple text message to the
          # echo server.
          puts "Connected on $sock"
          test $sock
      }
      te* {
          # On reception of the text message that we have sent,
          # print it out and disconnect.
          puts "RECEIVED: $msg"
          ::websocket::close $sock
      }
      cl* -
      dis* {
          # Exit on disconnection, i.e. as soon as we've received
          # the message that we had sent to the echo server.
          exit
      }
    }

}
```

```
# Send a single message to the echo server after having printed
# out information about the current websocket connection.
proc test { sock } {
    # Print connection info
    puts "[::websocket::conninfo $sock type]:\
          [::websocket::conninfo $sock sockname] ->\
          [::websocket::conninfo $sock peername]"

    # Send message to echo server.
    ::websocket::send $sock text "Testing, testing..."
}

# Open connection to the echo server and arrange for the handler
# procedure to be called whenever receiving information from the
# server or about the connection.
set sock [::websocket::open ws://echo.websocket.org/ handler]

# Arrange for the event loop to kick-off
vwait forever
```

For servers, the process is slightly more involved in order to account for the differences of their inner workings. Indeed, as the library wishes to take over the socket, it needs to interface with the remaining of the server code to know at which point of the transaction the overtaking should occur. Servers will start by registering themselves and especially their handlers (see central procedure above) to the library. Then for each incoming client connection, they should test the incoming request to detect if it is an upgrade request by providing the path and HTTP headers to a testing procedure. Finally, a server will perform the final handshake to place the socket connection under the control of the `websocket` library and its central procedure. This is exemplified in the following pseudo-code.

```
# Assuming the server is listening on connections on srvSock,
# create a handler for the webserver socket
set srvHandler [::websocket::server $srvSock]

# Provide matching between paths and handlers, this would match
# any path to be handled by the handler command, but there can
# be any number of such declarations, for different paths at
# the server.
::websocket::live $srvSock * handler

...

# At a later time, when a client has connected at sock and all headers
# for the incoming connection have been parsed and are made
# available in the headers dictionary, test if this is a websocket
# upgrade request. If it is, upgrade the socket to a websocket.
if { [::websocket::test $srvSock $sock $url $headers] } {
      ...
      # Upgrading the socket will arrange to call the handler
      # depending on the path and place the whole socket under
      # the control of the library. This also means that the
      # library will send back some handshaking data on the
      # client socket.
      ::websocket::upgrade $sock
}
```

The `websocket` library is, since a few months back, part of `tcllib`. The mini web server that is part of the `efr-tools`[8] is able to hosts websockets and can serve as a more elaborate example of how servers and the library should interface in order to provide support for WebSockets.

## 5. STOMP

The STOMP library provides both a client side implementation and a minimal server broker in pure Tcl. Coding constructs are constrained so as to be able to run under jTcl [13] if necessary. The library is composed of a three sub-packages: a messaging package provides core facilities for creating, parsing and sending STOMP messages; the client package is a near-complete implementation of all versions of the protocol and the server package provides a minimal broker implementation in pure Tcl, ready for integration in any existing application.

The messaging package provides procedures to create messages, set and get their various parameters and components. STOMP Messages are composed of a command, a set of headers (loosely similar to HTTP headers) and a body. The library makes it possible to strictly follow the message constraints as imposed by the protocol, i.e. control which header are mandatory and optional for a given command. The messaging package also offers procedures to send and receive messages to and from a STOMP server. Reception is implemented via a "reader" context as messages can arrive at any time from a server on a socket. The reader context contains a handler that will be called once incoming messages have been parsed and validated. Finally, the messaging package can keep alive a connection by sending (almost) empty data at regular intervals. It watches that a remote end continuously sends heartbeats, and provides a callback whenever the remote is deemed to have disconnected, i.e. when no heartbeats have been received for a given time period.

The client package encapsulates all traffic to a remote server into a single object. The layer provides an implementation that follows the v1.2 of the specification, except that it does not provide a high-level support for transactions. The library is able to open connections to remote servers, while respecting the initial connection handshake that will establish how heartbeats will be exchanged between the client and the server. It also implements facilities

[8] `efr-tools` is a google code project hosting both a number of libraries, but also the context manager [14], a cloud service for IoT applications. The project is available at https://code.google.com/p/efr-tools/ and it contains an experimental copy of the websocket library.

to keep the connection alive at all times, continuously attempting to reconnect whenever the connection has been lost. The client library provides callbacks to follow the current state of the connection (in progress, connected, disconnecting, etc.), but also to be notified of the reception of any message. Sending messages to the remote end is a high-level wrapper around the messaging package, a wrapper that will take care of reconnections, header assignments, etc. The client package offers a procedure to subscribe to data coming from the server and to arrange for callbacks to be delivered each time a message matching the subscription is received from the server. Subscriptions are represented by a context and can be ended at all time, in cooperation with the server. Finally, the client library is able to request for receipts, to be called back on reception of the receipt and to perform graceful disconnections from the server, i.e. disconnections in cooperation with the server. A basic level of security is ensured by supporting authorisation, but also by some basic security mechanisms against packets that would be too long.

The following code exercise the library by connecting to a remote server, subscribing to a topic, sending messages and verifying that even binary messages can transit between clients and servers.

```
package require stomp::client

# This code supposes that the global variable dta contains the
# binary content of a Windows .ico file (anything else would do)
# and will check that what has been sent to the server is also
# receive, even when sending larger binary messages.

# Print out reception acknowledgment
proc received { cid rid } {
    puts "Server has received message, id: $rid"
}

# Handler called on reception, print the command and headers
# contained in the message, and if it was an icon, compare it
# to what we had sent. This provides examples for how to use
# the messaging package part of the library.
proc incoming { msg } {
    global dta
    puts "Incoming message: [::stomp::message::getCommand $msg]"
    set type [::stomp::message::getHeader $msg content-type]
    if { $type eq "image/x-icon" } {
        puts "Comparing: [string eq $dta [::stomp::message::getBody $msg]]"
    }
}

# Unsubscribe from the only topic that we’ve ever subscribed to,
# which demonstrates how to look for existing subscriptions if any
proc stop {} {
    global s
    set sub [lindex [::stomp::client::subscriptions $s /queue/a] 0]
    ::stomp::client::unsubscribe $sub
}

# Connect to a STOMP server, these credentials are the default one
# on the Apache Apollo implementation of STOMP
set s [::stomp::client::connect -user admin -password password]

# Arrange to stepwise in the near-future, subscribe to a queue,
# send a text message, send a binary message, unsubscribe from the
# topic, verify we've stopped listening for that topic and finally
# to disconnect.
set ops [list \
       1000 [list ::stomp::client::subscribe $s /queue/a \
                       -handler incoming] \
       2000 [list ::stomp::client::send $s /queue/a -body "Hello world" \
                       -receipt received] \
       4000 [list ::stomp::client::send $s /queue/a \
                       -type image/x-icon \
                       -body $dta] \
       6000 stop \
       7000 [list ::stomp::client::send $s /queue/a -body "Gone away?"] \
       30000 [list ::stomp::client::disconnect $s] \
       ]

foreach { when what } $ops {
    after $when $what
}
vwait forever
```

Similarly to the client library, the server broker implements heart-beating and simple message receipts. The broker will fan out incoming messages to all clients that have expressed interest via a subscription. It implements basic security mechanisms to discard messages from non-connected clients, to discard messages that would be too long or to check for user authorisation. Finally, the server supports virtual hosting.

## 6. Conclusion and Future Work

This paper has described the design and implementation of two Tcl libraries targeting the use of Web(-like) protocols in sensor networks applications. The implementation served as a an assessment of the suitability of the protocols to the few resources available on sensors. As most sensor networks use 16-bits microcontrollers, running Tcl on top of these hardware constrained devices is difficult[9]. However, the implementations discussed in this paper have been tested for longer active sessions on small hardware platforms such as the

[9] Running Tcl(-like) languages on small hardware platforms can however be achieved through languages such as Jim [15] or Hecl (see http://www.hecl.org/).

BeagleBoard-XM[10] or the Raspberry Pi[11], both as servers and clients. Given the low price tag and the presence of GPIOs on the Raspberry Pi, this opens up for the use of Tcl at all levels of sensor applications, i.e. not only in the cloud services and clients, but also within sensors. Stepping away from Tcl, C-based WebSocket client code has already been integrated into the Contiki OS and is part of Thingsquare Mist, thus demonstrating the suitability of WebSockets as a low-level protocol down to the sensors. WebSockets exhibit a larger header at connection establishment, while keeping protocol overhead to a minimum in subsequent message exchanges. STOMP uses plaintext headers, though keeping them to a minimum which makes it a possible alternative to WebSockets if ever needed. At the time of writing, the STOMP library is in the half-bakery on the wiki[12] and integration to the context engine [14] is in progress.

This work has been sponsored by the ARTEMIS project me$^3$gas and the Swedish project Intelligent Energy Services. Many thanks to all colleagues at SICS and especially Joakim Eriksson for lively and interesting discussions along the development of these libraries and their purpose. Finally, hats off to Andy Goth for the initial version of the WebSocket implementation and to Immanuel Patzschke for bug testing and fixing of the WebSocket library.

## 7. References

[1] "*The WebSocket Protocol*", I. Fette and A. Melnikov, RFC 6455, IETF.

[2] "*STOMP Protocol Specification*", available at http://stomp.github.io/stomp-specification-1.2.html.

[3] "*IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*", N. Kushalnagar, G. Montenegro, C. Schumacher, RFC 4919, IETF.

[4] "*A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems*", M. Eisenhauer, P. Rosengren, P. Antolin, Proceedings of the 6th Annual IEEE Communication Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks Workshops, June 2009.

[5] "*Ad-hoc Composition of Pervasive Services in the PalCom Architecture. In Proceedings of International Conference on Pervasive Service*" D. Svensson Fors, B. Magnusson, S. Gestegård Robertz, G. Hedin and E. Nilsson-Nyman, Proceedings of ACM CPS09, London July 2009.

[6] "*Zero-programming Sensor Network Deployment*", K. Aberer, M. Hauswirth, A. Salehi, Proceedings of the IEEE Service Platforms for Future Mobile Systems (SAINT 2007), Japan, 2007.

[7] "*Role of Middleware for Internet of Things: A Study*", S. Bandyopadhyay, M. Sengupta, S. Maiti, S. Dutta, International Journal of Computer Science & Engineering Survey (IJCSES), Vol. 2, no. 3, August 2011.

[8] "*Constrained Application Protocol (CoAP)*", Z. Shelby, K. Hartke, C. Bormann,

[10] BeagleBoards are a family of low-cost, fan-less single-board open source computers featuring an ARM cortex core and expansion facilities. More information about the boards is available at http://beagleboard.org/.

[11] The Raspberry Pi is comparable to BeagleBoards, but packages a less powerful ARM core to an ever lower price. More information available at http://www.raspberrypi.org/.

[12] More information about the various stomp implementations can be found at http://wiki.tcl.tk/38103.

draft-ietf-core-coap-18, June 2013.

[9] “*Contiki - a lightweight and flexible operating system for tiny networked sensors*”, A. Dunkels, B. Grönvall, T. Voigt, Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, Nov. 2004.

[10] “*The Emergence of Networking Abstractions and Techniques in TinyOS*”, P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, D. Culler, Proceedings of the 1st USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI), 2004.

[11] “*Advanced Message Queuing Protocol*”, S. Vinoski, IEEE Internet Computing, Vol. 10, no. 6, Nov/Dec 2006, pp 87-89.

[12] “*New Siemens Open Source Contribution for TCL: tStomp Messaging library. Why and how to use TCL in a very large scale Software Development*”, D. Münchhausen, Talk at EuroTcl’2012.

[13] “*JTcl and Swank: What’s new with Tcl and Tk on the JVM*”, B. Johson, T. Pointdexter, D. Bodoh, Proceedings of the 18th Tcl/Tk conference, Manassas, 2011.

[14] “*Bringing Context To the Internet of Things”*, Emmanuel Frécon, Proceedings of the 19th Tcl/Tk Conference, Chicago, Sept. 2012.

[15] “*Jim Tcl - A Small Footprint Tcl Implementation*”, S. Bennett, Proceedings of the 18th Tcl/Tk Conference, Manassas, 2011.