Bridging MQTT & XMPP Internet of Things networks

Peter Waher
Clayster Laboratorios Chile S.A, Blanco 1623, of. 1402, Valparaíso, Chile
peter.waher@clayster.com

Abstract. This paper starts by providing a general overview of Internet of Things architecture patterns, and then comparing the similarities and differences between MQTT and XMPP Internet of Things network patterns in particular. Finally, a mechanism is provided whereby networks of these two types can be bridged.

Keywords: Internet of Things, Peer-to-peer networks, XMPP, MQTT.

1 Introduction

One of the first problems any Internet of Things application needs to address as they move into private spheres is how to communicate with things behind firewalls. For classical internet applications, the client – normally a browser behind a firewall – connects to a server on the Internet. The server in turn contains all content or knows where to find it. But, as soon as you wish to include Things into the equation, you end up with the problem: How does the server you’re connected to (via the browser) reach the Thing that is located beside you behind the firewall, so that it can present its current status to the user in the browser?

Even though Internet of Things is much larger than the above mentioned example, it serves as a good example of the mentioned problem. Before delving deeper into the particulars of MQTT and XMPP networks, a short introduction to the different solutions to this problem is presented. This will give some background to our comparison of the two technologies. Following is a brief list of solution patterns:

1. Client-side APIs
2. Cloud storage of data
3. Publish/Subscribe architecture patterns
4. Peer-to-Peer communication
5. Hybrid approaches

1.1 Client-side APIs

There are various efforts, like the W3C Device API Working Group [1], that develop client-side APIs that allow applications running in the client to communicate with devices in the same network. The Node.js project [2] provides a JavaScript API
for Chrome’s JavaScript runtime that allows for different types of I/O operations not normally supported in browsers. For instance UPnP [3], a popular protocol for communication with devices and peripherals in the home or office, is supported through the Node.js project.

The client-side API solution might work well for the limited case listed above. However, if the client wants to reach devices in another network, this approach does not work. Also, this approach is limited to what type of client you are using when viewing the application, or to the fact that a client must be used at all. What happens if you want the service to communicate with the device, when your client (i.e. browser) is turned off?

Even though such client-side APIs provide an interesting opportunity in accessing devices close to the client, normal IoT applications require other solutions to be able to communicate with things behind firewalls.

### 1.2 Cloud storage of data

Another popular method is for the Things to regularly publish information about themselves to the server (i.e. to “the Cloud”) through some kind of proprietary API, often HTTP(S)-based. The server can then choose to present the latest information to the client, when requested.

To allow other Things to access the data, the server API is often extended to include methods to extract either the latest reported value or historical data.

Common APIs can be RESTful web services [4] returning XML, using either proprietary formats or standardized formats such as RSS [5] or ATOM [6]. JSON is also popular, since it allows for easy implementation in script languages.

Examples of platforms that use this technique for Internet of Things applications include Xively [7], Open.sen.se [8] and Sics"Sense [9] etc.

One of the major problems with this architecture pattern is that there is latency, first in the interval of which the Thing reports its data to the server, and then, in the interval in which the client polls the server to see if there are any new values available. The shorter the interval is, the higher the load on the server will be. This problem is largely a consequence of the HTTP protocol, that only allows for request/response type of communication and not full-duplex communication where messages can be sent in both directions. Even though there are techniques for minimizing these limitations, they will still only be fixes, and the original problem persists.

Another major problem is that publishers of the data have no control of who can see what data. As soon as the data has left the device, it is in essence owned by the server. This also adds the risk of exposing private data to unknown corporate or government interests as the data is stored centrally. This might minimize the desire of end-users to use the approach when it comes to private information.

### 1.3 Publish/Subscribe architecture pattern

The publish/subscribe architecture pattern [10] is an abstraction of the above solution from a communication perspective, where the problem has been reduced, gener-
alized and simplified. Also, unnecessary restrictions imposed on the architecture by the HTTP protocol have been removed.

The publish/subscribe architecture pattern basically consists of three types of actors: Publishers, message brokers and subscribers. Publishers generate content and publish it to a message broker. The message broker immediately distributes the content, or information about the content, to subscribers having subscribed to the particular content.

Using this pattern, publishers and subscribers connect to the message broker and can therefore reside behind firewalls. The message broker, however, needs to be reachable by all actors in the network.

Using this pattern, publishers and subscribers connect to the message broker and can therefore reside behind firewalls. The message broker, however, needs to be reachable by all actors in the network.

![Fig. 1. Publish/Subscribe pattern](image)

As described above, all publishers and subscribers connect to the broker, bypassing any firewall restrictions on traffic in the opposite direction. Subscribers are also required to maintain the connection with the broker in order to be able to receive the corresponding information. In Fig. 1, the flow of information is from left to right. However, publishers can also be subscribers.

The publish/subscribe architecture is very efficient in distributing messages in large networks. However, it only supports one type of communication pattern: Things publishing information, consumed by others. It is difficult to create an environment permitting two-way communication and impossible for the publishers to control who gets access to what.

There are various protocols that readily provide publish/subscribe architecture support, like XMPP [11] with the publish/subscribe extension [12]. There are also protocols or platforms that are designed primarily with publish/subscribe in mind, like MQTT [13]. Web platforms, such as Twitter [14], also work according to the publish/subscribe pattern, even though they largely use HTTP as a transport for data.
1.4 Peer-to-peer communication

The field of Peer-to-peer communication techniques is another interesting field of study for Internet of Things applications. It allows for Things to communicate directly with each other without having to go through middleware or servers in the process. However, many such peer-to-peer techniques open up the network and create vulnerabilities that are not desirable. The paper “Extending the Semantic Web to Peer-to-Peer-Like Sensor Networks based on XMPP” [15] discusses different peer-to-peer communication techniques with their pros and cons.

1.5 Hybrid approaches

As described in [15], there are hybrid approaches that emulate the flexibility of peer-to-peer networks allowing point-to-point communication but use message brokers that authenticate all Things and restrict to whom they can communicate, this to avoid opening up the network and create security vulnerabilities. This assures that communication from unknown or undesired parties is avoided. XMPP fulfills all these requirements [16].

2 MQTT – Message Queuing Telemetry Transport

MQTT is a designed to be a very simple and minimalistic publish/subscribe protocol that can be implemented in small resource constrained devices. In [13] it is stated: “MQTT … was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. ... It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers”.

The design principles in the development of the MQTT protocol are:

1. code footprint
2. network bandwidth
3. some assurance of delivery.

The following sections will discuss how well MQTT solves the above priorities. It will list its pros and in detail also list the weaknesses of the MQTT approach.

2.1 MQTT – An overview

The MQTT protocol specification v3.1 [17] outlines how the protocol works: Data, which is always binary and without a content type specification, is published on topics. Topics and subtopics are strings, sorted into a tree structure, with any number of root nodes. The full topic name is computed by taking the path of the topics and subtopics, concatenating them all, delimiting them using a slash character (/).
Data is then published using one of three quality-of-service levels: “Fire and Forget” (QoS level 0) means the server will forward the data, if possible. It will not apply any retries or acknowledgements to make sure the data is delivered to all interested parties. The second level (QoS level 1) is called “Acknowledged”. All messages are resent in case an acknowledgement is not received within a particular time period. This makes sure the data is delivered. But it can be delivered multiple times. The third level (QoS level 2) is called “Assured”. Here, a double acknowledge process is employed, not only to make sure the message is delivered to all interested parties, but also to make sure that it is delivered only once to each party. When publishing data, the sender can ask the server to retain the latest information on the server. This makes the data directly available at a later time if anybody show interest in a topic after the data was published.

To retrieve data from an MQTT server, you subscribe to topics. If you don’t subscribe to anything, which is the default state, you don’t receive any data. You can subscribe to any number of topics, and you can use path wildcards to subscribe to sets of topics, or possibly all topics available on the server.

All messages are encapsulated into binary packets that add very little overhead to data being sent. This is not necessarily the same as minimizing network bandwidth, as it depends on what data is being sent and the topic structure used, as we shall see later.

2.2 MQTT – Intrinsic weaknesses

There are several intrinsic weaknesses in the MQTT protocol. The weaknesses are discussed in the following subsections, one at a time.

User Authentication

One of the major security issues with MQTT is its user login feature. To add a layer of security you can pass a user name and password to the server. This however, is done in clear text, which obviously is a very bad idea on the Internet. Since the original form of the protocol is unencrypted, it is very easy to extract both user names and passwords from devices connected to MQTT.

To solve this weakness, they have afterwards added a layer of encryption outside of the protocol itself. This means, that the protocol can be used over an SSL/TLS encrypted connection through another port, thus hiding the password from eavesdroppers. The problem with this solution, as they recognize in the protocol description, is the following: “SSL is not the lightest of protocols, and does add significant network overhead”. Not only does it incur extra processing power during the entire communication session, particularly in cases where it’s only during authentication it is required, it also forces resource constrained devices to dedicate a not insignificant portion of its memory to an SSL/TLS library, a library that by some measurements take several times more space than the communication library itself. So, by choosing such a solution to the problem, instead of redesigning the protocol to use a lightweight digest-based authentication method [18], salted challenge response authentication
mechanism [19], or better, SASL [20], they enforce supposedly resource constrained devices to use SSL/TLS when communicating with the server, even when it is not necessary.

Furthermore, using SSL/TLS does not entirely remove the problem of sending the password in clear text. It only removes the possibility to eavesdrop. Of all devices connected to a server, how many correctly validates the server certificate? Sufficient that one device does not do so, and it will be possible to extract the username and password of the device using a Man-in-the-middle attack [21]. And once given access to the server, a malicious user can publish information to any topic manipulating the entire network. The risk obviously becomes greater the more devices connect. To avoid this, all clients, without exception, must also use valid client certificates that the server validates, to avoid the use of passwords. But how simple is it to implement such a solution that requires manufacturers to produce valid client certificates for each device, update them when they become invalid, and connect certificates with user accounts on the server? Self-signed certificates will obviously not do in an open environment.

To prevent similar attacks, even more burden is placed on the already resource constrained device. Now it needs to not only communicate over SSL/TLS, it needs to securely validate the server-side certificate, have means to update invalid client side certificates, take into consideration certificate revocation lists, certificate renewals, domain validation, etc. Furthermore, the IT department of each MQTT installation needs to regularly update server certificates, not to mention update client side certificates in devices, etc. All this is necessary because passwords are sent in clear text.

To avoid some of the complexities using certificate validation, it can be noted that some MQTT server providers allow for a third type of connection which also uses SSL/TLS and also requires a client certificate to be provided for each device, but it uses pre-shared keys (PSK) together with TLS [22]. This can be used as a somewhat lighter version to avoid the public key operations phase and limit the load on the constrained device during authentication. But TLS+PSK still requires manual or out-of-band configuration of both clients and brokers at the same time, making it suitable only for very high value connections that have small user populations.

To summarize: To minimize code footprint in the smallest version, they have actually created a protocol that requires more code size to be secure, invalidating the original premises of the protocol. It would have been better if the protocol development team would have incorporated a digest-based user authentication scheme, salted challenge response authentication mechanism, or the more flexible and generally accepted SASL authentication scheme, both in terms of security, code size and network bandwidth. These mechanisms would avoid sending the password in clear text and all the problems that are a consequence of this, and it gives a reasonable protection against eavesdropping, even though the data itself is not necessarily encrypted.

It can be noted, that an encrypted connection in MQTT does not protect the privacy of the data itself, which makes the effort implementing it a bit wasted. Since data can be retrieved anyway by anybody, simply by subscribing to the topics concerned, encrypting of contents through encrypting the connection does not solve the privacy issue. An end-to-end encryption scheme is required. To achieve this in MQTT, en-
cryption of content has to be done on the content itself by the device to assure privacy of data. So, in this case, the same data has to be encrypted twice. Furthermore, to encrypt the entire connection during its entire life time just to hide the password in the beginning is a terrible waste of resources, especially if the content has to be encrypted a second time as well.

**Topic ownership, content types & reliable messaging**

MQTT is considered a protocol for “reliable messaging” [23]. This label is somewhat unfortunate as will be demonstrated below.

The MQTT protocol, as mentioned above, is designed to allow for some “assurance of delivery”, as any Message Queuing protocol. And it does support “reliable delivery”, as long as clients request it and the server accepts it. And this is also normally what is meant by “reliable messaging”, at least in the realm of point-to-point web services [24].

However, in the MQTT case, you have a multipoint to multipoint communication infrastructure that does not follow the same logic as point-to-point web service infrastructures.

For instance, anybody subscribing to a given topic also receives data published on that topic. The sender cannot control who receives the data. Also, anybody can publish data on any topic. There is no creator and owner of a topic. And so the receiver cannot control from whom the data comes. This makes it not only possible to confuse topics, thus confusing (and possibly destroying) communication by mistakenly using the same topic names, it also makes it impossible to validate if data comes from the correct source without implementing specific manufacturer specific code into each telegram making interoperability a problem. It is also possible to inject bad or destructive data into the stream without parties taking notice.

As an example, consider a simple thermostat application. A temperature sensor publishes temperature information. The controller receives these temperatures and sends control messages to a heater and a fan to control temperature. However, it’s easy to hijack such a system. First, a device could subscribe to the same temperature value. It would receive the temperature reported by the temperature sensor, and then immediately send another temperature value to the same topic, for instance lowering the value one degree. The controller would react to the new value and leave the heater on for an extra degree, increasing the ambient temperature with one degree. To solve such vulnerabilities each sender has to subscribe to itself and analyze that nobody else writes to the same topic, or each controller receiving sensor data has to analyze incoming data and sound an alarm if data does not behave as it should. A third method would be to encode data with the sender’s signature, so that recipients could validate that the data comes from the correct sensor. However, such measures would have to take into account the possibility of servicing or replacing the sensor in the future. All these methods are complex for simple devices to perform.

There are server-specific features, not part of the MQTT-protocol, that would allow an operator to control who can subscribe to what topics and who can publish to

---

1 There is no return code returned from the server when publishing information.
the same topics [25]. However, this must be done on a message broker operator level and cannot be controlled by the devices themselves or the device operator, unless they are the same. This limits the possibility to interoperate on the Internet, as not all parties can have access to the message broker, if using the same, especially if they are competitors.

Furthermore, as MQTT does not provide a means of transmitting content type together with the content, resource constrained devices will probably have limited possibility to validate incoming data. It can probably be assumed that these devices will themselves assume incoming data to have certain properties, such as format, data size, etc. Since injecting data to any device is simple, special care has to be taken to check all incoming data for size (to avoid buffer overflows), data types (simple data types, XML, JSON, etc.), injected code (for instance if JSON is opened in JavaScript directly), XSL and DTD statements in XML, etc.

All these aspects could be considered the opposite of what otherwise be considered “reliable messaging”.

2.3 Content Types & Implementation patterns

There are multiple ways to publish IoT information on MQTT. Choice of format greatly affects performance and what can be done with the data. There are basically two different approaches.

The first approach is to use different topics for different data fields being reported by a device. This would allow different recipients to subscribe to different types of data. It could also be used theoretically to limit access to different types of data, if the message broker supported such features. The downside of this approach is that communication gets bloated, since base topic names has to be repeated for each field name, thus contradicting one of the primary reasons for the developing the protocol: Limiting band-width required.

The second approach is to use a compound format and store all data from the device under a single topic. This would decrease the amount of data required to transport the information, but it would remove any possibilities to limit access to subsets of the data.

It is difficult to obtain any information on what formats are used by the MQTT community in their implementations. However, to get a feel for how people use MQTT a test application was created that would connect to different MQTT message brokers and subscribe to all topics and then analyze what kind of data was stored on different topics. It was tested on a pair of public message brokers [26] and a private message broker available through a research institute. Table 1 lists the distribution of different encoding schemes found during first one minute and then during two hours. The first number would represent mostly topics with retained information, and the second number retained and un-retained topics. The numbers represent topics encoding data in the corresponding format, not the number of data items received.
The content types listed in the table are divided between simple content types and compound content types. Topics without any content are classified as empty. The content types are classified in the following manner:

If the content starts with a < character and ends with a > character, the content is attempted to be loaded into an XML DOM object. If successful, the content will be classified as XML, Soap Envelope or SenML depending on the root element of the XML.

A non-empty block of binary data is considered to be a string if, using the UTF-8 encoding [27], the transformation $\text{StrToBin(BinToStr(Data))}$ does not change the data, and that the string $\text{BinToStr(Data)}$ does not contain any control characters (character code below 32) except 8, 9, 10, 12 and 13. It will also be classified as string content in the table above if it is not possible to classify it as any other of the simple content types, or if it is not XML (including a SOAP Envelope) or JSON.

A string that contains a floating point number using either a period or a comma as decimal separator will be classified as a number above.

A string that contains any of the following values (case insensitive), will be considered a boolean value: 1, true, yes, on, ok, 0, false, no, off, cancel.

A string containing a value that conforms to an XML date & time value (xs:dateTime) or a Web Date Time according to RFC 822, will be considered a Date & Time value.

### Table 1. Encoding distribution - 1 minute vs. 2 hours

<table>
<thead>
<tr>
<th>Type</th>
<th>Private broker</th>
<th>m2m.eclipse.org</th>
<th>test.mosquitto.org</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 min</td>
<td>2 h</td>
<td>1 min</td>
</tr>
<tr>
<td>String</td>
<td>4</td>
<td>28</td>
<td>153</td>
</tr>
<tr>
<td>Number</td>
<td>3</td>
<td>3</td>
<td>128</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Boolean</td>
<td>0</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>Date &amp; Time</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Guid</td>
<td>0</td>
<td>0</td>
<td>2859</td>
</tr>
<tr>
<td>Physical Magnitude</td>
<td>12</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>Empty</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Xml</td>
<td>133</td>
<td>133</td>
<td>1</td>
</tr>
<tr>
<td>Soap Envelope</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>SenML</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Json</td>
<td>199</td>
<td>237</td>
<td>17</td>
</tr>
<tr>
<td>Image</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Binary</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
A string encoded as a GUID [28] will be classified as a **guid** in the table above.

A string containing at most 16 hexadecimal characters (64-bit hexadecimal integer) will be classified as **hexadecimal** in the table above.

A string containing a floating point number followed by optional white space and then a unit will be classified as a **physical magnitude**.

If a string has not been classified as a simple type or XML, a JSON decoding is attempted. If successful, the content is classified as **JSON**.

If content is a string, but not classified as another content type as above, it is classified as a **string** in the table above.

If content is not a string, an attempt to decode the content as an image is performed (supporting JPG, PNG, BMP and TIFF formats). If successful, the content will be classified as **image**.

Otherwise the content will be classified as **binary**.

One interesting thing to note here is that there are very few topics that report binary data: 0% on the private broker, 0.23% on m2m.eclipse.org and 0.18% on test.mosquitto.org. This means that package size and binary encoding is not the primary concern of developers choosing MQTT as a protocol, at least among those that experiment with the public servers.

Another interesting thing to note is the distribution between simple topics, and compound topics[2]. On the private broker only 18.6% of the topics contained simple content. Here, a conscious choice has been made to use XML and JSON to encode data. But on m2m.eclipse.org this number of simple content types is 99.3%. Even if we exclude the GUID topics, this number is 89.9%. At test.mosquitto.org this number is 98.1%. So the predominant pattern on the public servers is to publish values as simple content types in separate topics, and not to use compound types. This reinforces the conclusion above, that the primary concern of developers choosing MQTT as a protocol is not package size, but ease of implementation. Concerns about package size and security do not affect implementation, at least when it comes to the public message brokers.

---

[2] The following numbers might be somewhat misleading, even though the conclusion remains intact. Because how we classify simple types, some compound content types might be classified as strings. A manual revision of values classified as strings in the example provided, shows that some strings actually are compound data items, either several delimited data items including partial XML or JSON. However, the amount of such topics do not change the conclusion that the predominant pattern on the public servers is to store data as simple types in a topic tree rather than using compound data.
2.4 Publish/subscribe pattern

MQTT is heavily focused on a publish/subscribe communication pattern. This is an efficient pattern if the following conditions are met:

1. If there are multiple subscribers to each published data item. In this case, the message is published only once, and the broker distributes the information to each subscriber.

2. If there’s no need for on-demand data, i.e. it is sufficient to receive periodical-ly updated values, or values updated on change and there’s no need for more detailed information on-demand.

3. Most of the published items are actually used, and not only stored for possible later use.

If, on the other hand, any of the following is true, the publish/subscribe pattern might not be a suitable pattern to use:

1. If there are few subscribers to the data being published. If the general communication flow is from a publisher to a receiver, a point-to-point communication pattern would be more relevant.

2. If there’s a need for on-demand data. If on-demand data is required, a request/response communication pattern would be more suitable. This can be solved by creating command topics where subscribers can publish requests for new data. Such commands are inefficient message-wise, as well as manufacturer specific, which further reduces interoperability.

3. If relatively few data points are actually used. Consider a service where the client can view the temperature somewhere by logging into the service. Perhaps this service is used only once a day, week or month. But to be able to provide any relatively actual temperature readings, values have to be published multiple times per hour. It’s an inefficient way to achieve the goal, unless there’s a logging of temperature values and presentation of temperature history as well. If only current status is desired, a request/response communication pattern would be more efficient.

2.5 Summary

To summarize our conclusions regarding MQTT we see that the primary reason for using MQTT seems to be ease of implementation. And implementing MQTT is very easy. It is very easy to start communicating data. However, this simplicity comes at a price: Very low security and interoperability. Each developer needs to address the security issues separately, which decreases interoperability further, and adds complex-
ity to the implementation contradicting the main premise. As ease of implementation is the primary factor to start with, it is our guess that this step is omitted even in the long run. At least from looking at the public servers, this step has been largely omitted. When analyzing solutions using MQTT, special attention has to be given to these issues, to make sure large backdoors are not implemented that can be taken advantage of in the future.

From the data analyzed, network bandwidth does not seem to be an issue either, as this would have spawned other types of solutions. Simple content topics should have been avoided, and compound types should have been used. But instead of using XML or JSON, we would have seen EXI-compressed XML or similar methods used, or purely binary packets, to decrease packet sizes.

Our conclusion is that MQTT is an efficient means to distributing information, but because of its intrinsic weaknesses it should only be used where the manufacturer controls all parties involved in the communication flow: Publishers, subscribers, message brokers and IT infrastructure. It should not be used in a more open infrastructure where multiple parties are involved to communicate interoperable and secured information.

3 XMPP – eXtensible Messaging and Presence Protocol

XMPP is a protocol that allows for direct instant messaging between peers by using a message broker or possibly a federated set of message brokers. By using message brokers, the problem with firewalls is solved. Devices behind firewalls create sockets to the message brokers lying outside of the firewalls. The message brokers assure that clients are authenticated and that approved friendships exist between clients. Messages can thus only be sent between clients if the message broker authorizes them.

The major aspects of the XMPP protocol, stated in [29], are:

1. **Open**: The protocol is free, open public and easy to understand.

2. **Standard**: The core XML streaming protocols has been formalized by IETF [30] [31] [32], and extensions are continuously being developed by the XMPP Standards Foundation.

3. **Proven**: The XMPP protocol has been in use since 1998, with multitudes of implementations, client libraries, server installations, etc.

4. **Decentralized**: There are no centralized components. Anybody can host an XMPP Server, and they can be freely federated.

5. **Secure**: It has robust security built into the core using SASL and TLS. XMPP networks are virtually spam-free. Furthermore, work is being performed to standardize end-to-end encryption solutions on-top of XMPP to raise the security bar even higher [33].

6. **Extensible**: XMPP has a built-in architecture for extending the protocol without risk of confusion. Extensions can be maintained private or published openly as XMPP Extensions (XEPs) [34]. This is the basis for interoperability.
7. **Flexible**: The XMPP protocol allows for many different types of communication patterns and usage. Not only point-to-point communication, asynchronous messaging, request/response patterns, but also content syndication using the publish/subscribe pattern, or multi-cast patterns etc.

8. **Diverse**: There are many different companies and projects working in distinct fields using XMPP not only for Instant Messaging or Internet of Things, but also for network management, syndication, collaboration, gaming, monitoring, middleware, file sharing, cloud computing, etc. XMPP will not lock you in, but allow you to use it in the way that best suits your needs.

The following sections will discuss how well XMPP solves the above priorities. It will also list its pros and in detail also list the weaknesses of the approach.

3.1 **XMPP – An overview**

An XMPP client connects to the network by using one of a series of available bindings. The normal binding, as prescribed by the XMPP core documents [35], is a normal socket connection to the server. On this connection XML fragments are sent. The connection can be both encrypted and/or compressed. User authentication is performed using SASL [20], which means a pluggable user authentication mechanism is supported. Efforts are being made to upgrade the network to require encryption [36].

Other bindings available may include BOSH (“Bidirectional-streams Over Synchronous HTTP”) [37], web sockets [38] and EXI-compressed binary streams [39].

Once a connection to the broker has been made, and the user has been authenticated, **stanzas** can be sent and received over the connection. The main types of stanzas include **message** stanzas, **iq** stanzas and **presence** stanzas. Message stanzas allow for asynchronous messaging between peers or between clients and servers (message brokers). Iq-stanzas, on the other hand, allow for request/response between peers or between a client and a server or vice versa. Presence stanzas are used to control visibility or availability on the network, as well as controlling friendships between clients.

Stanzas sent can also contain payload. Payload can be features defined by the core specifications, but can also include others type of valid XML fragments. The meaning of the content is defined solely by the qualified name of the content (i.e. namespace + local element name). This is how XMPP achieves the extensibility feature intrinsic in the protocol. It is this simplicity to extend the original XMPP protocol without risking destroying or affecting existing implementations, that has made it so flexible and usable.

3.2 **XMPP – Intrinsic weaknesses**

There are a couple of intrinsic weaknesses built into XMPP that users should be aware of. The following sections describe them one at a time.

**Transfer of In-band binary data is limited**
Since XMPP is sending and receiving XML fragments, any binary data that has to be transmitted, has to be encoded to fit into an XML message before it is sent, for instance by using base-64 encoding [40]. Even if this effect is reduced if EXI compression is used [39], you still have a limit to how large each stanza in the stream can be. According to RFC 6120 [30] there is a smallest allowed maximum stanza size that all XMPP servers must support. According to §13.12.4 of that document, this limit is set to 10000 bytes including all characters from the opening < character to the closing > character. This is the smallest allowed maximum stanza size. Support for stanza sizes above that limit is server specific. So, large information blocks either require you to break them up in pieces and send them in separate stanzas, or you send them out-of-band, for instance using Jingle [41].

**Bandwidth throttling**

Some XMPP servers may include a feature called Bandwidth throttling. The idea comes from chat applications and that humans normally do not write very quickly. It basically states that humans in IM chat applications seldom need to send more data between themselves than a couple of thousand bytes per second. In large systems, using bandwidth throttling may be a way to perform load balancing.

However, in machine-to-machine applications, such bandwidth throttling can cause problems, especially when machines need to send large chunks of data quickly.

This feature can normally be configured to be disabled or set to an acceptable level, as long as you are in control of the XMPP server. However, it might place restrictions on what public message brokers you can use, if sending large bursts of data in-band in real-time is an issue.

**Code and Message Sizes**

It is common to criticize XMPP for putting a heavy load both on code footprint and packet size compared to other solutions. And even though it is true that it demands a little more than some solutions, it is not necessarily much as is demonstrated by Ronny Klauck and Michael Kirsche in their work related to Chatty Things [42] [43]. Packet size can also be reduced considerably using EXI-compressed communication [39]. It has to be remembered however, that this size is not ballast; it provides very important features such as security, interoperability, extensibility, etc.

### 3.3 XMPP Extensions

There are many public extensions available, published on the xmpp.org web page [34]. The point in creating multiple extensions is to form a large toolbox of tools. The implementer can choose to implement just those extensions that are required by their implementation. I will not list all extensions here, merely those that directly relate to an Internet-of-Things perspective, since the purpose is to compare two protocols for the Internet-of-Things:
XEP-0045: Defines a multi-cast communication pattern in the form of Multi-User Chat [44]. It works similarly to multi-casting in IP-networks using the IGMP protocol [45], except that you subscribe to messages in a room instead of a multicast address.

XEP-0060: Defines a publish/subscribe communication pattern for use with XMPP. In cases where a publish/subscribe pattern is desired and/or more efficient, this extension defines how such communication can be implemented. To use this feature, you need an XMPP Server that supports the publish/subscribe extension [46].

XEP-0174: Provides a mechanism whereby clients can communicate between themselves without the use of a message broker [47].

XEP-0322: Defines how to compress XML used in the XMPP session using Efficient XML interchange, reducing packet sizes [39].

XEP-0323: Provides a request/response mechanism for reading sensors and encoding sensor or Internet of Things data. The format includes the possible use of meta-information important in IoT applications, and the aim is to encode the data in a way so that it can be understood both by humans and machines [48].

XEP-0324: Provisioning of things, services and user privileges in Internet-of-Things networks [49]. This extension adds an additional security layer on top of XMPP, helping Things, normally with very limited user interfaces, to achieve high security by controlled delegation of trust. The extension defines mechanisms to control who can talk to whom, read what data, control what parameters, with what services and by which users. It also has a generic user privilege system that can be used by distributed services in Internet-of-Things applications.

XEP-0325: Defines a request/response mechanism for control of devices and actuators [50]. It defines a mechanism to publish controllable parameters including meta information to automatic integration in a way that is understandable both by humans and by machines.

XEP-0326: Creates an abstraction where XEP-0323, 0324 and 0325 can be used to access multiple devices behind a single JID. In this way, the XMPP client becomes a concentrator, allowing access to everything from small PLCs to large subsystems of devices behind it [51]. This extension can also be used to bridge or mirror other networks based on other protocols into XMPP networks.

XEP-0332: Allows transportation of HTTP requests and responses over XMPP networks. This permits, among other things, the extension of the semantic web to XMPP networks, and the fusion of the Semantic Web with the Internet of Things, using XMPP technology [52].
Interoperability: A yet to be published extension defines contracts and interoperability interfaces that can be used to furthermore enhance interoperability in Internet-of-Things applications where an important factor is the ability to mix Things from different manufacturers [53].

Chat: XMPP provides the unique ability to provide a human-to-machine interface to things. A yet to be published extensions furthermore defines a chat interface for Things that can be used both by humans to interact with the things, but also by machines for simple automation using chat messages [54].

3.4 Summary

XMPP is a very mature, secure and easily extensible protocol and suits the needs of the Internet of Things. It can be safely used in an Internet environment, and multiple parties can interact in an interoperable manner without risk of accidentally destroying each other. There are extensions for interoperable and manufacturer independent interchange of information, control operations and even bridging of protocols. All important communication patterns (asynchronous messaging, request/response, publish/subscribe and multicast) used within Internet of Things are also supported. The intrinsic weaknesses of the protocol can be easily overcome except in the most extreme conditions, and they do not leave any large security issues open which can be exploited in the future. In the cases where extreme conditions prohibit the use of XMPP, such parts can still be bridged to the rest of the XMPP network without loss of functionality, by the use of Concentrators [51]. All the above can be accomplished using only manufacturer-independent protocols and extensions.

4 Bridging MQTT and XMPP IoT networks

For use in a Clayster server [55], already supporting the XMPP extensions for the Internet of Things, an MQTT/XMPP bridge has been implemented as MQTT node types using the Concentrators pattern defined in XEP-0326 [51].

First, a sequence of MQTT Server nodes is added to the systems. They provide information on how to connect to the corresponding MQTT servers. During installation phase, the MQTT Server node subscribes to all topics (#), and automatically builds a tree of nodes corresponding to the topics received. Content is classified using the method described earlier, and corresponding MQTT Node Types are added to the system. When placing an MQTT Server node in test or production phase, the subscription changes from all topics to the topics currently being monitored. Apart from the content types discussed above, support for MQTT topics containing XEP-0323 sensor data is also supported, allowing the transmission of sensor data, as defined in XEP-0323 [48]. At this point, all devices published on the MQTT servers are available on XMPP and can be read using XEP-0323 [48].

Furthermore, all node types representing simple content types are also made configurable, and thus available through XEP-325 [50]. This means, that they publish a
single configurable parameter corresponding to the type of the node and content type representing the topic. When this parameter is written to, a PUBLISH message is sent containing the corresponding topic and the new value.

To export compound data objects, a Field Sink has been added to the system that can export sensor data to MQTT in blocks. A Field Sink is a receptor of sensor data that can be used for instance in recurrent readout jobs. Sensor data is siphoned to the field sink, and in this case, it is packaged and published on an MQTT server. The field sink can be configured to either publish data using simple content types using a topic tree, or use a compound XML format and publish compound XML documents on individual topics corresponding to the devices read. XML formats supported include the format defined in XEP-0323, enabling export of XML data compatible with XMPP Internet of Things extensions.

Even though there is a method for mimicking presence status on MQTT [56], no effort has been made to bridging presence status between the networks directly, only data.

5 Conclusion

Even though MQTT and XMPP have their separate pros and cons, MQTT should only be used when XMPP cannot be used due to extreme conditions. And when used, it should only be used in a controlled environment where total control of publishers, subscribers and message brokers can be achieved. In all other cases, XMPP provides a much more secure, stable, extensible and interoperable environment suitable for usage on the Internet by multiple possibly competing parties.

The only case where MQTT can outperform XMPP is in minimizing network traffic and in that case only during some very specific conditions (i.e. not in the general case): Only in a use case where regular publishing of data using the only publish/subscribe pattern, without on-demand support, using a compressed binary compound object, for instance using EXI+SenML [57] can packet size and total required network bandwidth be minimized below to what is possible using EXI+XMPP [39]. The reason here being that the extensions defined for use with XMPP are made to promote interoperability and are more general and contain more meta information that restricted formats such as SenML, which is defined only for “simple sensor measurements”, or as they state themselves in the SenML specification: “There are many types of more complex measurements and measurements that this media type would not be suitable for.” However, even using EXI+SenML over XMPP using publish/subscribe would still create packets that would be slightly larger than the corresponding MQTT case. If these bytes are more important than the other features provided by XMPP, MQTT should be used, however always in a controlled environment and bridged to XMPP as soon as possible.

But by using compound objects in MQTT, provisioning (i.e. controlling who can access what data) becomes impossible. On the other hand, if publishing data using simple content topics or uncompressed compound objects using XML or JSON is the case, this is better implemented in an XMPP environment, as shown in previous sec-
tions. Note here that a small header (as in MQTT) is not the same as a small bandwidth (header + content). Therefore, using simple content topics might actually demand more network bandwidth than using compound objects. But, as stated previously, publishing compound objects eliminates the possibility of providing provisioning functionality on top of the solution. Therefore, using XMPP is recommended in all but the most extreme use cases. A note should also be taken that XMPP can be extended, but MQTT is locked in its communication patterns.

Another interesting aspect of XMPP is that it is the only protocol for Internet-of-Things that provides a human-to-machine interface [54], without the use of special client applications, perhaps with the exception of REST and pages that can be seen in a browser.

In many cases it might be tempting to use MQTT simply because it is simple to start using it. However, just because it is simple, doesn’t mean it is a good idea, unless you weight in all future consequences and limitations imposed on the system. Furthermore, limiting the choice of solution only based on package size to save a few bytes in constrained environments, closely resembles the erroneous decisions taken when deciding to use only two digits to store years in databases, saving one or two bytes per data record, resulting in Y2K problems further ahead. The same can be said about networks and Internet of Things. Taking a decision today, looking only at today’s limitations and not taking into account future development of Internet in general and radio technology in particular, may be a very great mistake and has the capacity of creating huge problems down the road.

6 Acknowledgements

Financial support for this study was provided by KTC [58], Manodo [59] and Weevio [60]. The author wishes to thank Dr. Karin Forsell for her suggestions and comments on this document.

7 References


