biot - Information Pipelines in IoT-Clouds

Dr. Emmanuel Frécon
SSE - Software and Systems Engineering Laboratory
SICS Swedish ICT
emmanuel@sics.se

Abstract

There are a plethora of (proposed) standards and proprietary solutions for the Internet-of-Things: from protocols, to data modelling, and abstractions. Integrating applications struggle to move data between these silos. Biot implements a generic network-aware information pipe to move data from anywhere to anywhere: from devices to the cloud (sensing), from the cloud to devices (actuating) but also between clouds (integration). Biot is tuned to be embedded in modern cloud architectures based on micro-services and has been deployed within CoreOS and Docker Swarm clusters.

1. Introduction

The Internet of Things promises a near future where domestic and work environments, but also cities, factories or even human bodies, are augmented with sensors and actuators that all are Internet entities. The real power of these technologies relies in their communicating abilities, thus in applications that are able to combine and connect sensor data from different sources and of different types in order to provide an enhanced experience or an additional service.

An example set in the domestic domain would be the combination of temperature sensors in fire detectors, of sensed external climate data from a neighbouring weather station and weather forecasts to tune the functioning of the heating system. When the heating system is heavily dependent on the electricity grid, when a ground heat pump is used for example, tuning can further be enhanced to accommodate the current load on the electricity grid and introduce slight delays or anticipate inner temperature increases to better spread out grid demand on a larger scale. The realisation of such a scenario requires complex data flows between a number of actors and organisations, including business models that respect personal integrity and generate value at several levels of the data flow.

This paper presents biot, the "Bridge for the Internet of Things", a flexible tool for bridging the

data silos that otherwise occur when vendors try to lock in customers in proprietary solutions covering sensors and actuators of various sorts and their corresponding mobile and web apps. While biot is tuned to create information flows between heterogeneous APIs within web clusters, it has also been used closer to the wireless sensor networks, in gateways or similar. Simply put biot accepts a number of sources, acquires data from these sources, extracts any number of variables from this data and pushes templates using these variables to any number of destinations. In other words, biot is an easy API masher for the Internet of Things. Biot uses domain-specific configuration files to describe its sources, variables and destinations, while template generation relies on the flexibility of the Tcl language to control the output to destinations. Biot supports a wide-range of protocols for its sources and destinations and can be encapsulated in Docker[1] components for scalability and security.

2. Motivation and Background

As the Internet-of-Things is gaining momentum and as connected devices are being deployed on bodies and in homes, offices, factories or cities, many competing (and sometimes similar) technologies and solutions are being deployed. There are several radio and (meshing) protocols: ZigBee [1], Z-Wave [2], BLE (aka. Bluetooth Smart) [3], 6LoWPAN [4] to mention a few; but also several transmission protocols: HTTP [5], CoAP [6], WebSockets [7], MQTT [8], XMPP [9] and several data modelling protocols: LWM2M [10], XMPP, WoT [11], AllJoyn [12]. In addition, there are a number of proprietary solutions, most of which with an open and/or well-specified API to send or receive data to and from the sensor networks.

Given the current pace of IoT penetration, integrating legacy deployments into new solutions already requires attention. Key to this integration is the ability to move data between the different layers, protocols and APIs, or to map abstractions onto one another. Fortunately, most of these protocols and specifications share a number of common points at the data level. For example, JSON [13] and to a lesser extent XML [14], are prevalently used for the description of sensor or actuator data. Different compression, minification or translation techniques are being put in place to adapt to the resource constraints of sensor networks while still building upon the flexibility of structured data representations.

Biot is a network-aware information pipe for the IoT. Biot aims at easily bridging together the various APIs that are involved in IoT projects. There are other existing projects with similar goals. For example, the core of Node-RED [15] consists of hiding the details of acquiring/sending data to things and API. However, Node-RED focuses more on the visual creation of IoT applications, while biot mainly targets cloud applications with high data throughput. Biot also shares some similarities to the DSA Links[2] of the IOT-DSA platform [16].

---

[1] Docker is a popular operating system-level virtualisation technology. Best starting point is probably the online documentation at https://docs.docker.com/.
[2] The list of available links can be found in the github repository, https://github.com/IOT-DSA/links.

However, the purpose of those links is to abstract away data sources and protocols, while biot focuses on the complete chain from the sources to the destinations.

On a broader scope, biot shares a number of ideas with systems and platforms for the realisation of web mashups, e.g. the former Yahoo! Pipes or the WSO2 Mashup Server. Biot differentiates itself through a specific focus on the Internet of Things and the (sometimes) domain-specific protocols that are associated. Also, while Pipes or WSO2 will sometimes rely on web scraping for getting data, biot lacks some of the flexibility necessary to such activities.

Developing for the Internet of Things requires a lot of plumbing code, probably because there are so many protocols and APIs at hand. The motivation behind biot is the realisation of a generic plumbing solution that would allow application designers and developers to focus on essential details such as the flow(s) of data or how to extract information from the various message formats involved, but less on low-level protocols or API details.

3. Requirements and Design

Biot tries to minimise network accesses as much as possible. This is because it also aims at being able to communicate directly with sensor networks (or bridges that are tightly connected to sensor networks), usually using modern protocols such as IPv6, CoAP or WebSockets. To this end, biot introduces the concept of variables, together with a number of generic methods to extract these from the information acquired at or sent by the source. All variables will be extracted at once from source data before being pushed further to relevant destination templates. In other words, even when polling is necessary, biot will try to maximise the information flow by creating and/or updating all the variables in one go.

Biot also tries to minimise network accesses on the way to the destinations. For example, variables that get extracted as a source is updated, are sent for treatment in one go in order to trigger as many destinations as possible. The state of all relevant variables is represented through an ephemeral snapshot to ensure data consistency on its way to destinations. Also the order of the destinations is relevant in case there were some inter-dependencies.
All templating code is run in safe interpreters [17] to guarantee a minimum security level. However, as biot mainly aims at being encapsulated into micro services such as docker components, security measures are otherwise kept to a minimum. For example, biot can use the output of external programs for data analysis without any sandboxing or restrictions over program execution. The advantages of supporting any external program, and thus any protocol that is not originally supported by biot, overcomes the security issues in the controlled context that should be surrounding biot deployments.

Apart from the concept of global variables (see below), biot integrates with modern cloud architectures through being able to acquire its settings from remote key-value stores. For example, an instance of biot will be able to get its list of sources from a remote, cluster-local, HTTP server or from a remote etcd [18] store. In addition, biot implements itself a REST-like [19] interface to access and modify its variables. Whenever variables are modified through the web interface, destinations will be triggered similarly to what happens when new source data has reached biot.

The code base is modular and as portable as possible, so as to ease integrating the features implemented as part of biot into other software. While biot is a command-line tool, ready for docker packaging, the core of its implementation is leveraged as a number of libraries that can be integrated in other software. All sources and destinations are implemented using a plugin system to ease future extensions. Plugins are selected based on the URI scheme of the source or destinations, and biot comes with plugins for HTTP, STOMP [20], syslog [21], piped external programs, local files, XMPP, raw line-based TCP connections and websockets.

Biot uses the + sign in the scheme of URLs as a separator to provide extra information as to the ways to get or send data from sources and to destinations. For example, specifying `http+put://` instead of the more common `http://` will force a PUT operation when the HTTP request is made. Biot uses the + sign as a separator and the keywords that it separates are given to all source and destination plugins for internal interpretation.

4. Concepts

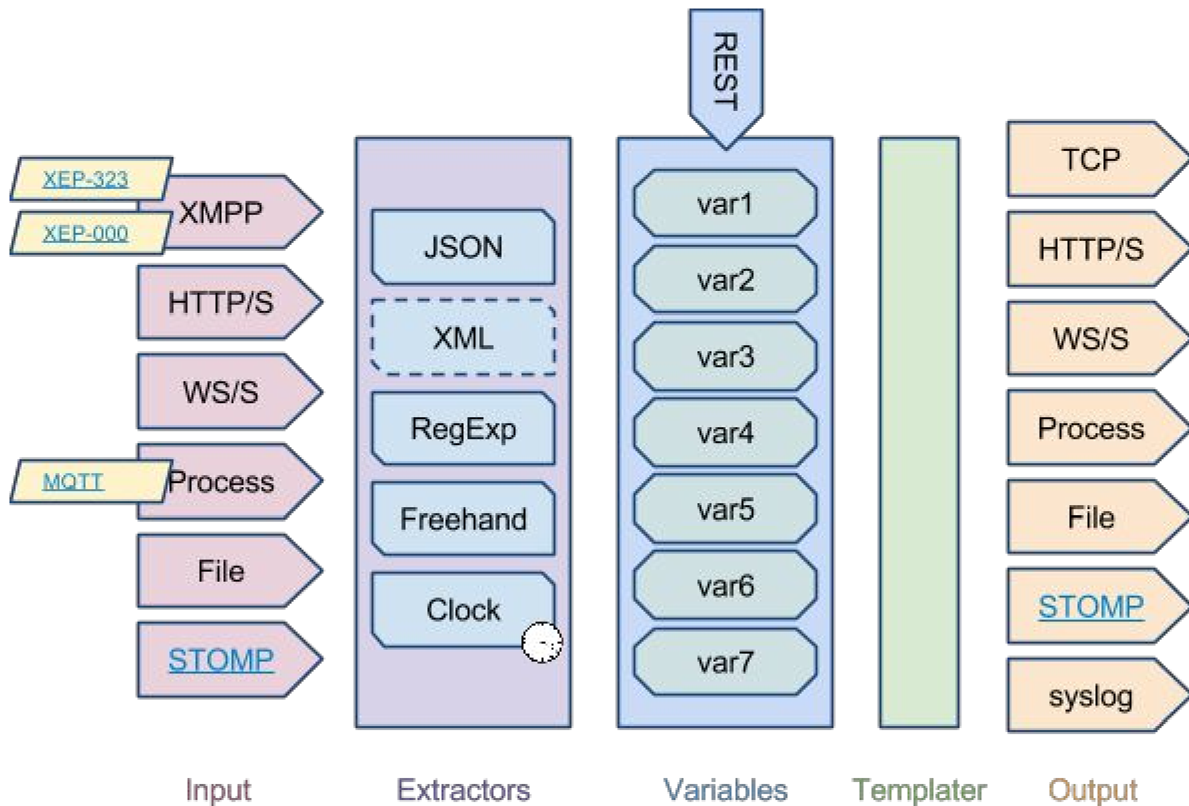
There are four major conceptual entities that are dealt with inside biot: sources, variables, destinations and global variables.

Sources are where information can be fetched or received from, such as a remote URL, a file or an external program. Sources are represented by URIs, with support for some extra sugaring rules on the schemes. Biot will associate natural semantics to sources by default: if a polling frequency is associated to a file, it will read the file block by block and consider this data as the core piece of information; however, if no frequency is specified, the file is supposed to be a log file (or similar) and the unit of growth will be lines, which will be the core unit of information to extract from. Biot uses the scheme to detect the type of the remote source, it has support for sources as various as HTTP (polling), WebSockets, XMPP, STOMP server, local files, external processes, etc. All relevant protocols can be secured further using TLS.

Variables represent the unit of information to extract from the sources. Biot supports querying received JSON structure to extract parts of the JSON tree, but more generic extraction types using regular expression or free-form Tcl templates can also be used. In their most advanced

form, variables can spontaneously be created when particular information is spotted. Biot also supports orphan variables, i.e. variables that are not bound to sources, but which values can depend on the value(s) from variable(s) extracted from source(s). Among those orphan variables, some are automatic variables and their values will always reflect the current date and time. Typical use is for timestamping information that would not contain time data or for pulsing to destinations at regular intervals. But the creation of variables that are not directly tighten to a source also promotes support for aggregating source variables or performing complex calculations on the sometimes simplistic data flows that originates from less capable sensors.

Destinations represent where to send the information, these are defined through a location (a URL, a file or an external program, for example) but also an output template. The content of the template will be filled using information about the sources, the variables, their value and which variables were changed. Again, biot uses the scheme to detect the type of the destination, and has support for a wide range of types, from local files and external processes, to remote HTTP, websocket or STOMP servers, including TLS secured destinations.



The figure above summarises the data flows from the various types of sources to the currently supported destinations as information travels within biot. Biot is also able to host a micro web

server, accepting REST operations to get and set the variables that have been declared or spontaneously generated.

Finally biot also supports *global variables*, in addition to also supporting OS-level environment variables. These global variables are typically used, together with environment variables, to propagate settings across the definition of sources and destinations. For example, they could contain a rootname used by several sources to find some information, or a common directory under which to place several destination files. Global variables fit with how docker components usually are parametrised and configured. While their naming is somewhat misleading, global variables are not to be confused with the other variables that are at the core of the information processing that occurs within biot.

5. Example and File Format

5.1. A Constructed Example

The following example would arrange to poll an external Internet service to detect your external IPv4 address. There is no destination in this example, as it is only meant to exhibit the domain-specific language that is used to tune biot's behaviour.

The following configuration snippet defines a source called `whatismyip` and associated to a polling frequency of `120` seconds. Note the, here unnecessary, `+get` that forces an HTTP GET operation and exemplifies the kind of sugaring that biot can use on URI schemes to refine source behaviour. For example, the whole REST vocabulary is made available for HTTP sources. The dashes are synonyms for empty strings, but these lines could be used to specify headers to the request, data to send as a POST query or basic data source filtering.

```
whatismyip
    http+get://whatismyip.org/
    -
    -
    -
    120
```

The following variable configuration snippet defines a variable called `ip`, bound to the source `whatismyaddr` defined above. Since the variable is marked with the type `RX`, biot arranges to update the variable with the content of the regular expression specified in the last line whenever the IP address changes.

```
whatismyip.ip:
    RX
```

```
((([2][5][0-5]|([2][0-4]|[1][0-9]|[0-9])?[0-9])\.){3})([2][5][0-5]|([
2][0-4]|[1][0-9]|[0-9])?[0-9])
```

5.2. File Format

The lists of sources, variables and descriptions can be described using a file format, but there are also procedural constructs for their creation. The formats are based on groups of consecutive lines. Blank lines and comments will be ignored, and so will all leading spaces. The file formats are described here more as a more detailed introduction to the features offered by biot.

5.2.1. Sources

Each source requires exactly 6 lines for its definition, and single dashes will be understood as empty lines. These 6 lines are, respectively:

1. The name of the source (only alphanumeric characters and dashes and underscores allowed).
2. The location of the source, biot will match (glob-style) the location to find a proper plugin for that type of source. There are currently plugins for `http`, `file`, `exec`, `stomp`, `xmpp` and `ws`.
3. A list of headers containing keys and their values. This is only meaningful for some of the source types, for example `HTTP` or `STOMP` sources, and can be used to specify API keys or similar.
4. An initial request when accessing the source location, e.g. request data when the source was specified as a `http+post`. When the first character of that line is a `<`, the initial request will be the result of a templating operation instead.
5. A list of transform specifications to apply on every chunk of data acquired from the source. The core of these transformations is a sed-like language, but there is also support for encoding conversions.
6. A polling period, expressed in fractions of seconds. Biot tries to find good default behaviour for most sources. For example, when `file:` is specified, if no polling is specified, data will be got from the source line by line as the file grows, but if polling is specified, the whole content will be sent for analysis each time.

5.2.2. Variables

Each variable requires exactly 3 lines for its definition, and single dashes will be understood as empty lines. These 3 lines are, respectively:

1. The specification of the variable. It is composed of the name of the source that the variable is bound to (possibly empty), followed by a dot `.`, followed by the name of the variable, possibly followed by a `:` and the initial value to set the variable to. Only alphanumeric characters (including dash and underscores) are allowed for the variable name. When the name starts with a `<`, this refers to a dynamic variable, meaning that the

matching and extraction operations (see below) will generate variables themselves. While a detailed description is out of the scope of this document, the facility enables the discovery of variables when the list of variables for a given source is not known in advance.

2. The type of the variable, composed of a keyword, possibly followed by a dot and an argument. The keyword is case insensitive and there are 4 types that are currently implemented: `RX` for regular expression extraction, `JSON` for json selection and extraction, `TPL` for generic templating and `XPR` for mathematical expressions (that will use the content of other variables).
3. An extraction specification, which can be conceptualised as the argument to the extraction and depends on the type. In the simplest form, this would be the regular expression to apply on the text if the variable was of the `RX` type, the path to the template for `TPL` variables, or the Xpath-like selection for `JSON` types. Complete details can be found in the biot reference.

5.2.3. Destinations

Each destination requires exactly 4 lines for its definition, and single dashes will be understood as empty lines. These 4 lines are, respectively:

1. The specification of the destination, which commences with a name for the destination. Only alphanumeric characters (including dash and underscores) are allowed. Following that name might be a colon followed by glob-style patterns that will filter which variable names are relevant for that destination, i.e. the variables which will have to be modified to trigger the destination.
2. The location of the destination. As for sources, plugins register for types using glob-style patterns on this location. There are currently plugins for `exec`, `file`, `http`, `stomp`, `syslog`, `tcp` and `ws`.
3. The template to generate the payload from, most of the time this will be the specification to a file. The template is run through a safe-Tcl interpreter where all the variables defined in the previous section can be accessed.
4. A list of headers, this list should be an even long list with, in order, keys and their values. This list will be mapped to headers whenever applicable, e.g. for `http` or `ws` destinations. A typical example would be the specification of an API key or similar when posting data to a remote cloud service.

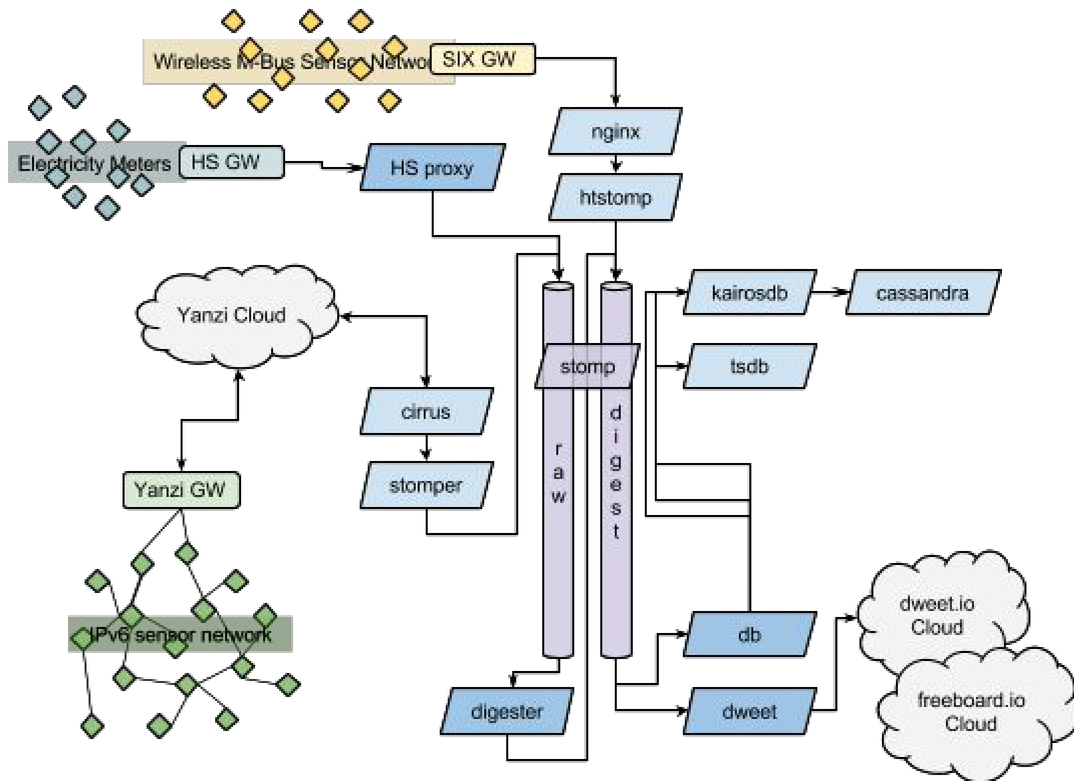
Note that destinations will only be triggered whenever one of their specified variables has changed and/or if the resulting payload has changed. Biot remembers payload data from one variable detection to the next automatically to avoid duplicate sending. Destinations are also triggered in the same order as their specification in the file.

6. Architectures using Biot

Biot has been or is being used in a number of data collection projects, most of them aiming at energy savings for villas and condominiums in Sweden. Central to those projects are a number of STOMP queues where data flows are represented using timestamped JSON objects on a main topic. For some sources, additional temporary queues containing flows in another format are used before being converted to the internal JSON format. A large number of biot-based micro-services are instantiated in order to enqueue data, convert from temporary queues or send data further for long time storage in time-series databases (KairosDB[3] and InfluxDB[4]), graphing (using Grafana[5]), or real-time presentation for end-users (freeboard[6] through dweet[7]).

The following figure shows the various possible connections between components within the clusters. Docker components are all represented by parallelograms, and biot-based components use a darker background. The exact instantiation of those components is abstracted away from the figure to ease its understanding. In practice, each “installation”, e.g. a block of flats, a villa, an office building, is mapped to as many components as required depending on the devices on premises. All installations share a common database (and common STOMP server[8] at present), a common web-based entry point, but all other components are replicated at will to create a replicated architecture with few point of failures. This architecture has proven to scale: there are for example 50 wireless M-bus temperature sensors pushing data at short intervals, we receive electricity data from 200+ flats and the biggest Yanzi-based installation uses a bit less than 2000 sensors.

[3] KairosDB is a time-series database built on top of Cassandra. For complete documentation over KairosDB and the protocols that it supports for data insertion and query, see: http://kairosdb.github.io/.
[4] InfluxDB is another time-series database with clustering capabilities. InfluxDB has no external dependencies to facilitate setup. More information is available at: https://influxdb.com/.
[5] Grafana is a full-featured metrics dashboard interfacing a number of time-series database, including KairosDB and InfluxDB. For more information, read at: https://influxdb.com/.
[6] freeboard.io enables the creation of real-time dashboards to highlight current (and historical) key values for things. Its API interfaces directly dweet.io.
[7] dweet.io implements an API (and short term storage) for IoT-oriented message passing.
[8] The `efrecon/stomp` docker component implements a simple STOMP server with no persistence or load-balancing. It is based on the Tcl STOMP library [22].



A variety of sources can be involved. A Tcl-based micro web server[9], protected by an NGINX reverse proxy receives data securely from devices that are HTTP(S) capable (heat pumps, wireless M-Bus sensors via gateway from six innovation[10], M-bus meter coupled to Elvaco[11] gateways, etc.). Electricity data from the condominiums is polled at regular intervals through a biot-component, using a proprietary API. Some heat pumps send their data using XMPP, while websockets connections are used to collect data from Yanzi[12]-powered IPv6 networks of sensors and actuators.

[9] The `efrecon/htsomp` docker component offers a flexible framework to implement a web-based entry point for external services and devices that are able to push data. Messages can be converted on reception and automatically pushed to STOMP queues. The code for the component is available as a Tcl project on github at https://github.com/efrecon/htstomp.

[10] Six Innovation is a small partner company, more information at http://www.six-innovation.com/.

[11] The Elvaco CMe Series are gateways capable of capturing MBus messages and sends these to remote Internet resources. See http://www.elvaco.com/en/products/cme-series for more information.

[12] Yanzi is a partner company selling a number of IPv6-based sensors and actuators.

The clusters are built and administrated using machinery[13], a Docker meta orchestration tool written in Tcl. Typically, the clusters will host a number of backend components on a number of pre-defined virtual machines for services such as the STOMP queues or the databases. A scalable number of worker virtual machines will host the components that are dedicated to on-premises installations. Components on these machines are spread out and orchestrated using Docker Swarm. A prototype cluster is built on top of virtual machines based on Oracle VirtualBox, while a production-oriented cluster runs on top of Microsoft Azure. For most of the data sources and site installations, but also internally and towards a large number of the external services, biot will mutate to adapt to the protocols in use or the data formats required by the various APIs. Each biot instance dynamically gets its configuration when it starts up from a cluster-internal etcd key-store.

In these architectures, all Tcl-based components have their roots in a minimal docker Tcl component[14], itself built on top of Alpine Linux. This enables quicker deployments, for example when new components need to be started to better balance application load. For example, the biot distribution contains tools to create a minimal Docker image that only is 23Mb.

7. Future Work

Biot is able to acquire its configuration not only from local files and the command-line, but also from remote locations such as web servers or an `etcd` keystore[15]. While this covers a number of scenarios, and especially the use of biot in CoreOS-based clusters, there are other competing technologies for the scalable storage of cluster-confined data and configuration. One widespread candidate is consul[16], which offers an API following similar principles to the one for etcd.

Biot still lacks supports for a number of widespread IoT protocols. While it is possible to support them using external command-line tools, implementing these would provide for more flexibility and less overhead. Possible candidates are proper support for the various XEPs, including the proposed publish/subscribe specifications, and for MQTT. Similarly, as message queues are essential to how modern web applications are created, supporting other protocols than STOMP

---

[13] machinery tries to be the missing piece at the top of the Docker pyramid. machinery is a command-line tool that integrates Machine, Swarm, Compose and Docker itself to manage the lifecycle of entire clusters. machinery combines a specifically crafted YAML file format with compose-compatible files to provide an at-a-glance view of whole clusters and all of their components. More information at https://github.com/efrecon/machinery.

[14] The `efrecon/mini-tcl` is a minimal batteries-included Tcl installation based on the standard Alpine Linux component. This enables to keep it size down while still providing a wide range of Tcl packages. It is available directly from the docker hub, but also from its github project at https://github.com/efrecon/mini-tcl.

[15] Support for etcd keystores is achieved through an implementation of the etcd v 2.0 API in Tcl. This implementation is available separately at https://github.com/efrecon/etcd-tcl.

[16] Consul is another fault-tolerant key-value store, more information at https://www.consul.io/.

should also be considered. For example, beanstalkd offers a clear-text simple protocol for the scheduling and consumption of jobs.

There are a wide number of external services that offer an HTTP-based API that is compatible with biot, and biot itself has been tested against a number of these services. Out of all the existing services, mashape stands out through providing an API that aims at integrating other APIs. Testing support for mashape would open up biot to an increasing number of web-based services.

Configuring larger biot projects can lead to the creation of many files, since biot usually offers an increased flexibility when using templates. The provision of a graphical editor, perhaps based on dataflows in a manner similar to what NodeRED offers, would be of great benefit and would allow developers to better benefit from the minimal rule-based system that it implements.

8. Conclusion

Biot is a flexible tool to shuffle and transform data from any source to any destination in modern cloud architectures. Aside the Internet of Things, biot can also be used in other contexts where real-time data mashup pipelines are necessary. Biot benefits from a number of Tcl-specific features, such as its fast regular expression implementation or its flexibility when mixing code and data in output templates. Tcl is sometimes considered as a glue language between heterogeneous software components. Biot shows that Tcl's glueing capacity also applies to the cloud and that it can be used to participate to the building of the complex architectures behind modern web applications.

9. References

[1] "*ZigBee 3.0: The Foundation for the Internet of Things*", available at http://www.zigbee.org/zigbee-for-developers/zigbee3-0/.

[2] "*G.9959 : Short range narrow-band digital radiocommunication transceivers - PHY, MAC, SAR and LLC layer specifications*", ITU-T Recommendation, 2015.

[3] "*Bluetooth Smart Technology: Powering the Internet of Things*", available at http://www.bluetooth.com/Pages/Bluetooth-Smart.aspx.

[4] "*Transmission of IPv6 Packets over IEEE 802.15.4 Networks*", G. Montenegro, N. Kushalnagar, J. Hui, D. Culler, IEEE RFC 4944, 2007.

[5] "*Hypertext Transfer Protocol -- HTTP/1.1*", R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, IEEE RFC 2616, 1999.

[6] "*The Constrained Application Protocol (CoAP)*", Z. Shelby, K. Hartke, C. Bormann, IEEE RFC 7252, 2014.

[7] "*The WebSocket Protocol*", I. Fette, A. Melnikov, IEEE RFC 6455, 2011.

[8] "*MQTT Version 3.1.1*", A. Banks, R. Gupta (eds.), OASIS Standard, 2014, available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[9] "*XEP-0323: Internet of Things - Sensor Data*", P. Waher, Experimental XMPP Extension Protocol, 2015, available at: http://xmpp.org/extensions/xep-0323.html.

[10] "*OMA Lightweight M2M*", OMA Specification, 2015, available at: http://openmobilealliance.hs-sites.com/lightweight-m2m-specification-from-oma.

[11] "*Web of Things*", Wikipedia, available at: https://en.wikipedia.org/wiki/Web_of_Things.

[12] "*AllJoyn Framework Architecture*", AllSeen Alliance, available at: https://allseenalliance.org/developers/learn/architecture.

[13] "*The JSON Data Interchange Format*", ECMA standard 404, 2013.

[14] "*Extensible Markup Language (XML) 1.0 (Fifth Edition)*", T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, W3C Recommendation, 2008.

[15] "*Node-RED Documentation*", available at: http://nodered.org/docs/

[16] "IOT-DSA", available at http://iot-dsa.org/how-it-works.

[17] "*The Safe-Tcl Security Model*", J. Levy, L. Demailly, J. Ousterhout, B. Welch, Proceedings of the USENIX Annual Technical Conference (NO 98), New Orleans, Louisiana, 1998.

[18] "*Documentation - etcd*", available at: https://coreos.com/docs/etcd/.

[19] "*Architectural Styles and the Design of Network-based Software Architectures*", R. Fielding. Doctoral dissertation, University of California, Irvine, 2000.

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

[21] "*The syslog protocol*", R. Gerhards, IEEE RFC 5424, 2009.

[22] "*Web(-like) Protocols for the Internet of Things"*, E. Frécon, Proceedings of the 20'th Tcl/Tk Conference, 2013.