APPIOT LAB-4

XMPP

Shubhika GARG

PART-1

1. On the website of sleekxmpp run the examples:

- **SleekXMPP Quickstart - Echo Bot**
- **Sign in, Send a Message, and Disconnect**

Explain the purpose of each example. You may use Bob and Alice accounts. These accounts are associated to "usertp-VirtualBox" domain and not to "example.com".

a. Echo Bot

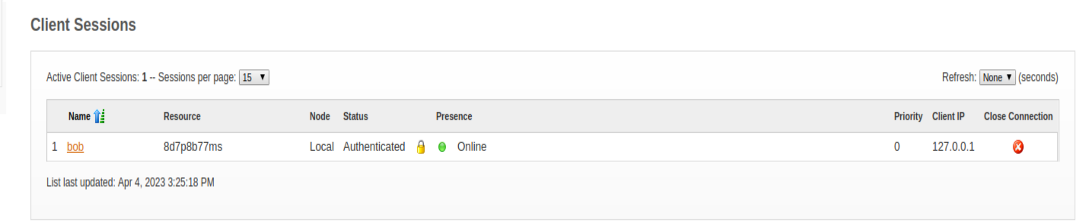
The purpose of the echo bot is to listen for incoming chat messages and then respond with a simple echo of the message received, along with a "Thanks for sending" message.

The Echo Bot class extends the sleekxmpp.ClientXMPP class and overrides the init, start, and message methods. The event handlers for session_start and message events are added in the init method.

The start method is called when the session_start event is triggered and sends the initial presence and requests the roster.

The message method is called whenever a message is received and checks if the message type is either 'chat' or 'normal' before replying with an echo of the message. It will process the incoming messages but it should be aware that the messages also include MUC messages and error messages.

On the Openfire server we can see that the bob is shown online after the start of the echo bot server.



Client Sessions

Active Client Sessions: **1** -- Sessions per page: 15 ▼

Refresh: None ▼ (seconds)

	Name	Resource	Node	Status	Presence	Priority	Client IP	Close Connection
1	bob	8d7p8b77ms	Local	Authenticated 🔒	● Online	0	127.0.0.1	☒

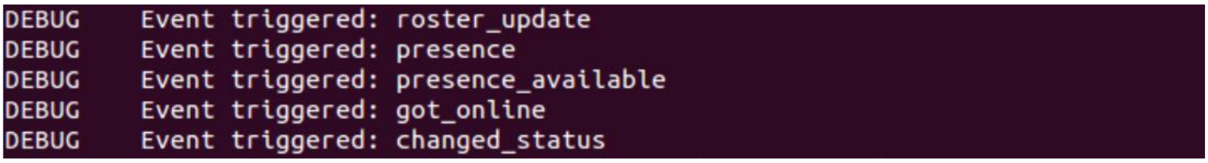
List last updated: Apr 4, 2023 3:25:18 PM

b. Sign in, Send a Message, and Disconnect

In this code, the chat bot logs in, sends a message to a recipient, and then logs out. The code uses the SleekXMPP library to establish an XMPP connection and send messages.

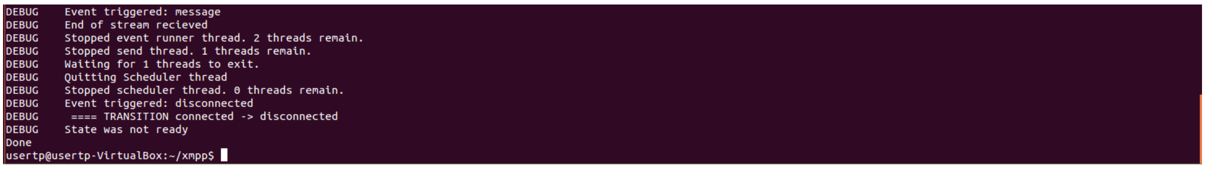
Once the bot establishes its connection with the server and the XML streams are ready for use, the "session_start" event is triggered, which initializes the bot's roster. The bot then sends a message to the recipient using the provided JID and message, and logs out.

Here, we have a session_start which will broadcast its presence, get the roster, send the message and then disconnect.



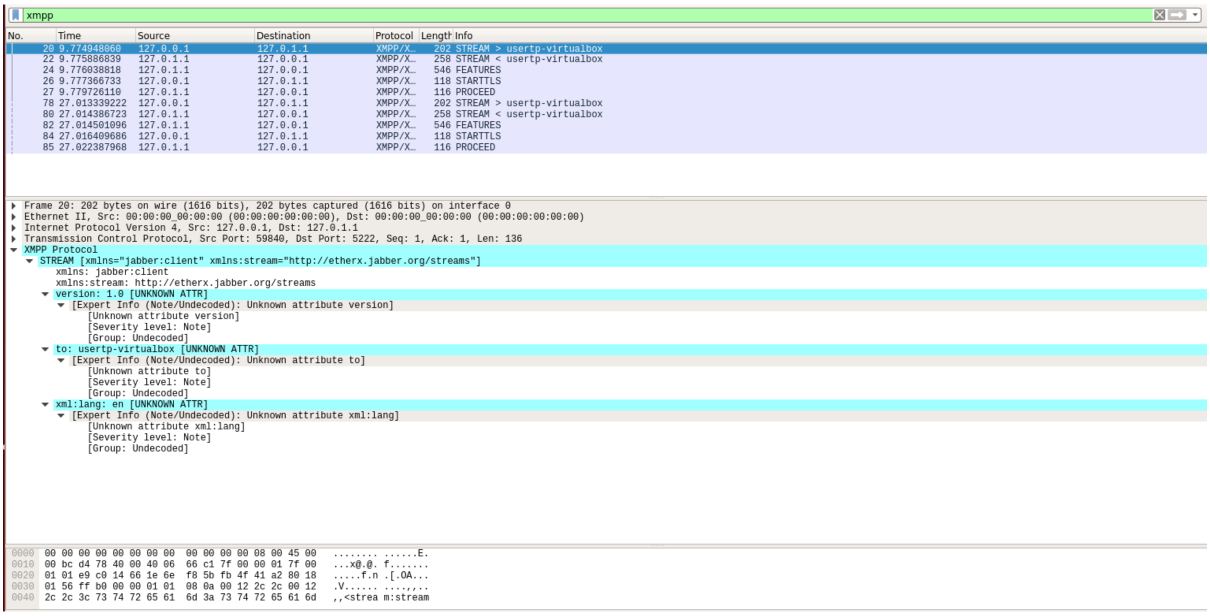
```
DEBUG    Event triggered: roster_update
DEBUG    Event triggered: presence
DEBUG    Event triggered: presence_available
DEBUG    Event triggered: got_online
DEBUG    Event triggered: changed_status
```

Below snip shows disconnection from the client:



```
DEBUG    Event triggered: message
DEBUG    End of stream recieved
DEBUG    Stopped event runner thread. 2 threads remain.
DEBUG    Stopped send thread. 1 threads remain.
DEBUG    Waiting for 1 threads to exit.
DEBUG    Quitting Scheduler thread
DEBUG    Stopped scheduler thread. 0 threads remain.
DEBUG    Event triggered: disconnected
DEBUG     ==== TRANSITION connected -> disconnected
DEBUG    State was not ready
Done
usertp@usertp-VirtualBox:~/xmpp$ 
```

2. Start Wireshark, and launch a capture. Try each application. Stop the capture.



```
xmpp
No.  Time          Source       Destination  Protocol Length Info
  20 9.774948060   127.0.0.1    127.0.1.1    XMPP/X…   202 STREAM > usertp-virtualbox
  22 9.775886839   127.0.1.1    127.0.0.1    XMPP/X…   258 STREAM < usertp-virtualbox
  24 9.776038818   127.0.1.1    127.0.0.1    XMPP/X…   546 FEATURES
  26 9.777366733   127.0.0.1    127.0.1.1    XMPP/X…   118 STARTTLS
  27 9.779726110   127.0.1.1    127.0.0.1    XMPP/X…   116 PROCEED
  78 27.013339222  127.0.0.1    127.0.1.1    XMPP/X…   202 STREAM > usertp-virtualbox
  80 27.014386723  127.0.1.1    127.0.0.1    XMPP/X…   258 STREAM < usertp-virtualbox
  82 27.014501096  127.0.1.1    127.0.0.1    XMPP/X…   546 FEATURES
  84 27.016409686  127.0.0.1    127.0.1.1    XMPP/X…   118 STARTTLS
  85 27.022387968  127.0.1.1    127.0.0.1    XMPP/X…   116 PROCEED

▸ Frame 20: 202 bytes on wire (1616 bits), 202 bytes captured (1616 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.1.1
▸ Transmission Control Protocol, Src Port: 59840, Dst Port: 5222, Seq: 1, Ack: 1, Len: 136
▾ XMPP Protocol
  ▾ STREAM [xmlns="jabber:client" xmlns:stream="http://etherx.jabber.org/streams"]
      xmlns: jabber:client
      xmlns:stream: http://etherx.jabber.org/streams
    ▾ version: 1.0 [UNKNOWN ATTR]
      ▾ [Expert Info (Note/Undecoded): Unknown attribute version]
          [Unknown attribute version]
          [Severity level: Note]
          [Group: Undecoded]
    ▾ to: usertp-virtualbox [UNKNOWN ATTR]
      ▾ [Expert Info (Note/Undecoded): Unknown attribute to]
          [Unknown attribute to]
          [Severity level: Note]
          [Group: Undecoded]
    ▾ xml:lang: en [UNKNOWN ATTR]
      ▾ [Expert Info (Note/Undecoded): Unknown attribute xml:lang]
          [Unknown attribute xml:lang]
          [Severity level: Note]
          [Group: Undecoded]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 bc d4 78 40 00 40 06  66 c1 7f 00 00 01 7f 00   ...x@.@. f.......
0020  01 01 e9 c0 14 66 1e 6e  f8 5b fb 4f 41 a2 80 18   .....f.n .[.OA...
0030  01 56 ff b0 00 00 01 01  08 0a 00 12 2c 2c 00 12   .V...... ....,,..
0040  2c 2c 3c 73 74 72 65 61  6d 3a 73 74 72 65 61 6d   ,,<strea m:stream
```

3. What are the exchanged messages? Draw an exchange diagram.

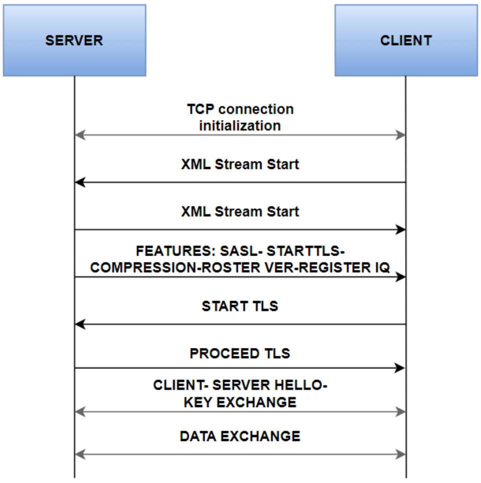
Extensible Messaging and Presence Protocol (XMPP) uses stanzas to exchange messages between clients and servers. There are three types of stanzas used in XMPP: message, presence, and IQ.

Message stanzas are used to send data from one entity to another, and they come in five different types: normal, chat, group chat, headline, and error.

Presence stanzas are used to broadcast network availability to other clients, allowing clients to receive messages from subscribed clients.

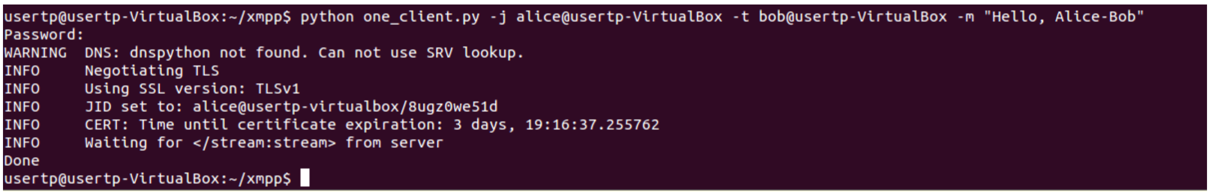
IQ stanzas are used to request and retrieve information and only contain a single payload.

Before sending stanzas, the XMPP client and server establish a TCP connection, authenticate, and encrypt the communication. Once authenticated, the stanzas are sent as XML streams.


SERVER
CLIENT
TCP connection initialization
XML Stream Start
XML Stream Start
FEATURES: SASL- STARTTLS- COMPRESSION-ROSTER VER-REGISTER IQ
START TLS
PROCEED TLS
CLIENT- SERVER HELLO- KEY EXCHANGE
DATA EXCHANGE

4. Why are we not seeing the messages exchanged after the authentication?

Encryption is done using the TLS (Transport Layer Security) parameters between the client and server and vice versa. TLS ensures that the data is transmitted securely and cannot be intercepted by unauthorized parties. Hence, the message is not directly visible in Wireshark, but can be read after decryption.



```
usertp@usertp-VirtualBox:~/xmpp$ python one_client.py -j alice@usertp-VirtualBox -t bob@usertp-VirtualBox -m "Hello, Alice-Bob"
Password:
WARNING  DNS: dnspython not found. Can not use SRV lookup.
INFO     Negotiating TLS
INFO     Using SSL version: TLSv1
INFO     JID set to: alice@usertp-virtualbox/8ugz0we51d
INFO     CERT: Time until certificate expiration: 3 days, 19:16:37.255762
INFO     Waiting for </stream:stream> from server
Done
usertp@usertp-VirtualBox:~/xmpp$ 
```

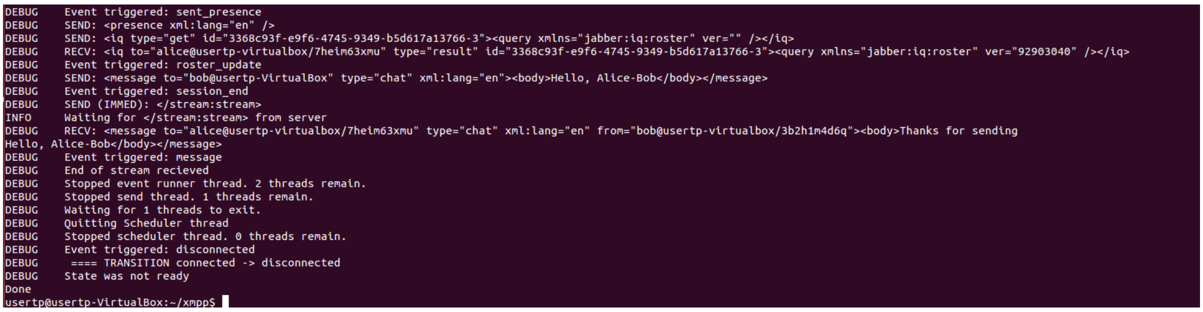
5. Now run the applications using the debug mode (option –d). Explain the exchanged messages shown in the console?

The "-d" option enables the debug mode. The debug mode will provide additional information during the execution of the script.

In debug mode, the SEND output will show the raw XML message being sent from the client to the server, including any necessary headers, authorization information, and the message body.

The RECV output will show the corresponding XML response received from the server, including any headers and response information.

In the debug mode, we can also see the exchange of JIDs (Jabber Identifier) between the client and the server.



```
DEBUG    Event triggered: sent_presence
DEBUG    SEND: <presence xml:lang="en" />
DEBUG    SEND: <iq type="get" id="3368c93f-e9f6-4745-9349-b5d617a13766-3"><query xmlns="jabber:iq:roster" ver="" /></iq>
DEBUG    RECV: <iq to="alice@usertp-virtualbox/7heim63xmu" type="result" id="3368c93f-e9f6-4745-9349-b5d617a13766-3"><query xmlns="jabber:iq:roster" ver="92903040" /></iq>
DEBUG    Event triggered: roster_update
DEBUG    SEND: <message to="bob@usertp-VirtualBox" type="chat" xml:lang="en"><body>Hello, Alice-Bob</body></message>
DEBUG    Event triggered: session_end
DEBUG    SEND (IMMED): </stream:stream>
INFO     Waiting for </stream:stream> from server
DEBUG    RECV: <message to="alice@usertp-virtualbox/7heim63xmu" type="chat" xml:lang="en" from="bob@usertp-virtualbox/3b2h1m4d6q"><body>Thanks for sending
Hello, Alice-Bob</body></message>
DEBUG    Event triggered: message
DEBUG    End of stream recieved
DEBUG    Stopped event runner thread. 2 threads remain.
DEBUG    Stopped send thread. 1 threads remain.
DEBUG    Waiting for 1 threads to exit.
DEBUG    Quitting Scheduler thread
DEBUG    Stopped scheduler thread. 0 threads remain.
DEBUG    Event triggered: disconnected
DEBUG     ==== TRANSITION connected -> disconnected
DEBUG    State was not ready
Done
usertp@usertp-VirtualBox:~/xmpp$ 
```

PART-2

1. In the directory xmpp, you may find two codes. One code is called a client, and the other a server. To recall Openfire is associated to “usertp-VirtualBox” domain. Explain the code and indicate where the IOT resources are available?

Client:

The code is a Python implementation of a client for the XMPP. It is specifically designed to communicate with an IoT sensor device over XMPP. The code uses the SleekXMPP library, which provides an XMPP client framework for Python.

The class **IoT_Client** extends the sleekxmpp.ClientXMPP class.

The **init()** method of the IoT_Client class takes three arguments: the JID of the client, the password for the client's JID, and the JID of the sensor device that the client will communicate with. The method also sets up several event handlers and initializes some instance variables.

The **session_start()** method is called when the XMPP session starts, and sends a presence message and retrieves service discovery information from the sensor device.

The **message()** method is called when a message is received from the sensor device, and responds with a message containing the JID and IP address of the client.

The **datacallback()** method is called when data is received from the sensor device, and prints out the data in a tabular format.

Server:

The code is an implementation of an XMPP server that can handle IoT devices using the XEP-0323 standard. The server uses the SleekXMPP library to handle XMPP connections.

The code starts by importing the necessary modules:

- The logging module for debugging purposes
- The sys module for system-related functionality
- The optparse module for parsing command-line arguments
- The socket module for obtaining the IP address of the server
- The sleekxmpp module for handling XMPP connections. If the Python version being used is less than version 3, the code sets the default encoding to UTF-8.

The **IoT_Server** class is defined, which inherits from the **ClientXMPP** class in the sleekxmpp module.

- The class has an **init()** method that initializes the object, adds event handlers for session_start and message events, and sets the device attribute to **None**.
- The **testForRelease()** method returns the value of the releaseMe attribute
- The **doReleaseMe()** method sets the value of the releaseMe attribute to True.
- The **addDevice()** method is used to add a device to the server.
- The **session_start()** method is called when the session starts and sends a presence signal and gets the roster.
- The **message()** method is called when a message is received and checks the type of message. If the message type is 'chat' or 'normal', it obtains the IP address of the server and sends a reply message containing the JID, IP address, and instructions for communicating using the XEP-0323 standard.

The **IoT_Device** class is defined, which inherits from the Device class in the sleekxmpp.plugins.xep_0323.device module.

- This class has an **init()** method that initializes the object and sets the temperature attribute to 25.
- The **refresh()** method is called to refresh the device data and increments the temperature value by one.
- The **update_sensor_data()** method is called to update the sensor data and adds a field named "**Temperature**".
  It then sets the momentary timestamp and adds the momentary data for the temperature field.
- The **get_temperature()** method returns the temperature value as a string.

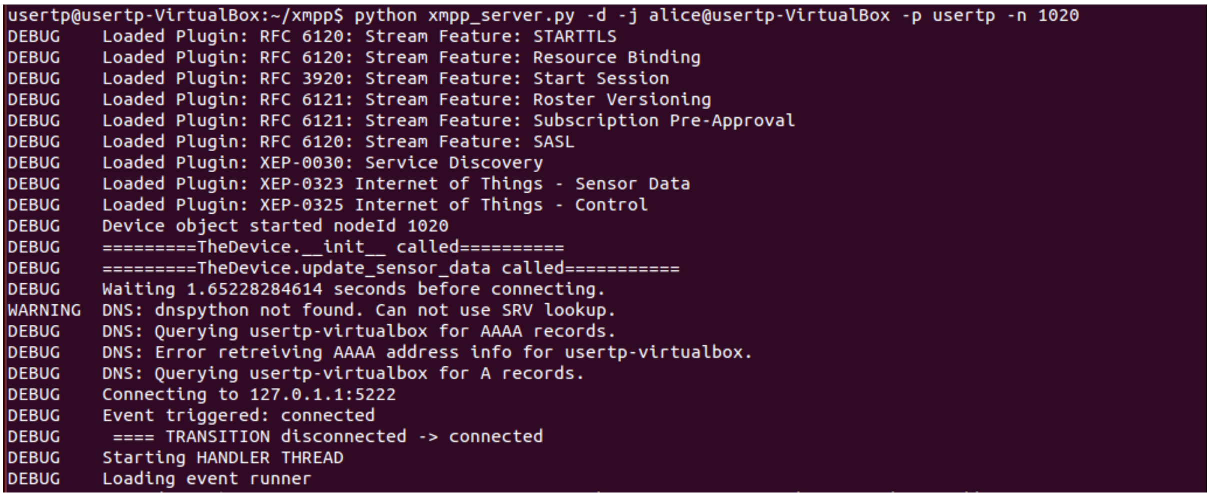
The available IoT resources in XMPP are:

- XEP 0030
- XEP 0323
- XEP 0325

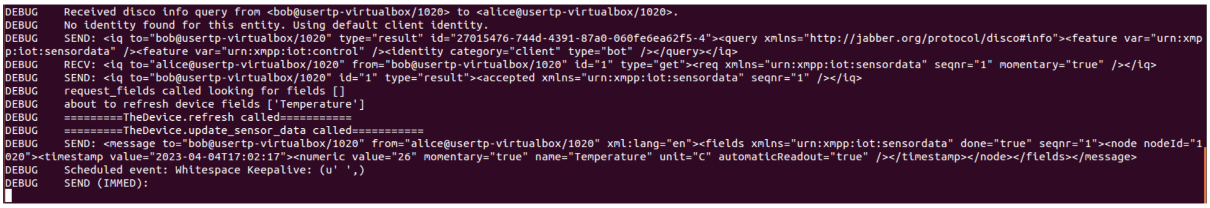
These XEPs enable XMPP clients and servers to communicate and exchange data with IoT devices and services, making XMPP a viable option for IoT applications.

2. Run both the client and server (start with the server).

Server:

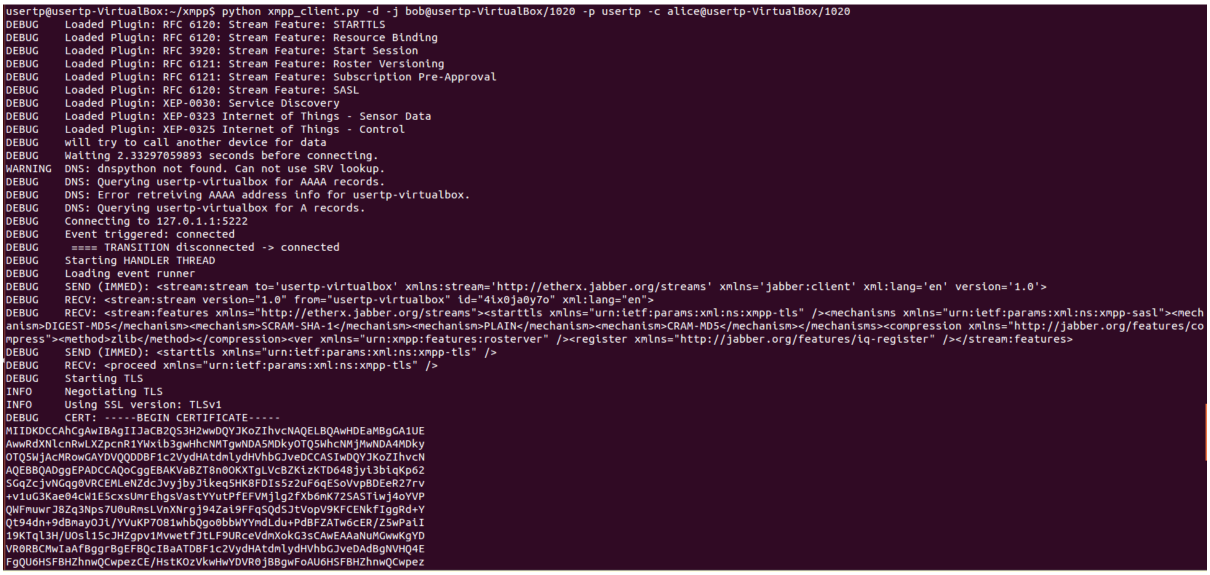


```
usertp@usertp-VirtualBox:~/xmpp$ python xmpp_server.py -d -j alice@usertp-VirtualBox -p usertp -n 1020
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: STARTTLS
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: Resource Binding
DEBUG    Loaded Plugin: RFC 3920: Stream Feature: Start Session
DEBUG    Loaded Plugin: RFC 6121: Stream Feature: Roster Versioning
DEBUG    Loaded Plugin: RFC 6121: Stream Feature: Subscription Pre-Approval
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: SASL
DEBUG    Loaded Plugin: XEP-0030: Service Discovery
DEBUG    Loaded Plugin: XEP-0323 Internet of Things - Sensor Data
DEBUG    Loaded Plugin: XEP-0325 Internet of Things - Control
DEBUG    Device object started nodeId 1020
DEBUG    =========TheDevice.__init__ called==========
DEBUG    =========TheDevice.update_sensor_data called==========
DEBUG    Waiting 1.65228284614 seconds before connecting.
WARNING  DNS: dnspython not found. Can not use SRV lookup.
DEBUG    DNS: Querying usertp-virtualbox for AAAA records.
DEBUG    DNS: Error retreiving AAAA address info for usertp-virtualbox.
DEBUG    DNS: Querying usertp-virtualbox for A records.
DEBUG    Connecting to 127.0.1.1:5222
DEBUG    Event triggered: connected
DEBUG     ==== TRANSITION disconnected -> connected
DEBUG    Starting HANDLER THREAD
DEBUG    Loading event runner
```

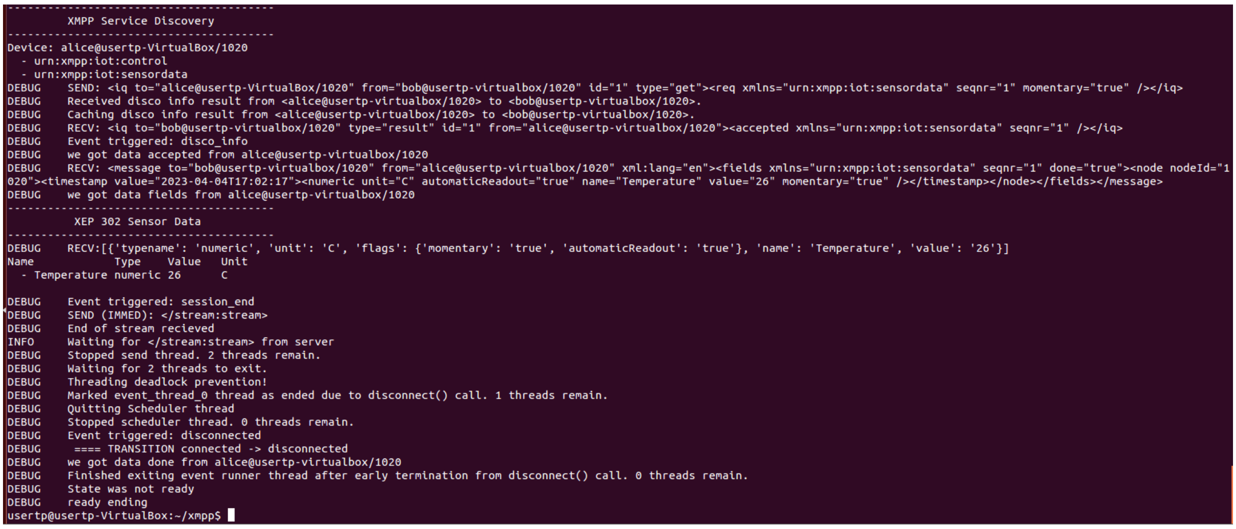


```
DEBUG    Received disco info query from <bob@usertp-virtualbox/1020> to <alice@usertp-virtualbox/1020>.
DEBUG    No identity found for this entity. Using default client identity.
DEBUG    SEND: <iq to="bob@usertp-virtualbox/1020" type="result" id="27015476-744d-4391-87a0-060fe6ea62f5-4"><query xmlns="http://jabber.org/protocol/disco#info"><feature var="urn:xmp
p:iot:sensordata" /><feature var="urn:xmpp:iot:control" /><identity category="client" type="bot" /></query></iq>
DEBUG    RECV: <iq to="alice@usertp-virtualbox/1020" from="bob@usertp-virtualbox/1020" id="1" type="get"><req xmlns="urn:xmpp:iot:sensordata" seqnr="1" momentary="true" /></iq>
DEBUG    SEND: <iq to="bob@usertp-virtualbox/1020" id="1" type="result"><accepted xmlns="urn:xmpp:iot:sensordata" seqnr="1" /></iq>
DEBUG    request_fields called looking for fields []
DEBUG    about to refresh device fields ['Temperature']
DEBUG    =========TheDevice.refresh called==========
DEBUG    =========TheDevice.update_sensor_data called==========
DEBUG    SEND: <message to="bob@usertp-virtualbox/1020" from="alice@usertp-virtualbox/1020" xml:lang="en"><fields xmlns="urn:xmpp:iot:sensordata" done="true" seqnr="1"><node nodeId="1
020"><timestamp value="2023-04-04T17:02:17"><numeric value="26" momentary="true" name="Temperature" unit="C" automaticReadout="true" /></timestamp></node></fields></message>
DEBUG    Scheduled event: Whitespace Keepalive: (u' ',)
DEBUG    SEND (IMMED):
```

Client:



```
usertp@usertp-VirtualBox:~/xmpp$ python xmpp_client.py -d -j bob@usertp-VirtualBox/1020 -p usertp -c alice@usertp-VirtualBox/1020
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: STARTTLS
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: Resource Binding
DEBUG    Loaded Plugin: RFC 3920: Stream Feature: Start Session
DEBUG    Loaded Plugin: RFC 6121: Stream Feature: Roster Versioning
DEBUG    Loaded Plugin: RFC 6121: Stream Feature: Subscription Pre-Approval
DEBUG    Loaded Plugin: RFC 6120: Stream Feature: SASL
DEBUG    Loaded Plugin: XEP-0030: Service Discovery
DEBUG    Loaded Plugin: XEP-0323 Internet of Things - Sensor Data
DEBUG    Loaded Plugin: XEP-0325 Internet of Things - Control
DEBUG    will try to call another device for data
DEBUG    Waiting 2.33297059893 seconds before connecting.
WARNING  DNS: dnspython not found. Can not use SRV lookup.
DEBUG    DNS: Querying usertp-virtualbox for AAAA records.
DEBUG    DNS: Error retreiving AAAA address info for usertp-virtualbox.
DEBUG    DNS: Querying usertp-virtualbox for A records.
DEBUG    Connecting to 127.0.1.1:5222
DEBUG    Event triggered: connected
DEBUG     ==== TRANSITION disconnected -> connected
DEBUG    Starting HANDLER THREAD
DEBUG    Loading event runner
DEBUG    SEND (IMMED): <stream:stream to='usertp-virtualbox' xmlns:stream='http://etherx.jabber.org/streams' xmlns='jabber:client' xml:lang='en' version='1.0'>
DEBUG    RECV: <stream:stream version="1.0" from="usertp-virtualbox" id="4ix0ja0y7o" xml:lang="en">
DEBUG    RECV: <stream:features xmlns="http://etherx.jabber.org/streams"><starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls" /><mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl"><mech
anism>DIGEST-MD5</mechanism><mechanism>SCRAM-SHA-1</mechanism><mechanism>PLAIN</mechanism><mechanism>CRAM-MD5</mechanism></mechanisms><compression xmlns="http://jabber.org/features/co
mpress"><method>zlib</method></compression><ver xmlns="urn:xmpp:features:rosterver" /><register xmlns="http://jabber.org/features/iq-register" /></stream:features>
DEBUG    SEND (IMMED): <starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls" />
DEBUG    RECV: <proceed xmlns="urn:ietf:params:xml:ns:xmpp-tls" />
DEBUG    Starting TLS
INFO     Negotiating TLS
INFO     Using SSL version: TLSv1
DEBUG    CERT: -----BEGIN CERTIFICATE-----
MIIDKDCCAhCgAwIBAgIIJaCB2QS3H2wwDQYJKoZIhvcNAQELBQAwHDEaMBgGA1UE
AwwRdXNlcnRwLXZpcnR1YWxib3gwHhcNMTgwNDA5MDkyOTQ5WhcNMjMwNDA4MDky
OTQ5WjAcMRowGAYDVQQDDBF1c2VydHAtdmlydHVhbGJveDCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAKVaBZT8n0OKXTgLVcBZKizKTD648jyi3biqKp62
SGqZcjvNGqg0VRCEMLeNZdcJvyjbyJikeq5HK8FDIs5z2uF6qESoVvpBDEeR27rv
+v1uG3Kae04cW1E5cxsUmrEhgsVastYYutPfEFVMjlg2fXb6mK72SASTiwj4oYVP
QWFmuwrJ8Zq3Nps7U0uRmsLVnXNrgj94Zai9FFqSQdSJtVopV9KFCENkfIggRd+Y
Qt94dn+9dBmayOJi/YVuKP7O81whbQgo0bbWYYmdLdu+PdBFZATw6cER/Z5wPaiI
19KTql3H/UOsl15cJHZgpv1MvwetfJtLF9URceVdmXokG3sCAwEAAaNuMGwwKgYD
VR0RBCMwIaAfBggrBgEFBQcIBaATDBF1c2VydHAtdmlydHVhbGJveDAdBgNVHQ4E
FgQU6HSFBHZhnwQCwpezCE/HstKOzVkwHwYDVR0jBBgwFoAU6HSFBHZhnwQCwpez
```



```
----------------------------------------
         XMPP Service Discovery
----------------------------------------
Device: alice@usertp-VirtualBox/1020
  - urn:xmpp:iot:control
  - urn:xmpp:iot:sensordata
DEBUG    SEND: <iq to="alice@usertp-VirtualBox/1020" from="bob@usertp-virtualbox/1020" id="1" type="get"><req xmlns="urn:xmpp:iot:sensordata" seqnr="1" momentary="true" /></iq>
DEBUG    Received disco info result from <alice@usertp-virtualbox/1020> to <bob@usertp-virtualbox/1020>.
DEBUG    Caching disco info result from <alice@usertp-virtualbox/1020> to <bob@usertp-virtualbox/1020>.
DEBUG    RECV: <iq to="bob@usertp-virtualbox/1020" type="result" id="1" from="alice@usertp-virtualbox/1020"><accepted xmlns="urn:xmpp:iot:sensordata" seqnr="1" /></iq>
DEBUG    Event triggered: disco_info
DEBUG    we got data accepted from alice@usertp-virtualbox/1020
DEBUG    RECV: <message to="bob@usertp-virtualbox/1020" from="alice@usertp-virtualbox/1020" xml:lang="en"><fields xmlns="urn:xmpp:iot:sensordata" seqnr="1" done="true"><node nodeId="1
020"><timestamp value="2023-04-04T17:02:17"><numeric unit="C" automaticReadout="true" name="Temperature" value="26" momentary="true" /></timestamp></node></fields></message>
DEBUG    we got data fields from alice@usertp-virtualbox/1020
----------------------------------------
          XEP 302 Sensor Data
----------------------------------------
DEBUG    RECV:[{'typename': 'numeric', 'unit': 'C', 'flags': {'momentary': 'true', 'automaticReadout': 'true'}, 'name': 'Temperature', 'value': '26'}]
Name            Type    Value   Unit
  - Temperature numeric 26      C

DEBUG    Event triggered: session_end
DEBUG    SEND (IMMED): </stream:stream>
DEBUG    End of stream recieved
INFO     Waiting for </stream:stream> from server
DEBUG    Stopped send thread. 2 threads remain.
DEBUG    Waiting for 2 threads to exit.
DEBUG    Threading deadlock prevention!
DEBUG    Marked event_thread_0 thread as ended due to disconnect() call. 1 threads remain.
DEBUG    Quitting Scheduler thread
DEBUG    Stopped scheduler thread. 0 threads remain.
DEBUG    Event triggered: disconnected
DEBUG     ==== TRANSITION connected -> disconnected
DEBUG    we got data done from alice@usertp-virtualbox/1020
DEBUG    Finished exiting event runner thread after early termination from disconnect() call. 0 threads remain.
DEBUG    State was not ready
DEBUG    ready ending
usertp@usertp-VirtualBox:~/xmpp$ █
```

3. What are the exchanged Stanzas between Bob and Alice? Identify those related to IOT and detail their content.

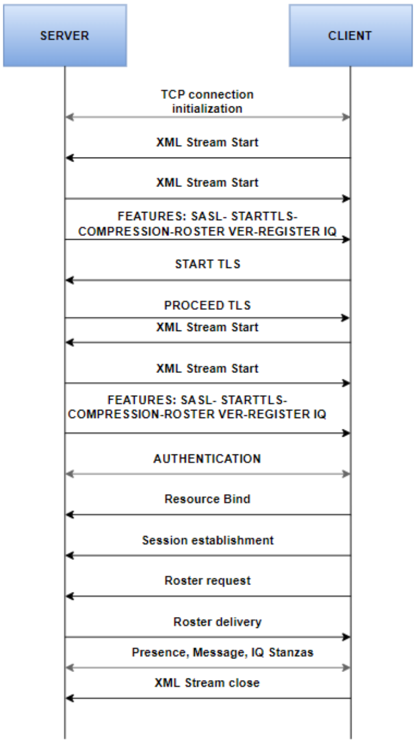
Using the debug option , -d on the server side we can see the stanza messages exchanged on the console.

Presence: This stanza is sent by Bob to announce its online status and capabilities to Alice. The stanza contains information about the entity sending the presence, its priority level, and any additional capabilities or features it supports.

Message: This stanza is sent by Alice to Bob, encapsulating the temperature data. The stanza contains the payload, which is the temperature data, and other metadata such as the recipient, sender, message type, etc.

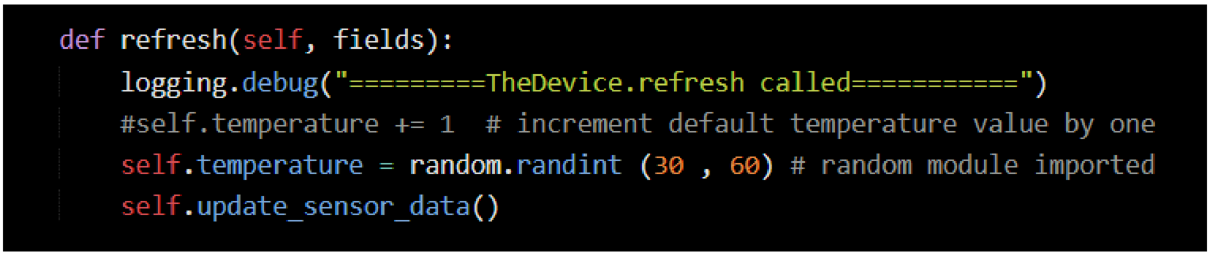
IQ: To get further information, a 1 payload query is sent which can be get, set, result or error.

4. Draw a workflow detailing the exchanged messages between Bob and Alice?



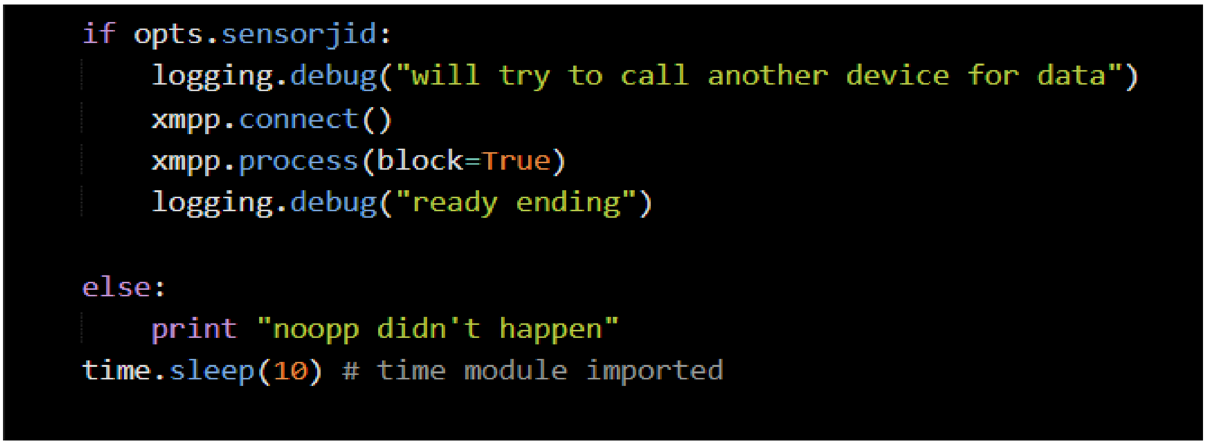
5. Modify the application to generate a random value of the temperature and allow a periodic request of the temperature.

Server:



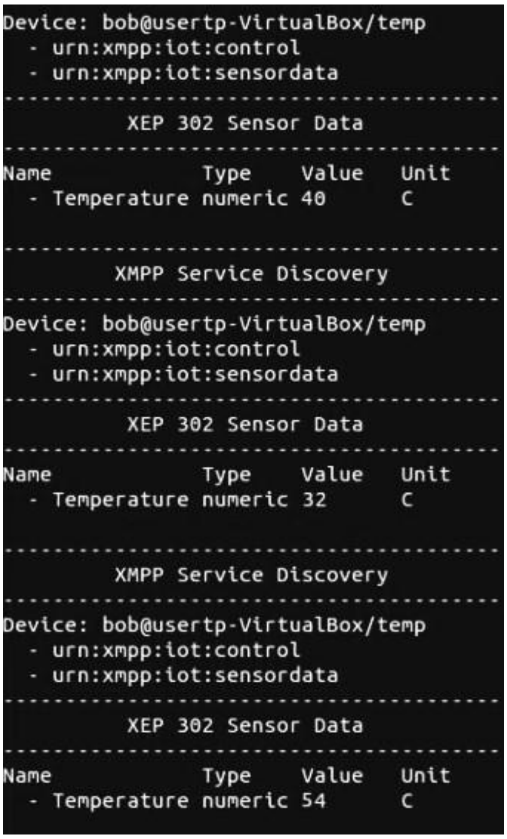
```
def refresh(self, fields):
    logging.debug("=========TheDevice.refresh called===========")
    #self.temperature += 1  # increment default temperature value by one
    self.temperature = random.randint (30 , 60) # random module imported
    self.update_sensor_data()
```

Client:



```
if opts.sensorjid:
    logging.debug("will try to call another device for data")
    xmpp.connect()
    xmpp.process(block=True)
    logging.debug("ready ending")

else:
    print "noopp didn't happen"
time.sleep(10) # time module imported
```



```
Device: bob@usertp-VirtualBox/temp
  - urn:xmpp:iot:control
  - urn:xmpp:iot:sensordata
----------------------------------------
          XEP 302 Sensor Data
----------------------------------------
Name            Type    Value   Unit
  - Temperature numeric 40      C

----------------------------------------
         XMPP Service Discovery
----------------------------------------
Device: bob@usertp-VirtualBox/temp
  - urn:xmpp:iot:control
  - urn:xmpp:iot:sensordata
----------------------------------------
          XEP 302 Sensor Data
----------------------------------------
Name            Type    Value   Unit
  - Temperature numeric 32      C

----------------------------------------
         XMPP Service Discovery
----------------------------------------
Device: bob@usertp-VirtualBox/temp
  - urn:xmpp:iot:control
  - urn:xmpp:iot:sensordata
----------------------------------------
          XEP 302 Sensor Data
----------------------------------------
Name            Type    Value   Unit
  - Temperature numeric 54      C
```