# Chatty Things – Making the Internet of Things Readily Usable for the Masses with XMPP

Roland Hieber

Seminar Communications and Multimedia, TU Braunschweig

January 27, 2014

## 1 Introduction

Following the vision of the "Internet of Things", the amount of wireless devices is steadily increasing, which not only improves our standard of living in an age where information is expected to be available at one's fingertips, but also poses challenges. New devices need to be integrated into existing network setups, but when considering sensor nodes with very limited input capabilities, manual configuration can be a cumbersome process – such devices must be readily usable out of the box, and should interoperate with existing infrastructure. Moreover, users need to communicate with their devices, they need to know which devices exist and use the information they collect, while too much or unstructured information leads to information overflow, effectively discouraging the user from using the network.

This paper gives an overview of the "Chatty Things" approach as presented by Klauck and Kirsche [9], which proposes solutions for information filtering, auto-configuration of devices and service discovery, while using standard chat clients for human-to-machine communication. Section 2 introduces the used techniques, Section 3 describes the proposed system architecture for Chatty Things, while Section 4 gives an outlook how this approach can be further enhanced. Finally, Section 5 compares Chatty Things to related techniques and wraps this paper up.

## 2 Prerequisites

In order to build a distributed, failure-tolerant network for our Chatty Things, we will first look at some techniques which can be used to eliminate centralized infrastructure. Most of these techniques are standardized by the IETF and widely used in existing networks. Finally, we will look at XMPP as the basic communication protocol used in Chatty Things.

## 2.1 Address allocation

Considering the TCP/IP protocol suite, in order to be able to communicate on the IP layer, a device needs to configure one of its network interfaces with an IP address that can be reached from the network that the device wants to connect to. Letting the user choose and configure IP addresses manually is a cumbersome when it comes to several devices. Deploying a central server for assigning IP addresses automatically from a pre-configured address pool is possible (e. g. by using DHCP [5]), however, there is also the alternative to use a distributed protocol which enables the devices on a network to choose addresses in accordance with each other, so no IP address is used twice.

In respect to the Internet of Things, this decentralized approach has the advantage that devices can easily be used in different deployments, even where central infrastructures do not exist, and it also allows them to change their addresses dynamically in order to react to changes in the network.

There are two major protocols which are used for dynamic configuration of IP addresses. In the IPv4 world, Link-Local Addressing [2] is often used, and in IPv6 networks, Stateless Address Autoconfiguration [20] is a fundamental feature specified in the IPv6 protocol.

**IPv4 Link-Local Addressing** *Link-Local Addressing*, also known as *Automatic Private IP Addressing (APIPA)* or *Zeroconf*, uses the IPv4 subnet `169.254.0.0/16` for addressing. Every device first chooses a random address from that address space. Then it checks if the chosen address is used by any other device on the network by probing the chosen address, which is usually done using the ARP protocol. If the probing process results that the address is not used on the network (e. g. no device returned an ARP response during a random time interval), the device claims its chosen address and uses it for communication on the IPv4 layer. If the chosen address is already used, the device continues the process, subsequently choosing a new random address and trying to claim it, until a free address has been found.

**IPv6 Stateless Address Autoconfiguration** Similar to IPv4 Link-Local Addressing, devices configured with *IPv6 Stateless Address Autoconfiguration* use an IPv6 address from the subnet `fe80::/64`. First, a 64-bit *interface identifier* is generated, which can be random, or based on the interface's MAC address. Most likely, this interface identifier is unique in the network, so a unique IPv6 address is obtained by combining the subnet prefix and the interface identifier. Nonetheless, to ensure that no other device uses the generated IPv6 address, the device performs *Duplicate Address Detection* on the network by broadcasting its generated address with *Neighbor Advertisement* messages and listening for *Neighbor Solicitation* messages. If such a message is received from another hosts, the generated address cannot be used by the device and must be discarded, and the address generation process is repeated until a unique address has been found.

In contrast to IPv4 Link-Local Addressing, IPv6 Stateless Address Autoconfiguration can also be used with a central server. In this case, the server broadcasts *Router Solicitation* messages on the network which contain a global network prefix. The hosts on the network can then use that prefix instead to configure a global IPv6 address.

## 2.2 Extensions to the Domain Name System

In a distributed context, it is often not feasible to rely on a central, authoritative DNS server, and there is usually no easy way to discover services. The first problem is addressed with *Multicast DNS*, and since DNS is basically a key-value store, it can also be used for service discovery, which is achieved using *DNS-Based Service Discovery.* Both techniques were first developed by Apple as part of the *Bonjour* project[1], and are now maintained by the IETF Zeroconf working group[2].

### 2.2.1 Multicast DNS

*Multicast DNS* (mDNS) [4] describes an extension to the Domain Name System that allows DNS resource records to be distributed on multiple hosts in a network, therefore avoiding central authorities and enabling every host to publish its own entries. For that purpose, a special top-level domain, is used, usually named .local, which contains those entries.

Software that supports mDNS listens on the reserved link-local multicast address 224.0.0.251 (for IPv4 queries) or ff02::fb (for IPv6 queries) on UDP port 5353 for incoming queries. Queries sent to those multicast address and port are standard DNS queries. If a host receives a query and knows about the queried resource, it responds to the querying host with a standard DNS response. The querying host can then simply finish and use the result, or wait until other hosts respond to its query. The latter is typically the case when a record can have multiple values, as it is the case with SRV and PTR records (which will be discussed in the next section).

Another feature of Multicast DNS is the reduction of traffic through *Known-Answer Suppression.* It allows a querying host to specify already known resources in its query when querying resources that could exist on more than one host (e. g., SRV records). The hosts matching those resources then do not generate a response, thus reducing the messages in the network

[1]https://developer.apple.com/bonjour/
[2]http://zeroconf.org

and saving bandwidth, which is usually a scarce resource in wireless networks.

Finally, hosts may also send unsolicited responses. This can be used to notify the network of new services available on a host.

### 2.2.2 DNS-Based Service Discovery

As another recent extension for the Domain Name System, *DNS-Based Service Discovery (DNS-SD)* [3] uses DNS records of types SRV [7] and PTR [12] in a way that allows hosts to browse for services in a domain. While SRV records specify the location of services on a host, PTR records hold a reverse mapping from IP address to host name. DNS-SD now relies on a two-step process, consisting of *Service Instance Enumeration* and *Service Instance Resolution*.

**1. Service Instance Enumeration** At first, to enumerate the available services in a domain for a given protocol, a DNS-SD-enabled client queries PTR resources of the form _service._proto.domain. The result of this query is then a list of *instance names* of the form name._service._proto.domain which point to the hosts providing the service. For example, by querying for _ipp._tcp._example.org, the instance names for all printers supporting the IPP protocol in the domain example.org are returned.

**2. Service Instance Resolution** As a second step, the returned instance names are resolved as SRV records to retrieve the actual host names and port numbers of a service. For example, resolution of one instance name shows that an IPP server is running at host gutenberg.example.org on port 5222. Additionally, an optional TXT record with the same instance name can contain further information about the service (e. g. information about the supported paper sizes).

Through the usage of SRV records, it is easily possible for a service to inform clients about non-standard port numbers, and especially in connection with Multicast DNS, this makes it easy to deploy decentralized systems for the Internet of Things. [10]

## 2.3 XMPP

The *Extensible Messaging and Presence Protocol (XMPP)* is a distributed, XML-based protocol for real-time communication. Its core functionalities are specified in RFC 6120 [14] and RFC 6122 [15], while protocol extensions are usually defined by the XMPP community in *XMPP Extension Proposals (XEPs)*.

### 2.3.1 Addressing

Every user account in XMPP is addressed by a globally unique identifier, called the *Jabber ID (JID)* [13]. It has the form localpart@domain/resource, where domain is the DNS name of an XMPP server, and localpart is the name of a user account on that server. Since a user can be logged in from multiple clients at the same time, the resource part is a string chosen by the user to distinguish those clients. Only the part localpart@domain (the *bare JID*) is needed to identify a user, the resource is only needed for routing between client and server.

### 2.3.2 Architecture

The original architecture underlying XMPP strongly leans on the established design of Internet Mail, and an example is depicted in Fig. 1. The distributed network is formed by *XMPP servers* on one hand, which make up the always-on backbone of the network used for message routing, and which manage user accounts and statuses. On the other hand, *XMPP clients* represent a single logged-in user and make up the interface for communication with other users.

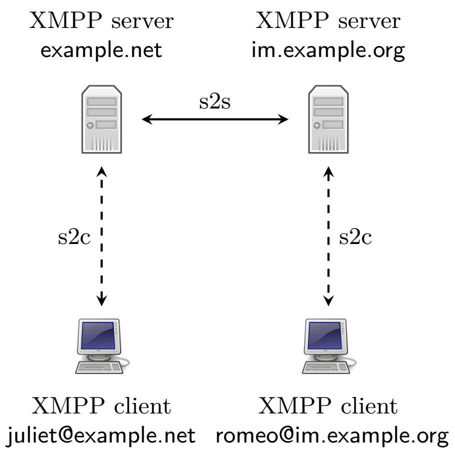


Figure 1: XMPP architecture, showing server-to-server (s2s) and server-to-client (s2c) connections

Every client communicates only with the server that manages the respective user account which is configured in the client, as given in the user's JID. The server then routes the messages to their recipients, using the JID to determine the correct server for a message to be sent to. Finally, the receiving server sends the message to a client where the receiving JID is logged in. If the user is not logged in at the time the message is sent, the server can store it for the user and deliver it on the next login.

XMPP strongly relies on DNS Service Discovery (see Section 2.2.2) to determine the server being in charge of a domain. For example, the server who manages the users for the domain example.org is given by the SRV record _xmpp-server._tcp.example.org.

### 2.3.3 Communication primitives

All communication over XMPP is based on XML. To minimize communication overhead, only fragments of XML, called *stanzas*, are sent between hosts. A stanza is always well-formed as a whole; it consists of a root element, which in most cases also includes routing attributes (to and from), and its optional child elements.

On top of that, living connections between hosts are represented by *XML streams*. The client initiates a connection by sending an XML declaration followed by an opening <stream> tag. The server then responds also with an opening <stream> tag. The client then performs SASL authentication and binds its stream to a resource for proper addressing. If this process succeeded, both client and server can send an unlimited number of stanzas, until the connection is closed by one side by sending a closing </stream> tag. The other side then has the chance to send all outstanding stanzas and then likewise closes its stream. If both streams are closed, the underlying TCP connection is terminated.

### 2.3.4 Publish/Subscribe and Presence

Typically, a user wants to chat with a more or less fixed set of other users, whose JIDs she needs to know, so she needs some kind of "address book" that remembers the JIDs for her. In XMPP, this address book is called *roster*, and it also shows the users' willingness to chat ("presence"). In order to see their chat status (which can be one of "online", "offline", and several "away" or "do not disturb" states), a user needs to subscribe to the other user's status. The mechanism behind this is called *Publish-Subscribe* and is specified in XEP-0060 [11]. It can be used to notify interested users about changes in personal information, and implements the well-known Observer pattern [6].

A user publishes information by creating a *node* on the XMPP server, which acts as a handle for the data. Interested users can then query the server for nodes, and request subscription to them. When the owner of the node confirms the subscription request, subscribers get notified whenever the owner updates the respective node.

All communication takes place between the client and the server over <iq> ("information query") stanzas.

### 2.3.5 Multi-User Chats

Besides one-to-one messaging, XMPP also allows users to create multi-user chat rooms, which is specified in XEP-0045 [18]. Each chat room is given a unique JID on the server managing the room to which the users send their messages to. Each incoming message is then dispatched to all users which have joined the room.

To join a room, the user sends a <presence> stanza to the room JID, where the resource part of the room JID specifies the desired nick name.

### 2.3.6 XMPP Serverless Messaging

To overcome the need for a central server and authentication, XMPP Serverless Messaging [16] allows XMPP clients on a network to build a peer-to-peer mesh network and chat directly with each other. This feature was first introduced by Apple as part of their *Bonjour* project, and nowadays it is also available in many other XMPP clients.

With XMPP Serverless Messaging, XMPP clients simply open a port on their host, and then rely on mDNS and DNS-SD (see Section 2.2) to publish instance names in the domain _presence._tcp.local. For example, if Juliet uses her machine (named capulet) with serverless messaging, her client would publish the following four mDNS records:

- an A record capulet.local, specifying her IP address,
- an SRV record juliet@capulet._presence._tcp.local, specifying the port on which her XMPP client listens, and referring to capulet.local as the host name
- a PTR record _presence._tcp.local for service discovery, pointing to juliet@capulet._presence._tcp.local
- and a TXT record juliet@capulet._presence._tcp.local specifying more information about her (e. g. her online status, contact data, etc.) in standardized key-value pairs.

When other clients in the same network enumerate the available services by querying _presence._tcp.local, they notice Juliet's presence and add her to the roster automatically. In that way, XMPP users can see who is currently available for communication. To start a chat session, clients initiate a TCP connection over the advertised ports, open their XML streams, and send message or IQ stanzas like they would to an XMPP server. Presence is managed over the corresponding TXT record in the mDNS. To go offline, a client announces the deletion of its mDNS records.

## 3 System Architecture of "Chatty Things"

After the underlying techniques have been explained, we can now have a look at the system architecture which Klauck and Kirsche [10] use to build Chatty Things.

### 3.1 Service Provisioning Sublayer

Considering the application in deeply embedded systems and the special needs of the Internet of Things on the one hand, the protocol stack needs to fulfill certain technical requirements.

First, memory, computing resources and bandwidth on embedded systems are limited, which demands for a lightweight protocol stack without too much overhead and predictable memory consumption, while also retaining enough flexibility for future development. The authors therefore provide only the most essential functions of XMPP (presence and message exchange, multi-user chats) to achieve basic communication, grouping of devices, and information filtering to prevent information overflow.

Their solution builds on the *Contiki* operating system on an MSP430 board, and uses the *uXMPP* project, which already implements core XMPP features and serves as a starting point for implementing further XEPs. However, as the uXMPP project was still in early development, they first needed to enhance its functionality to comply with the limited resources. In particular, they optimized the message flow and enabled uXMPP to fully use the available payload instead of sending one TCP packet per message and reduced the code footprint through compiler flags and refactoring.

Furthermore, Klauck and Kirsche implemented new features for uXMPP, which were realized as separate modules to allow enabling and disabling them at runtime, thus further reducing the memory footprint of a running system:

- support for IPv6
- support for Multi-User Chats (XEP-0045), which are used for information filtering
- support for SASL ANONYMOUS login for XMPP servers [17]
- a new publish-subscribe mechanism called Temporary Subscription for Presence (see Section 3.3)
- XMPP Serverless Messaging (XEP-0174), using *uBonjour* as underlying mDNS/DNS-SD implementation for Contiki.

The resulting implementation (uXMPP and uBonjour) gets by with 12.2 kBytes of ROM and 0.63 kBytes of RAM, which was about the size of the original, unoptimized uXMPP implementation while also implementing new features.

In order to react to different network infrastructures, their implementation allows both communication with a central XMPP server as well as peer-to-peer communication over XMPP Serverless Messaging. When a central XMPP server is detected over uBonjour, it is used instead with the ANONYMOUS login method, and the XEP-0174 module is disabled. The ANONYMOUS login method is chosen since TLS encryption is not yet implemented, and with this method, the server assigns a random JID to the client, which does not need to exist on the server. However, a server must exist and must be configured to supply this login method in order for this approach to work.

With a server, information filtering is achieved by creating topic-based Multi-User Chats where multiple devices can be grouped. A user can then simply join the chat with a standard XMPP client on her machine and interact with all devices of a topic, or she can also interact with single devices directly.

In scenarios without an XMPP server, the XEP-0174 module is activated and devices talk directly with the user or with other devices. This method has the drawback that Multi-User Chats cannot be used for topic filtering, since no method is specified to do XEP-0045 and XEP-0174 at the same time. In this case, a user must have an XEP-0174-compliant chat client, but it also gives her the opportunity to interact with things spontaneously on an ad hoc basis (e. g. when entering a room) without need for any additional gateway on the application level.

## 3.2 Bootstrapping

With the given approach, bootstrapping new Chatty Things is easy and no configuration is necessary: on the network layer, IP addresses can simply be obtained using IPv4 Link-Local Addressing or IPv6 Stateless Address Autoconfiguration. On the transport layer, all needed ports can be obtained over DNS-Based Service Discovery. Finally, on the application layer, host names can be resolved over Multicast DNS; and for the actual communication between devices it is possible to use auto-generated JIDs with the ANONYMOUS login method on an existing XMPP server, or if no server is found, use peer-to-peer communication over XMPP Serverless Messaging.

Bootstrapping a Chatty Thing therefore incorporates three steps:

1. Activate uBonjour and try to discover an XMPP server.
2. If a server is found, connect to it using ANONYMOUS login, join topic-based Multi-User Chats, deactivate the uBonjour client *(Infrastructure mode).*
3. If no server is found, activate the XEP-0174 client *(Ad hoc mode).*

During runtime, a device can then react to changes in network infrastructure by changing from one mode to the other:

- In Infrastructure mode: when connection to the server is lost, enable the uBonjour client, try to find a server, and when none is found, enable Serverless Messaging.
- In Ad hoc mode: if uBonjour detects a new XMPP server joining the network, try to connect to it. If this succeeds, disable Serverless Messaging and uBonjour and join topic-based Multi-User Chats.

## 3.3 Temporary Subscription for Presence

To further reduce the message overhead and allow more fine-grained control over information filtering, *Temporary Subscription for Presence* is introduced. This technique builds on top of presence stanzas as defined in core XMPP, which are sent by default without a `to` or `from` attribute, and therefore fit into a single TCP/IP packet over IEEE 802.15.4. However, a drawback of the presence mechanism defined by core XMPP is the fact that a client must manually subscribe to presence information of another client in order to receive it, which requires further communication between the clients. Since the network can change rapidly, and clients can frequently join and leave the network, subscriptions would often be outdated and must be renewed, leading to overhead of subscriptions and unsubscription messages, which would inhibit the flow of the actual information.

To solve this problem, a dynamic, topic-based roster is implemented on top of Multi-User Chats (XEP-0045). Every topic corresponds with a chat room, and nodes join the chat rooms which they are interested in. This allows nodes to inform only interested nodes about updates. This has the advantage that existing clients supporting Multi-User Chats can be used by a user, but Chatty Things and XMPP servers need to be adapted to the new subscription model. Also, this mechanism does not work with Serverless Messaging.

## 4 Outlook

In addition to the XEPs covered above, there are a few additional XEPs which can be implemented to further increase the effectivity of Chatty Things. Especially the documents XEP-0323 through XEP-0326 (which are currently in Experimental status) are targeted to the Internet of Things.

**Concentrators (XEP-0326) [24]** In contrast to sensor nodes which are focused on collecting data, concentrators can be used to serve as a proxy and control a subset of the network. The XEP defines messages to query a sensor node for data sources, and subscribing to them, while subscription is loosely modeled after the Publish-Subscribe mechanism (XEP-0060). It also specifies how clients can request data or control certain nodes over a concentrator.

This approach can be practical in large-scale sensor networks, where usually not every sensor node can be reached directly, and where sensor nodes only have a very limited amount of storage. Individual concentrators can then be equipped with larger storage and serve as a facility to aggregate data from sensor nodes. This structure can be implemented on several levels, forming a hierarchy. A user interested in specific values then only needs to communicate with a single node in the network.

**Sensor Data (XEP-0323) [21]** This XEP specifies a way of reading out values from a sensor node. It allows to specify multiple data sources (e. g. temperature, humidity) as well as multiple types of data (e. g., momentary values, historical values, peak values). As a simple use case, the client sends an IQ stanza containing the request and a sequence number used to identify the request. The sensor node then rejects or accepts the request by returning a corresponding IQ stanza. If it has accepted the request, it reads out the requested data and returns it in a subsequent message stanza to the client.

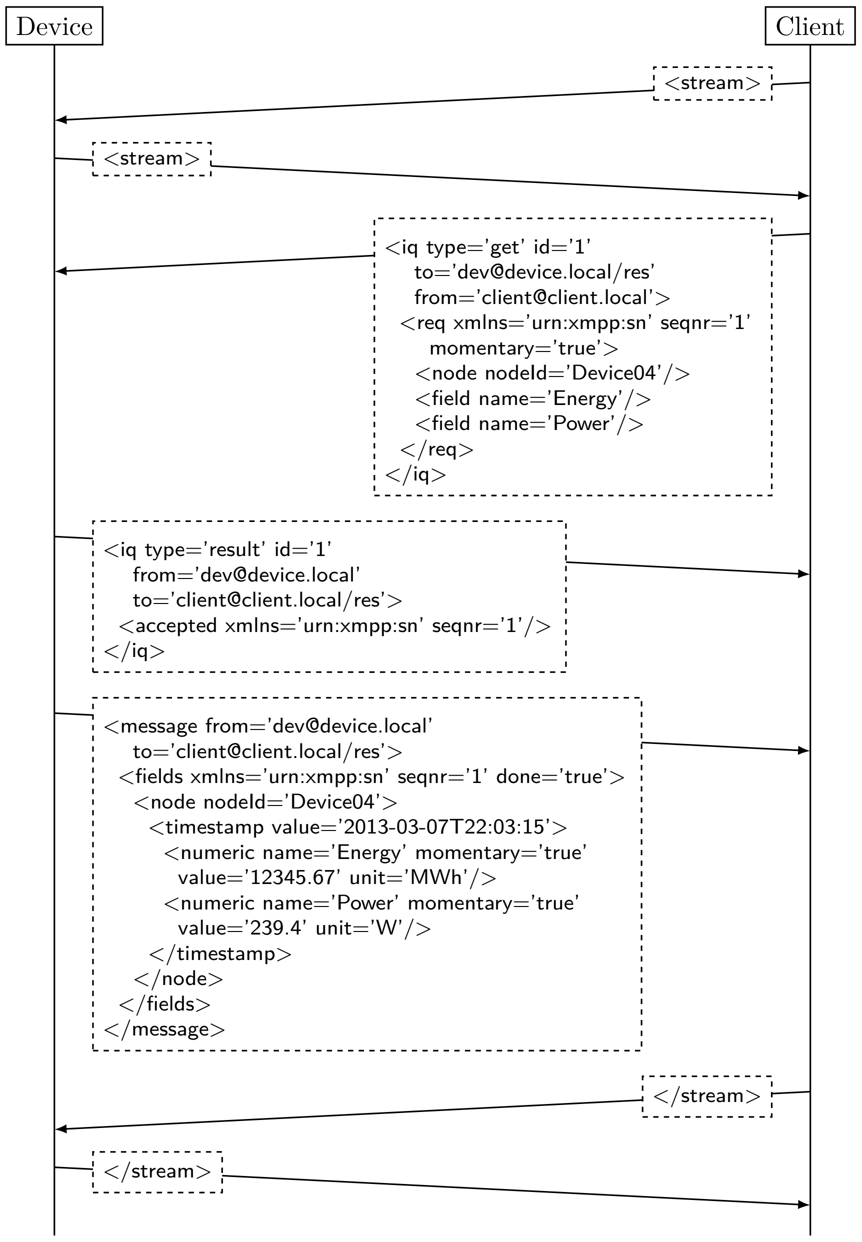
An example of this protocol can be seen in Figure 2: after both clients have opened their streams, the client requests the momentary values for power and energy from the node named *Device04*. The device first acknowledges this request, and, after retrieving the values, sends them back to the client. Afterwards, both sides close their streams.

**Control (XEP-0325) [23]** In this document, a way of controlling sensor nodes is specified, which allows a client to get and set control values on the node over message or IQ stanzas. As an example, in this way a sensor node could be instructed to return data in a different unit or range, or be put into power-safe mode.

**Provisioning (XEP-0324) [22]** To protect the integrity of a sensor network and securing the data being collected, this XEP specifies a way of implementing access rights and user privileges. Since a single sensor node is usually very restricted in user input and output, the approach is very simple and can be implemented e. g. using a button and an LED for interaction, while presentation of data takes places on a provisioning server with a rich user interface (which can be, for example, a concentrator).

When integrating a new sensor node into the network, the user instructs the provisioning server to generate a *friendship* request for the new node. The node can e. g. symbolize this request by blinking its LED and requesting a button press in the next 30 seconds. If the user presses the button, the node confirms the friendship to the server. The server then remembers this sensor node and generates a token which must be used in all further communication between the server and the sensor node, else communication is rejected.

**Efficient XML Interchange Format (EXI, XEP-0322) [25]** Finally, EXI describes how XMPP stanzas sent between nodes can be compressed, thereby effectively reducing the overhead in message size introduced by XML. XMPP nodes can negotiate a compressed stream inside their existing XMPP streams and exchange <compress> stanzas which then contain the payload. However, it is to be noted that this requires further implementation of compression algorithms as well as additional CPU and memory resources and thus might decrease message throughput and increase power consumption on embedded systems.



```
<stream>
```

```
<stream>
```

```
<iq type='get' id='1'
    to='dev@device.local/res'
    from='client@client.local'>
  <req xmlns='urn:xmpp:sn' seqnr='1'
      momentary='true'>
    <node nodeId='Device04'/>
    <field name='Energy'/>
    <field name='Power'/>
  </req>
</iq>
```

```
<iq type='result' id='1'
    from='dev@device.local'
    to='client@client.local/res'>
  <accepted xmlns='urn:xmpp:sn' seqnr='1'/>
</iq>
```

```
<message from='dev@device.local'
    to='client@client.local/res'>
  <fields xmlns='urn:xmpp:sn' seqnr='1' done='true'>
    <node nodeId='Device04'>
      <timestamp value='2013-03-07T22:03:15'>
        <numeric name='Energy' momentary='true'
          value='12345.67' unit='MWh'/>
        <numeric name='Power' momentary='true'
          value='239.4' unit='W'/>
      </timestamp>
    </node>
  </fields>
</message>
```

```
</stream>
```

```
</stream>
```

Figure 2: Example XMPP stream with sensor data (XEP-0323)

## 5 Discussion

### 5.1 Related Approaches

“Chatty Things” is not the only approach to implement communication in embedded networks. This section gives a short overview of related protocols for the Internet of Things and shows their advantages and disadvantages,

Table 1: Comparison of related approaches

| Feature | Chatty Things | CoAP | MQTT | WS4D |
|---|---|---|---|---|
| application gateways necessary | - | yes | yes | - |
| usable with standard clients | yes | - | - | (yes) |
| discovery support | yes | yes | - | yes |
| IPv6/6LoWPAN ready | yes | yes | ? | partial |
| asynchronous messages | yes | yes | | |
| protocol overhead | moderate | small | small | high |

which are summarized in Table 1.

**Constrained Application Protocol (CoAP)** The Constrained Application Protocol [19] focuses on machine-to-machine communication and originates from the IETF Constrained Resources Working Group[3], but still has been only in draft status since 2010. It allows a mapping to HTTP, and is therefore stateless, but it specifies a binary protocol, which makes it necessary to deploy application-level gateways and special client software to communicate with its environment. It relies on UDP, but emulates congestion control, message confirmation and message IDs, since – in contrast to HTTP – messages can be sent asynchronously. Discovery is also specified and done over multicast, service discovery is then done over a well-known URI on the host. Since it is a binary protocol and mostly self-contained, it has low protocol overhead and parsing complexity.

**MQ Telemetry Transport (MQTT)** Specified by IBM as a binary protocol, the MQ Telemetry Transport [8] has been proposed as an OASIS standard for machine-to-machine communication. It also relies on TCP/IP, and its fixed message header is only 2 bytes in size, but can contain further variable headers. Since it is also only used in embedded networks, application gateways and appropriate client software are necessary. Its main feature is a publish-subscribe mechanism with topic names, discovery is not specified.

**Web Service for Devices (WS4D)** As a different approach to avoid application-level gateways, WS4D has been specified as a Devices Profile for Web Services [26]. Since Web Services are wide-spread in the business world, this approach can probably be used in existing infrastructures, and is also focused on multiple platforms like embedded systems and servers. Web Services can be very flexible and composable, and discovery is already specified, however, this also comes at a cost: messages are enclosed in SOAP, which is enclosed in HTTP, which is transported over TCP, which introduces a substantial overhead, especially with SOAP being based on verbose XML.

[3]http://datatracker.ietf.org/wg/core/

IPv6 support is only partially implemented. For communication, standard APIs can be used.

## 5.2 Conclusion

With the XMPP protocol, there is the need to implement at least an XML parser on each node, which comes with protocol overhead and increased code size. However Klauck and Kirsche show that with good optimization (in the code as well as in the protocol), a complete stack can be implemented in 12 kBytes of ROM, which leaves enough space for other applications to be built onto it. As compared to Web Services, Chatty Things are probably not as flexible, but they have less overhead, even when using XML, while MQTT and CoAP provide less flexibility for future enhancement, but less protocol overhead and easier parsing.

With TCP, mDNS, DNS-SD and XMPP as foundation, the proposed architecture builds on reliable and established standards, which allows it to reuse Chatty Things in various contexts without the need for central infrastructure.

Nonetheless, a drawback is the virtual dependency from a centralized XMPP server in order to use Temporary Subscription for Presence for topic filtering, which is caused by the lack of support for Multi-User Chats in XEP-0174 (Serverless Messaging). If this gap can be closed, or a different way for topic filtering in distributed networks is found, the server can be eliminated and what remains is a highly distributed network without the need for much central infrastructure, therefore eliminating most single points of failure in the system.

It is always hard to trade flexibility and accessibility for efficiency. The Chatty Things approach is probably not one of the most efficient, and not the most flexible, but it has the big advantage that users can interact with their things using standard chat clients, without the need for application gateways or specialized software. In terms of efficiency, it chooses a compromise between binary protocols and Web Services, latter which were originally developed for servers with much less resource constraints as embedded systems. However, with additional XEPs focusing on the Internet of Things, enough flexibility can be achieved for this use case.

# References

[1] This work is licensed under the terms of the Creative Commons Attribution-NoDerivs 3.0 Unported license, see https://creativecommons.org/licenses/by-nd/3.0/ for the license text. Figures were drawn using public domain icons from the Tango Desktop Project, see http://tango.freedesktop.org.

[2] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005.

[3] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), February 2013.

[4] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), February 2013.

[5] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Proposed Standard), February 2000.

[8] IBM. MQ Telemetry Transport. http://www.ibm.com/developerworks/webservices/library/ws-mqtt/.

[9] R. Klauck and M. Kirsche. Chatty things – Making the Internet of Things readily usable for the masses with XMPP. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 60–69, 2012.

[10] Ronny Klauck and Michael Kirsche. Bonjour contiki: a case study of a DNS-based discovery service for the internet of things. In *Proceedings of the 11th international conference on Ad-hoc, Mobile, and Wireless Networks*, ADHOC-NOW'12, pages 316–329, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Peter Millard, Peter Saint-Andre, and Ralph Meijer. XEP-0060: Publish-Subscribe. http://xmpp.org/extensions/xep-0060.html, July 2010.

[12] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), November 1987.

[13] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 6122 (Proposed Standard), March 2011.

[14] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.

[15] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.

[16] Peter Saint-Andre. XEP-0174: Serverless Messaging. http://xmpp.org/extensions/xep-0174.html, November 2008.

[17] Peter Saint-Andre. XEP-0175: Best Practices for Use of SASL ANONYMOUS. http://xmpp.org/extensions/xep-0175.html, September 2009.

[18] Peter Saint-Andre. XEP-0045: Multi-User Chat. http://xmpp.org/extensions/xep-0045.html, February 2012.

[19] Z. Shelby, K. Hartke, and C. Bormann. Constrained Application Protocol (CoAP). https://tools.ietf.org/html/draft-ietf-core-coap-18, June 2013.

[20] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[21] Peter Waher. XEP-0323: Internet of Things - Sensor Data. http://xmpp.org/extensions/xep-0323.html, April 2013.

[22] Peter Waher. XEP-0324: Internet of Things - Provisioning. http://xmpp.org/extensions/xep-0324.html, April 2013.

[23] Peter Waher. XEP-0325: Internet of Things - Control. http://xmpp.org/extensions/xep-0325.html, May 2013.

[24] Peter Waher. XEP-0326: Internet of Things - Concentrators. http://xmpp.org/extensions/xep-0326.html, May 2013.

[25] Peter Waher and Yusuke DOI. XEP-0322: Efficient XML Interchange (EXI) Format. http://xmpp.org/extensions/xep-0322.html, July 2013.

[26] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski. WS4D: Toolkits for Networked Embedded Systems Based on the Devices Profile for Web Services. In *39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 1–8, 2010.