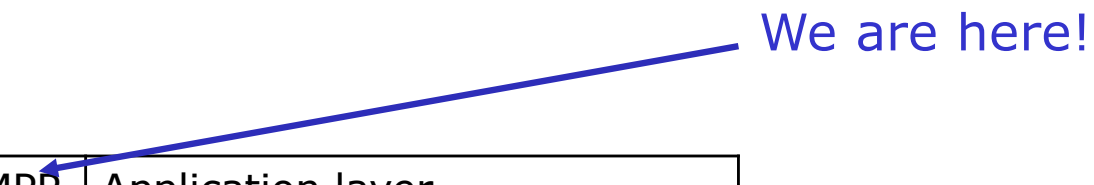
Carnegie Mellon University

95-733 Internet of Things
XMPP and Sensor Andrew

Where are we?

Internet Protocol Suite

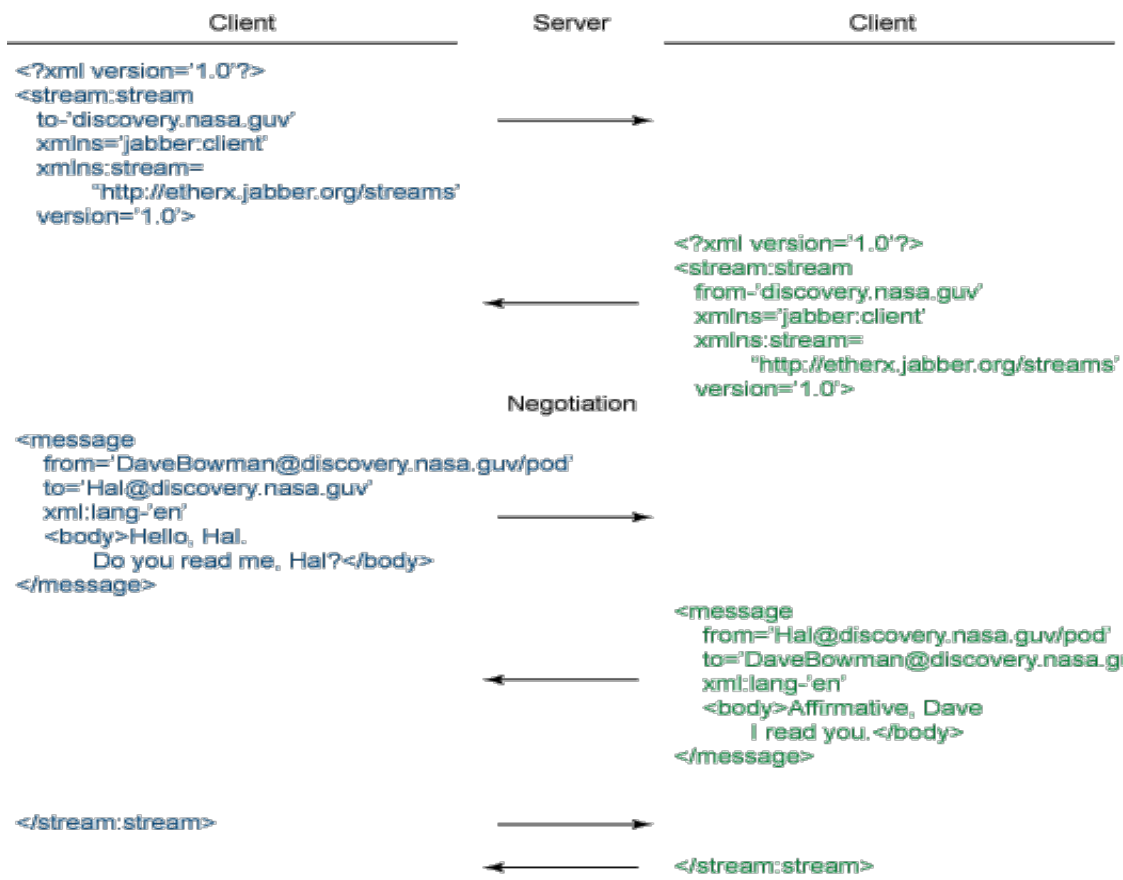


HTTP, Websockets, DNS, XMPP, MQTT, CoAp	Application layer
TLS, SSL	Application Layer (Encryption)
TCP, UDP	Transport
IP(V4, V6), 6LowPAN	Internet Layer
Ethernet, 802.11 WiFi, 802.15.4	Link Layer

XMPP

Extensible Messaging and Presence Protocol

Recall the IBM example



The XML is being transferred in pieces. The TCP connection only closes at the end.

Security parameters are established during negotiation.

Note: there are two XML documents involved.

Would this work over websockets?

Sure. It involves a bidirectional conversation.

Within streams are XML stanzas

- The example above uses the <message> stanza.
- Stanzas are well formed and complete XML messages.
- Three Stanza types enclosed in a stream tag:
- <Presence>
  user status shared with all on the XMPP roster, ”I am online” or ”I am interested in knowing about your presence”, ...
- <IQ>
  information query, request and change settings, discovery of services
- <Message>
  used for person to person chat
- These stanzas have many many options.

Example Messages (1)

- The server pings the client with an information (IQ) stanza

```
<iq from='capulet.lit' to='juliet@capulet.lit/balcony'
      id='s2c1' type='get'>
       <ping xmlns='urn:xmpp:ping'/>
</iq>
```

- The client responds:

```
<iq from='juliet@capulet.lit/balcony' to='capulet.lit'
  id='s2c1' type='result'/>
```

- A server pings another server:

```
<iq from='capulet.lit' to='montague.lit' id='s2s1' type='get'>
    <ping xmlns='urn:xmpp:ping'/>
</iq>
```

Example Messages (2)

- A client pings another client (an end-to-end ping):

```
<iq from='romeo@montague.lit/home'
    to='juliet@capulet.lit/chamber'
    type='get'
    id='e2e1'>
    <ping xmlns='urn:xmpp:ping'/>
</iq>
```

XMPP XML Ping Schema

An XML Schema is used to describe the grammar and vocabulary of an XML language. In this case, we are describing a simple ping message.

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:xmpp:ping'
    xmlns='urn:xmpp:ping'
    elementFormDefault='qualified'>
```

XMPP XML Ping Schema

```
<xs:annotation>
  <xs:documentation>
   The protocol documented by this schema is defined in
   XEP-0199: http://www.xmpp.org/extensions/xep-0199.html
  </xs:documentation>
 </xs:annotation>
```

XMPP XML Ping Schema

```
<xs:element name='ping' type='empty'/>
 <xs:simpleType name='empty'>
   <!– a type is being defined -- >
   <xs:restriction base='xs:string'>
   <!– with a type of string -->
        <xs:enumeration value=''/>  <!– with no content -->
   </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

XMPP Example Messages

- Presence example
- Multiple subscribers receive notifications whenever the publisher (typically an end user) generates an event related to network availability.
- Example publication

```
<presence
    from='juliet@capulet.lit/balcony'>
  <status>Happy</status>
</presence>
```

XMPP Example Messages

- Example messages sent to subscribers:

```
<presence from='juliet@capulet.lit/balcony'
    to='romeo@montague.lit/mobile'>
    <status>Happy</status>
</presence>
<presence from='juliet@capulet.lit/balcony'
    to='nurse@capulet.lit/chamber'>
    <status>Happy</status>
</presence>
<presence from='juliet@capulet.lit/balcony'
    to='benvolio@montague.lit/pda'>
    <status>Happy</status>
</presence>
```

XMPP From the perspective of the application developer

- We do not want to work at the level of XML or JSON.
- We want middleware to provide support.
- Middleware separates concerns. It hides the details associated with messaging.
- Details include marshalling and un-marshaling of parameters and addressing.
- Details include generating the correct XMPP message to send.
- Details include reading and writing messages to the TCP layer.
- At the application programmer level, WE WANT NONE OF THAT!
- Use middleware to hide all of that!

XMPP Client in Ruby

Listing 1. Simple XMPP agent for word definitions (IBM)

```
require 'xmpp4r/client'
# Create a *very* simple dictionary using a hash
hash = {}
hash['ruby'] = 'Great object oriented scripting language'
hash['xmpp4r'] = 'Simple XMPP library for ruby'
hash['xmpp'] = 'Extensible Messaging and Presence Protocol'
# Connect to the server and authenticate
jid = Jabber::JID::new('bot@default.rs/Home')
cl = Jabber::Client::new(jid)
cl.connect
cl.auth('password')
```

XMPP Client in Ruby

```
# Indicate our presence to the server
cl.send Jabber::Presence::new

# Send a salutation to a given user that we're ready
salutation = Jabber::Message::new( 'hal@default.rs', 'DictBot
ready' )
salutation.set_type(:chat).set_id('1')
cl.send salutation
```

XMPP Client in Ruby

```
# Add a message callback to respond to peer requests
cl.add_message_callback do |inmsg|
    # Lookup the word in the dictionary
    resp = hash[inmsg.body]
    if resp == nil
      resp = "don't know about " + inmsg.body
    end
    # Send the response
    outmsg = Jabber::Message::new( inmsg.from, resp )
    outmsg.set_type(:chat).set_id('1')
    cl.send outmsg
end
```

Java uses the Smack API

In order to test the client, we'll need an XMPP server. To do so, create an account on jabber.hot-chilli.net – a free Jabber/XMPP service.

```
import org.jivesoftware.smack.Chat;
import org.jivesoftware.smack.ConnectionConfiguration;
import org.jivesoftware.smack.MessageListener;
import org.jivesoftware.smack.Roster;
import org.jivesoftware.smack.RosterEntry;
import org.jivesoftware.smack.XMPPConnection;
import org.jivesoftware.smack.XMPPException;
import org.jivesoftware.smack.packet.Message;
// Works with Android
```

Java uses the Smack API

```
private XMPPConnection connection;
public void login(String userName, String password) throws
                        XMPPException {
    // Use a local XMPP server
    ConnectionConfiguration config = new
                    ConnectionConfiguration("localhost", 5222);
    connection = new XMPPConnection(config);
    connection.connect();
    connection.login(userName, password);
}
```


95-733 Internet of Things

Java uses the Smack API

```
public void displayBuddyList() {
     Roster roster = connection.getRoster();
     Collection<RosterEntry> entries = roster.getEntries();
     System.out.println("\n\n" + entries.size() + " buddy(ies):");
      for(RosterEntry r:entries) {
          System.out.println(r.getUser());
       }
 }
```

Many XMPP Javascript libraries exist for real time chat within a browser over websockets.

XMPP And Things

XMPP Thing Discovery allows users to build IoT services and applications using things without the need for a priori knowledge of things.

https://ubiquity.acm.org/article.cfm?id=2822529

General Discovery[1]

- A naming service provides service references if you have the name in hand. Example: Phone book, the Domain Name Service (DNS).
- A directory service provides service references if you have attributes in hand. Examples include Google search and the Lightweight Directory Access Protocol (LDAP).
- A discovery service is a directory service that allows registration, de-registration, and lookUp of services in a spontaneous network – where clients and services may come and go. Example: You provide printer attributes and discover a printer.
- Discovery may be done with a directory server or be serverless.
- In serverless discovery, participants collaborate to implement a decentralized discovery service. Two main models:

  push model : services regularly advertise their services (multicast).

  pull model: clients multicast their queries.

1. Notes from Coulouris text on Distributed Systems

XMPP and Thing Discovery

- During production of a Thing, decisions have to be made whether the following parameters should be pre-configured, manually entered after installation or automatically found and/or created by the device if possible (zero-configuration networking):

  Parameters:

  - Address and domain of XMPP Server.
  - JID of the Thing.
  - JID of Thing Registry, if separate from the XMPP Server.
  - JID of Provisioning server, if different from the Registry and XMPP server
- A provisioning server may be needed if the device needs special configuration parameters from that server. Perhaps with user involvement.

XMPP and Thing Discovery

- If the address of an XMPP Server is not preconfigured, the thing must attempt to find one in its local surroundings. This can be done using one of several methods:

  Dynamic Host Configuration Protocol (DHCP)

  - Server returns IP address as needed – device issues a query to a well known broadcast address. Response may include DNS location.

  Multicast Domain Naming Service (mDNS)

  - small network, no DNS server, P2P
  - multicast a query message with a name. The server with that name responds (broadcasts) its IP address. Machines may update caches. Build a directory of name, ip mappings

XMPP and Thing Discovery

- Simple Service Discovery protocol SSDP/UPnP (Universal Plug n Play)
  - No DNS or DHCP, Uses UDP and multicast addresses
  - A UPnP compatible device from any vendor can dynamically join a network, obtain an IP address, announce its name, advertise or convey its capabilities upon request, and learn about the presence and capabilities of other devices.[1] Uses XML messages.
- The XMPP server and registry are found and the Thing registers itself with the following XMPP message:

1. Wikipedia

XMPP and Thing Registration

```
<iq type='set'
      from='thing@example.org/imc'
      to='discovery.example.org'
      id='1'>
     <register xmlns='urn:xmpp:iot:discovery'>
        <str name='SN' value='39487234732948723'/>
        <str name='MAN' value='www.ktc.se'/>
        <str name='MODEL' value='IMC'/>
        <num name='V' value='1.2'/>
        <str name='KEY' value='4857402340298342'/>
     </register>
  </iq>
```

Suppose a sensor is registered. How do we read from it?

Reading sensor data

- Request to a Thing for an Automatic Meter Reading

```
<iq type='get' from='client@clayster.com/amr'
                    to='device@clayster.com' id='S0001'>
      <req xmlns='urn:xmpp:iot:sensordata' seqnr='1’
           momentary='true'/>
</iq>
```

- Response from the Thing – I got your request

```
<iq type='result' from='device@clayster.com'
                        to='client@clayster.com/amr' id='S0001'>
      <accepted xmlns='urn:xmpp:iot:sensordata' seqnr='1'/>
</iq>
```

Data arrives from a sensor

```
<message from='device@clayster.com'
              to='client@clayster.com/amr'>
    <fields xmlns='urn:xmpp:iot:sensordata' seqnr='1' done='true'>
        <node nodeId='Device01'>
            <timestamp value='2013-03-07T16:24:30'>
                <numeric name='Temperature' momentary='true'
                        automaticReadout='true' value='23.4' unit='°  C'/>
                <numeric name='load level' momentary='true'
                        automaticReadout='true'
                        value='75' unit='%'/>
            </timestamp>
        </node>
  </fields>
</message>

<!– Note how the units are stated and the lack of ambiguity -->
```

Sensor Andrew Based on XMPP (2007)

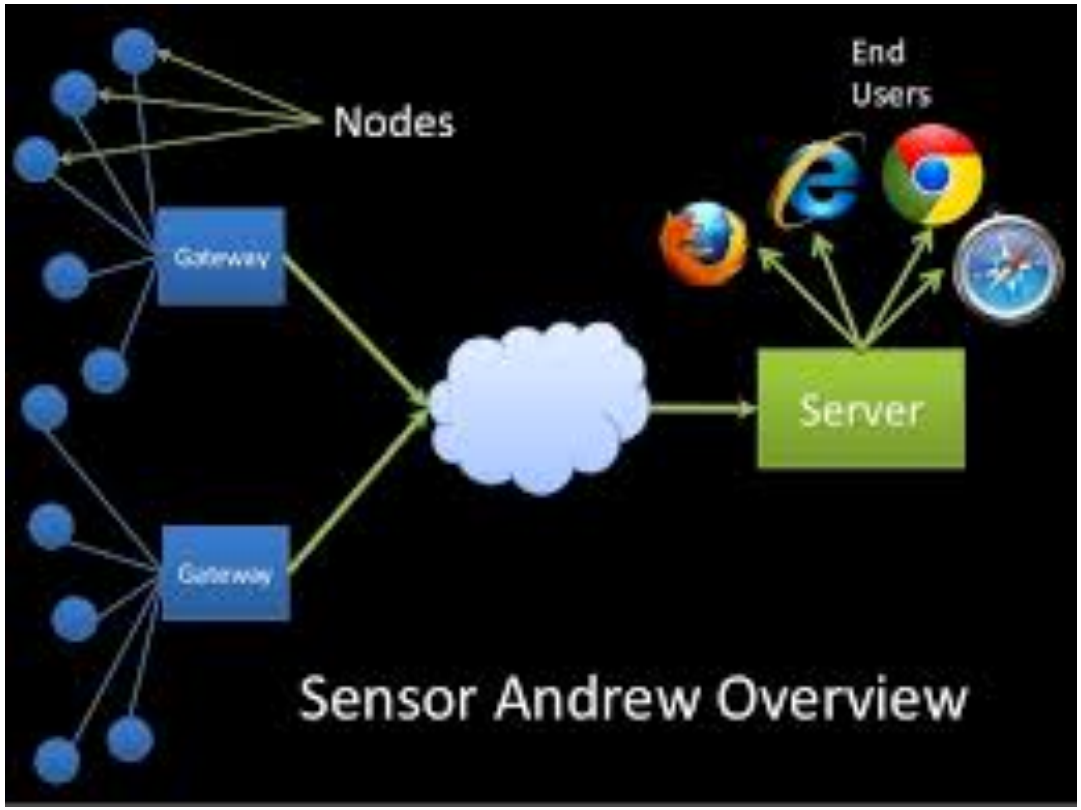
Non-functional characteristics:
Open (XMPP)
- Standards based
- Standard message formats

Heterogeneous sensors
Security, Privacy Challenges
Reliable (ejabberd – Erlang open source)
Fault tolerant (ejabberd, Erlang)
- In ejabberd all information can be stored on more than one node, nodes can be added or replaced "on the fly". Erlang is big on handling failures

Performance (speed) XML is typically far slower than compact binary messages
Extensible
Manageable
Cost

95-733 Internet of Things

Sensor Andrew



A good architecture survives change. What could change?

Price of things
Variety of things (sensors)
Applications
Ubiquity of networks
Speed of networks
Battery life
Speed of processors
Effects of failure
Government regulations?

We do not allow cars on the road without seatbelts. We may need governments to regulate IOT devices for security. New California IoT law goes into effect January 1, 2020